THE BASIL PROJECT

Programming in BASIL

A practical guide from first program to ownership, tasks and applications

Linux x86-64 · Maintained self-hosted implementation

Edition updated 12 November 2026

Contents

1. Your first program 3
2. Values, variables and expressions 5
3. Decisions and repetition 7
4. Functions and subroutines 9
5. Text and formatting 12
6. Records, alternatives and missing values 16
7. Lists, sets and dictionaries 19
8. Dense arrays and buffers 25
9. Ownership and borrowing, from first principles 27
10. Files, transfers and automatic cleanup 30
11. Checked errors and recovery 33
12. Modules, projects and packages 35
13. Generics, callbacks and custom value behavior 37
14. Coroutines and concurrent tasks 42
15. Channels, SELECT and timeouts 44
16. Console input, files and subprocesses 46
17. JSON, schemas and serialization 49
18. Dates, clocks, UUIDs and randomness 53
19. URLs, TCP and HTTP 55
20. Typed SQLite data 57
21. Binary formats, compression and numerical work 58
22. Building complete applications 61
23. Native interfaces and advanced library design 68
24. Diagnostics, limits and migration 70
25. Glossary and topic index 72
Library reference 74
String library reference 75
Format strings 78
Multiline strings 80
Collection library reference 81
Compiler-shipped collection families 86
Sequence operations and brackets 89
Initialized buffers, bulk I/O and allocation 91
Optional fallback and parameter defaults 93
Callback-dependent failure contracts 95
Local functions, callable values and visibility 96
Value equality, ordering, hashing and display 98
JSON 100
Shared serialization and custom format metadata 105
Recursive type metadata 108
Logging 110
Dates, times, durations and clocks 112
UUID values 114
Secure random bytes 117
URLs, paths and query parameters 118
HTTP and HTTPS 121
Regular expressions 124
Native BASIC compression 126
Coordinate and vector arithmetic 128
Unsigned numbers and bit operations 130
Floating point and math 131
BASIC terminal, file and process services 134
Commands, projects and libraries 136
Build-time embedded values 139
Text encoding 141
Byte buffers, text builders and streams 143
Native images and drawing 146
Native image format profiles 148
Native fonts and text 149
Language coverage 150
Public prelude declaration catalog 151

1. Your first program

BASIL is a compiled language for writing small utilities and substantial native applications. You write readable text, the compiler checks it, and the result is an executable program. You do not need to understand memory management to print a greeting. As your programs grow, the same language provides static types, structured errors, automatic resource cleanup and lightweight concurrent tasks.

This book describes the maintained self-hosted implementation, initially for Linux x86-64. It is not a compatibility manual for every historical BASIL. Read chapters 1 through 8 in order if programming is new to you. Chapters 9 through 15 explain the ownership and concurrency model. Later chapters build practical applications. The reference part supplies detailed library behavior and signatures.

Obtain a working compiler

From a source checkout, follow the bootstrap instructions. The canonical build is:

```
python3 tools/selfhost.py --report docs/selfhost-report.json
build/selfhost/stage3 --help
build/selfhost/stage3 toolchain
```

The bootstrap performs substantial validation. A successful build publishes `build/selfhost/stage3`. Throughout this book, shell commands use that path from the repository root. If your installation provides `basic` on PATH, substitute `basic`. Keep the compiler and its matching native SDK together; optional library features use that SDK. `toolchain` identifies the executable, target and native inputs.

Say hello

Save this complete program as `hello.bas`:

```
PRINT "Hello, world!"
```

Compile and run it:

```
build/selfhost/stage3 build hello.bas -o build/hello
./build/hello
```

The output is:

```
Hello, world!
```

`PRINT` writes text followed by a newline. Quotes delimit a string, which is a text value; the quotes themselves are not printed. The compiler's `build` command writes the executable and reports its path. The simpler one-step command is `build/selfhost/stage3 run hello.bas`.

The historical compiler interface is also supported: `stage3 hello.bas` writes executable bytes to stdout. Redirect those bytes to a file and check the command's exit status before executing it. For everyday use, prefer `build` and `run`.

Make the program your own

```
' An apostrophe starts a comment.
name = "Ada"
PRINT "Hello, " + name + "!"
PRINT "She said ""Hello""."
```

A variable gives a name to a value. The first assignment introduces `name` and infers its type as `STRING`. The plus signs concatenate strings. Double a quote inside a string to include a literal quote. Backslashes have no escape meaning: `"\n"` contains a backslash and an `n`.

Identifiers are ASCII case-insensitive, so `Name` and `name` identify the same variable. Strings and source files use UTF-8. Indentation helps readers; block keywords such as `END IF` determine structure. Put statements on separate lines.

Entry points and exit status

Top-level executable statements, as above, are convenient for small programs. Alternatively, supply a `SUB Main()` or a zero-argument `INTEGER FUNCTION Main()`. Choose one entry style per program.

```
FUNCTION Main() AS INTEGER
    PRINT "Ready"
    RETURN 0
END FUNCTION
```

An integer Main returns a process exit status: zero conventionally means success. Functions introduced later can declare checked failures. An unhandled root error is reported after recoverable cleanup and exits with status 1.

Read a compiler message

```
count = 3
count = "three"
```

This example is intentionally rejected. A variable's static type does not change when you assign again. Find the reported file and location, then read what operation and types conflict. Fix the first useful error and rebuild. Do not assume that a later message describes an independent problem.

Try changing the greeting, adding another `PRINT`, and deliberately removing a closing quote. Seeing a small error in a familiar program is a good way to learn the compiler's diagnostic style.

How examples are checked

Most teaching examples are complete programs with output checked by the book build. Deliberately rejected examples are tested for a diagnostic. Excerpts, interactive programs and examples requiring a server, GUI or native dependency are explicitly described as such. Detailed build and example records accompany the editable source. Examples establish the behavior they demonstrate, rather than proving every property of the language.

2. Values, variables and expressions

A type describes what a value represents and which operations make sense for it. BASIL checks types before the program runs. This catches mistakes such as adding a customer's name to an invoice total without an explicit conversion.

Declare and initialize variables

```
LET quantity = 3
DIM price AS DOUBLE = 2.5
DIM description AS STRING
description = "notebook"
quantity += 1
PRINT description
PRINT quantity
PRINT price
```

`LET` is optional on a first assignment. `DIM ... AS ...` supplies a type explicitly. It is especially useful for empty collections and optionals, where the initializer does not provide an element type. A declaration without a value requires proven initialization before a read. Do not expect an ordinary `DIM count AS INTEGER` to initialize a local to zero. Dense array cells have separate default rules.

Blocks introduce lexical scope: a variable declared inside a branch is local to that branch. Use an outer typed declaration when multiple branches must initialize one variable used afterward. The compiler checks that every reaching path assigns it.

The scalar types

Type	Use
`INTEGER`	Signed whole numbers from −2^63 through 2^63 - 1
`DOUBLE`	Binary64 floating-point measurements and fractional values
`BOOLEAN`	`TRUE` or `FALSE`
`BYTE`	Unsigned numbers from 0 through 255; packed binary data
`UINT32`	Unsigned 32-bit values
`UINT64`	Unsigned 64-bit values
`STRING`	Valid UTF-8 text

Decimal whole literals normally have type INTEGER. A decimal point or exponent produces DOUBLE: `1.0`, `2.5`, `1.0e3`. Hexadecimal `&H`, binary `&B` and octal `&O` literals have UINT64 type. Numeric types do not generally convert implicitly. Use explicit conversions such as `CDBL(count)`, `BYTE(255)` or `UINT32(7)`. Conversions check the target's range; do not use conversion as an overflow workaround.

```
PRINT 7 / 2
PRINT 7 MOD 2
PRINT 7.0 / 2.0
PRINT CDBL(3) + 0.5
PRINT 2 + 3 * 4
PRINT (2 + 3) * 4
```

INTEGER division truncates toward zero. Parentheses make grouping explicit. INTEGER and unsigned arithmetic check overflow and invalid division; these fatal runtime checks are distinct from catchable operational failures. DOUBLE has a finite precision, so decimal fractions such as 0.1 are generally approximate. Use an application's chosen representation and rounding policy for money.

Operators and conditions

= and == test equality in an expression; <> means unequal. Numeric ordering uses <, <=, > and >=. Assignment is selected by statement context, as in `count = 3`. `AND` and `OR` short-circuit: the right operand runs only when needed. `NOT` negates a Boolean. `&`, `|` and `XOR` provide integer bit operations; Boolean XOR is also supported. ^ is not an XOR spelling.

From lowest to highest precedence: optional fallback `??`; OR; AND; comparisons; `+ - | XOR`; `* / MOD &`; unary `+ - NOT`; primary expressions and postfix access. Use parentheses whenever the intended grouping might surprise a reader.

```
count = 4
ready = count > 0 AND count < 10
PRINT ready
PRINT NOT ready
PRINT count <> 0
```

Conditions require BOOLEAN. Zero, an empty string, an empty collection and NONE are not Boolean false values. Write `LEN(items) > 0` or use `IF LET` for an optional. This makes a missing value distinguishable from a present zero or empty string.

Constants

```
CONST MinutesPerHour = 60
CONST Greeting AS STRING = "Hello"
CONST Squares AS LIST OF INTEGER = [0, 1, 4, 9]
PRINT MinutesPerHour
PRINT Greeting
PRINT Squares[3]
copy = Squares
copy[0] = 99
PRINT Squares[0]
```

CONST names a compile-time value. A scalar table is an immutable flat list of scalars or one enum type. It supports reads, iteration and slicing without mutable access to its storage. Assigning it to an ordinary owned variable creates an independent value. It cannot be passed BYREF or moved with TAKE. Constants cannot execute arbitrary functions or initialize arbitrary nested owned objects.

Nonfinite floating-point results

Floating operations can produce NaN (not a number) or infinity. NaN is unequal even to itself; signed zeros compare equal. Use `IsNaN`, `IsInfinite` or `IsFinite` to classify results, and `Finite` when a nonfinite result should be a checked error.

```
PRINT IsNaN(SQRT(-1.0))
PRINT IsInfinite(EXP(1000.0))
TRY
    PRINT Finite(SQRT(-1.0))
CATCH problem AS NonFiniteError
    PRINT "A finite result was required"
END TRY
```

Floating results are not automatically operational errors. Integer conversions and integer-result operations still need a representable result. See the floating-point reference for exact policies.

3. Decisions and repetition

Programs become useful when they choose what to do and repeat work over data. BASIL provides structured blocks so the start, end and cleanup of each operation are visible.

Choose a branch

```
temperature = 18
IF temperature < 10 THEN
    PRINT "Take a coat"
ELSEIF temperature < 20 THEN
    PRINT "Take a sweater"
ELSE
    PRINT "Enjoy the sunshine"
END IF
IF temperature >= 0 THEN PRINT "Above freezing"
```

The first true branch runs. An ELSE handles all remaining cases. A single-line IF is useful for one simple statement; use a block for multiple operations or nested control flow. An assignment in one branch does not prove a variable initialized in all branches. Declare and initialize shared results deliberately.

Count with FOR

```
total = 0
FOR number = 1 TO 4
    total += number
NEXT
PRINT total
FOR remaining = 3 TO 1 STEP -1
    PRINT remaining
NEXT
```

TO includes the final value. STEP defaults to 1. Start, end and step evaluate once when the loop begins, so changing a variable used in the end expression does not change that loop's saved end. The counter is available after NEXT. A zero step is invalid; DOUBLE loops also reject nonfinite or stalled progress. Prefer INTEGER when you need an exact number of iterations.

Loop while a condition holds

```
remaining = 3
WHILE remaining > 0
    PRINT remaining
    remaining -= 1
WEND
DO
    remaining += 1
LOOP UNTIL remaining = 2
PRINT remaining
```

WHILE tests before each iteration and can run zero times. A footer-tested DO runs at least once. DO also permits a WHILE or UNTIL header; use either a header test or a footer test, rather than both. UNTIL stops when its condition becomes true. Ensure some operation can change the condition, or the loop will never end.

Visit elements

```
names = ["Ada", "Grace", "Mary"]
FOR EACH name IN names
    PRINT name
NEXT
```

FOR EACH avoids manual indexing. Its bindings are read-only. The collection stays protected against invalidating mutation while it is being visited, including when the body calls a helper or suspends. Build a separate result when filtering or transforming data. Dictionaries can bind both key and value.

Use `EXIT FOR`, `EXIT WHILE` or `EXIT DO` to leave that kind of loop early. Leaving a block releases its owned locals. There is no general CONTINUE or GOTO; use an IF around work that should run only for selected iterations.

Select a value

```
score = 85
SELECT CASE score
CASE 90 TO 100
    PRINT "Excellent"
CASE 70 TO 89
    PRINT "Passed"
CASE IS < 0
    PRINT "Invalid"
CASE ELSE
    PRINT "Try again"
END SELECT
```

The first matching case runs; there is no fallthrough. Cases can list values separated by commas, inclusive ranges, or relational tests. Put overlapping cases in a deliberate order. `SELECT TYPE` matches a closed choice's alternatives; scheduler `SELECT` waits for task/channel/timer events. Those are separate forms, introduced in chapters 6 and 15.

4. Functions and subroutines

A function gives a name to an operation and returns a result. A subroutine performs work without a successful result. Both let you divide a program into pieces you can understand and reuse.

Define an operation

```
PRINT Area(3.0, 4.0)
ShowGreeting("Ada")
FUNCTION Area(width AS DOUBLE, height AS DOUBLE) AS DOUBLE
    RETURN width * height
END FUNCTION
SUB ShowGreeting(name AS STRING)
    PRINT "Hello, " + name
END SUB
```

Parameters name the inputs. `AS DOUBLE` after the parameter list names the function's successful result type. Every reachable successful path in a FUNCTION must return a value of that type. A SUB can reach END SUB or use bare RETURN; `EXIT SUB` also returns successfully. A function cannot use bare RETURN or `EXIT FUNCTION`. Checked failure is a separate exit path, explained in chapter 11. An expression-bodied function is convenient for a short calculation:

```
PRINT Square(5)
FUNCTION Square(value AS INTEGER) AS INTEGER = value * value
```

Defaults and named arguments

```
PRINT Greeting("Ada")
PRINT Greeting(prefix="Welcome, ", name="Grace")
FUNCTION Greeting(name AS STRING, prefix AS STRING = "Hello, ") AS STRING
    RETURN prefix + name
END FUNCTION
```

Required parameters precede optional ones. Defaults use checked constant values and construction recipes, rather than arbitrary runtime calls. Positional arguments precede named arguments; a name cannot be supplied twice. Supplied expressions evaluate once in written order; omitted defaults follow. Naming arguments changes which parameter receives a value, not the order in which supplied expressions execute.

Default records and collections

Defaults can construct records, choices, typed errors, nested lists, empty collections, SOME/NONE and scalar tables. Only omitted arguments construct their defaults, after supplied arguments have evaluated. Names resolve in the function's declaring module; caller locals cannot change which constructor or constant a default means. Arbitrary function calls and mutable variables are not defaults.

```
TYPE Config
    Limit AS INTEGER = 20
    Names AS LIST OF STRING = []
END TYPE
FUNCTION Label(config AS Config = Config { Limit=20, Names=["default"] }) AS STRING
    RETURN config.Names[0] + ": " + STR(config.Limit)
END FUNCTION
PRINT Label()
PRINT Label(Config { Limit=42, Names=["custom"] })
```

Use an ordinary default when omission means a known configuration. Use an optional when absence is itself meaningful or a fallback must run ordinary code. BYVAL defaults create independent mutable owners on each call:

```
FUNCTION ExpandedSize(BYVAL values AS LIST OF INTEGER = [1, 2]) AS INTEGER
    Append(BYREF values, 3)
    RETURN LEN(values)
END FUNCTION
PRINT ExpandedSize()
PRINT ExpandedSize()
```

Both calls print 3. There is no persistent shared default list. Fresh resource literal defaults are constructed and cleaned normally, without copying an existing owner. BYREF and TAKE defaults remain rejected. Task entries use the same checked recipes. Generic recipes are checked when their concrete omitted value is needed.

Normalize an optional parameter locally

An explicit DIM can reuse the name of an ordinary or BYVAL parameter. The initializer reads the original parameter; subsequent statements see the new local, which may have a different type. This is called shadowing. Combined with the lazy `??` fallback from chapter 6, it gives a convenient configuration pattern:

```
TYPE Config
    Limit AS INTEGER = 20
    Names AS LIST OF STRING = []
END TYPE
FUNCTION Run(config AS OPTIONAL OF Config = NONE) AS INTEGER
    DIM config = config ?? Config { Limit=20 }
    Append(BYREF config.Names, "local")
    RETURN config.Limit + LEN(config.Names)
END FUNCTION
given = SOME(Config { Limit=40, Names=["original"] })
PRINT Run()
PRINT Run(given)
IF LET original = given THEN
    PRINT LEN(original.Names)
END IF
```

The local change leaves the caller's value independent. Leaving a nested block restores the original parameter's visibility. BYREF and TAKE parameters cannot be shadowed, and duplicate locals in one scope remain errors. Shadowing does not extend the original parameter's borrowed-return lifetime.

Grant writable access explicitly

Default parameters borrow read-only access. Reading a large list parameter does not automatically deep-copy the list. A writable operation requires BYREF at the declaration and the call site:

```
count = 10
Increment(BYREF count)
PRINT count
SUB Increment(BYREF value AS INTEGER)
    value += 1
END SUB
```

BYREF gives the function exclusive writable access for the call. Passing the same storage through conflicting arguments is rejected. BYVAL creates a writable independent local value instead. Chapter 9 explains when to choose each mode.

Overloads and local helpers

Functions may share a name when their signatures let the compiler choose a unique operation. Numeric conversions are explicit; an INTEGER overload and a DOUBLE overload do not make those types interchangeable. Result context can help resolve some otherwise incomplete calls, but the compiler rejects genuine ambiguity.

A local FUNCTION or SUB keeps a helper inside its lexical block. Declare it before use. It does not capture surrounding local variables; pass needed data as parameters. For a function value that snapshots outer copyable data, use an anonymous lambda rather than a named local declaration. Chapter 13 explains both styles and their different capture rules.

Recursion is supported, but stacks are finite. For data whose depth comes from untrusted input, consider an explicit list of pending work and a checked size limit rather than assuming arbitrarily deep recursive calls will fit.

5. Text and formatting

STRING is valid UTF-8 text. Its contents are immutable: an operation returns a new string, and a variable can be assigned that new value. This keeps text operations predictable even when other code has read access to the original.

Positions count Unicode scalars

```
text = "Aé🌍"
PRINT LEN(text)
PRINT ByteCount(text)
PRINT MID(text, 1, 1)
PRINT text[-1]
PRINT text[0:2]
```

The length is 3 Unicode scalar values but 7 UTF-8 bytes. Positions are zero-based; negative positions count backward from the end. A slice includes its start and excludes its finish. A visible character can contain multiple scalars, such as a letter followed by a combining accent. Scalar positions are not display columns or grapheme-cluster positions. Use explicit byte APIs for binary protocols.

`LEFT`, `MID`, `RIGHT`, search, trim, case conversion, padding and replacement are provided by the string reference. `$` aliases such as MID$ exist, but arbitrary dollar-suffixed variable names are not supported.

Write text across several source lines

Triple double quotes introduce a block literal when the opening quotes are immediately followed by a newline. This is useful for SQL, templates, messages and sample data that would otherwise need many quoted fragments.

```
name = "Ada"
message = DEDENT TRIM f"""
    Hello, {name}!
        Ticket: {42:04d}
"""
PRINT message
```

Output:

```
Hello, Ada!
    Ticket: 0042
```

The raw block preserves indentation and original newline bytes. DEDENT removes the exact common space/tab prefix of nonblank content lines. TRIM removes only boundary newline sequences; it does not strip horizontal whitespace. Together, they dedent first and then trim. Either modifier order is accepted, but duplicate modifiers are errors. Ordinary `TRIM(text)` remains a separate whitespace-trimming function.

Use `f"""` for interpolation and plain triple quotes for literal braces. Layout processing happens before interpolation, so indentation inside a substituted value is preserved. Interpolation expressions themselves must remain single-line. The opening and closing delimiter lines' whitespace matters: closing indentation is content unless removed by the layout rules. Tabs are literal characters, independent of the editor's display width.

The newline after the opener distinguishes a block from existing doubled-quote syntax: `"""word"""` still means an ordinary string containing quoted `word`. Six consecutive quotes inside a block represent three literal quotes. No backslash escapes are added. Blocks work in arguments, checked constant defaults and FORMAT options under the usual constant-expression rules. The multiline-

string reference specifies blank lines, quote runs and CR/LF preservation. The IDE highlights blocks across scrolling and edits without treating their contents as new declarations.

Split, join and search

```
words = SPLIT("red,green,blue", ",")
PRINT LEN(words)
PRINT JOIN(words, " / ")
PRINT UPPER("hello")
PRINT TRIM("  BASIC  ")
PRINT LEN(SPLIT("Aé"))
```

SPLIT with a nonempty separator splits on that exact text. SPLIT with no separator, or an empty separator, produces one string per scalar; it is not whitespace splitting. Empty input produces an empty list in that form. For line processing, use the dedicated LINES API and its documented line-ending behavior.

Exact string equality compares the encoded text; it does not silently normalize Unicode or apply locale rules. UPPER/LOWER and case folding are distinct from locale-aware collation. To compare text for an application's identifiers, choose and document the normalization/case policy you require.

Format output

```
name = "Ada"
count = 42
price = 12.3456
PRINT f"Hello, {name}!"
PRINT f"Count: {count:04d}"
PRINT f"Price: {price:.2f}"
PRINT f"{{literal braces}}"
```

The `f` must immediately precede the opening quote. Expressions inside braces evaluate once from left to right. A colon introduces a static format specification. Common presentations are `d` for decimal integers, `x` for hexadecimal, `f` for fixed decimals, `e` for scientific notation and `%` for percentages. Width is a minimum; `.2f` requests two fractional decimal places. Strings support alignment and a scalar precision limit.

Specifications support `<`, `>`, `^` alignment, fill, sign, base prefixes, zero padding and grouping. Read the included format-string reference for the complete mini-language. Dynamic nested width/precision, debug `=`, custom record formatting and locale presentation are not implemented. Invalid specifications are compiler errors, including in unused functions.

For repeated construction, use StringBuilder and append operations from the buffer reference rather than repeatedly concatenating an ever larger prefix. Only initialized content is visible; reserving capacity is not creating text.

Regular expressions

```
pattern = CompileRegex("[0-9]+")
IF LET found = RegexFind(pattern, "item 42") THEN
    PRINT found.Text
    PRINT found.Start
END IF
PRINT RegexReplace("[0-9]+", "item 42", "number")
```

Compile once when using a pattern repeatedly. The compiled Regex is an owned resource, while matches and captured text are independent values. String-pattern overloads compile on each call. The engine is now implemented in BASIL itself; no Rust runtime or separate regex library is needed. The default Native dialect retains the pinned Rust-regex syntax baseline: Unicode classes

and named captures are supported, but lookaround and backreferences reject. Choose the EcmaScript dialect explicitly for those constructs.

```
PRINT RegexIsMatch("^([a-z]+)-\1$", "blue-blue", Dialect=RegexDialect.EcmaScript)
PRINT RegexIsMatch("^(?!admin$)[a-z]+$", "user", Dialect=RegexDialect.EcmaScript)
PRINT RegexIsMatch("^ada$", "ADA", Flags="i")
```

Dialect changes pattern matching, not replacement syntax. Both dialects expand `$1`, `${name}` and `$$`; these are separate from BASIL interpolation. EcmaScript is the stateless ECMAScript 2025 Unicode-mode pattern profile, not a JavaScript RegExp object. Flags d/g/v/y are rejected. Its backreferences have no linear-time guarantee. Configure RegexLimits for untrusted work; limits measure logical work, memory and results rather than wall-clock deadlines.

```
TRY
    limits = RegexLimits {MaxPatternBytes=3}
    pattern = CompileRegex("abcd", Limits=limits)
CATCH exhausted AS RegexLimitError
    PRINT "Pattern limit reached"
END TRY
```

All regex operations declare RegexError, including searches with a compiled pattern. A limit failure is an error, never FALSE, NONE or a partial success. Syntax and unsupported-profile failures have specific error types too. Normal compiler scheduling and cancellation apply during native regex work. The regex reference explains captures, empty matches, flags and every limit.

Convert external encodings

A STRING always contains valid UTF-8. A legacy file is bytes until you explicitly decode it; opening a file does not guess its encoding. Use ReadBinaryFile with DecodeText, then EncodeText before writing external bytes. No import is required.

```
bytes = EncodeText("café", TextEncoding.Windows1252)
PRINT LEN(bytes)
PRINT DecodeText(bytes, TextEncoding.Windows1252)
result = EncodeTextWithInfo("a€😀", TextEncoding.ASCII, TextErrorPolicy.Replace)
PRINT DecodeUtf8(result.Bytes)
PRINT result.Info.Replacements
```

Strict conversion is the default: malformed bytes and unrepresentable characters raise TextEncodingError. Replace is an explicit lossy policy: decoding inserts U+FFFD; legacy encoding inserts `?`. WithInfo returns output plus counters, so a successful replacement need not silently hide lost characters. Error Offset and Length count source bytes, including UTF-8 bytes when encoding a STRING; they are not the scalar indexes used by ordinary string operations.

```
TRY
    text = DecodeText([BYTE(255)], TextEncoding.UTF8)
CATCH err AS TextEncodingError
    PRINT err.Offset
    PRINT err.Length
END TRY
```

Profiles cover UTF-8, explicit-endian UTF-16/32, ASCII, Latin1, Windows1252, CP437, ShiftJIS, Big5, EUCKR and separate GB18030_2000/GB18030_2022 versions. Choose the version required by your data: these are precise mapping profiles, not aliases for every regional Windows codec. No detection, normalization or newline conversion is implicit. The reference lists the exact profiles.

BOM handling is explicit. Decode defaults to Preserve; Consume removes a matching initial BOM and Require demands it. Neither changes the selected byte order. Encode defaults to Omit; Emit writes one initial BOM, even for empty text. These BOM options reject legacy encodings.

maxOutputBytes defaults to 64 MiB per call or chunk and counts emitted bytes; it does not bound an entire stream.

Decode text arriving in chunks

A network read can split a character between packets. A TextDecoder retains the unfinished sequence until the next chunk rather than treating each read as a complete document.

```
decoder = NewTextDecoder(TextEncoding.UTF16LE)
first = DecodeTextChunk(BYREF decoder, [BYTE(65)])
PRINT LEN(first)
PRINT DecodeTextChunk(BYREF decoder, [BYTE(0)], final=TRUE)
PRINT DecoderInfo(decoder).Finished
```

FinishTextDecoder supplies the final empty chunk when the input ends separately. NewTextEncoder, EncodeTextChunk and FinishTextEncoder provide the other direction. A converter finishes exactly once; later chunk/finish calls fail. Converters are copyable opaque values: copying one forks its state independently. BYREF gives exclusive access while advancing it. A recoverable chunk failure leaves that state unchanged and returns no partial chunk; already written earlier chunks remain written. Info snapshots contain cumulative byte counters, replacement counts and the first replacement offset. Keep an application-level total limit when processing untrusted streams.

6. Records, alternatives and missing values

Use a record when several values describe one thing. Use an enum when a value is one of a fixed set of names. Use a closed choice when different alternatives carry different kinds of data. Use an optional when an otherwise ordinary value may be absent. These representations let the compiler check your assumptions.

Records

```
TYPE Customer
    Name AS STRING
    Visits AS INTEGER = 0
    Notes AS LIST OF STRING = []
END TYPE
customer = Customer { Name="Ada" }
copy = customer
copy.Visits += 1
Append(BYREF copy.Notes, "Called today")
PRINT customer.Visits
PRINT LEN(customer.Notes)
PRINT copy.Visits
```

Named construction supplies fields in any written order. Every field needs an explicit value or a supported declared default. Explicit values evaluate in written order, then omitted defaults in declaration order. Empty collection defaults create independent collections for each record. Positional construction, such as `Customer("Ada")`, follows field declaration order and may use trailing defaults. Named function arguments are not a second spelling for record fields.

Ordinary copyable records compare their fields recursively. If a field owns a file or another resource, the containing record becomes noncopyable. A record with a cleanup hook also becomes a resource; chapter 10 explains that change.

Enums

```
ENUM Priority
    Low
    Normal
    High = 10
END ENUM
priority = Priority.High
PRINT EnumValue(priority)
PRINT priority = Priority.High
PRINT Priority(10) = priority
```

Members are qualified by the enum name. Values start at zero and increment unless explicitly assigned. Enums are distinct static types, not aliases for INTEGER. `EnumValue` extracts the number; construction from INTEGER checks membership and can fail with NumericError. Duplicate values, aliases and implicit numeric conversion are not provided.

Optionals

```
DIM nickname AS OPTIONAL OF STRING = NONE
IF LET text = nickname THEN
    PRINT text
ELSE
    PRINT "No nickname"
```

```
END IF
nickname = SOME("")
IF LET text = nickname THEN
    PRINT "Present, length="; LEN(text)
END IF
```

NONE means absence. SOME wraps a present value, even if that value is empty, zero or FALSE. IF LET checks presence and introduces a branch-local independent binding for a copyable payload. A borrowed stable optional supplies a read-only copy; a fresh result or `TAKE optional` supplies a writable owner. You cannot access an absent payload. Nested optionals distinguish outer absence from `SOME(NONE)`, which is useful when a dictionary contains an explicitly absent value or a task successfully returns NONE.

Resources have different IF LET ownership rules: a named optional loans its payload; a fresh or explicitly moved optional can provide an owned payload. Chapter 10 shows both.

Choose a fallback only when absent

`optional ?? fallback` unwraps one optional layer. A present value wins even if it is zero, FALSE, empty text or an empty collection. Only NONE evaluates the fallback:

```
PRINT SOME(0) ?? DefaultNumber()
DIM absent AS OPTIONAL OF INTEGER = NONE
PRINT absent ?? DefaultNumber()
PRINT "[" + (SOME("") ?? "missing") + "]"
FUNCTION DefaultNumber() AS INTEGER
    PRINT "Calculating fallback"
    RETURN 99
END FUNCTION
```

Output is 0, then Calculating fallback and 99, then empty brackets. This differs from dictionary Get with a fallback, whose ordinary argument evaluates eagerly. A fallback can fail through normal checked contracts; its contract is checked even when a particular execution skips it.

The operator associates to the right and binds below OR, AND, comparisons and arithmetic. Use explicit parentheses when unwrapping nested optionals, such as `(outer ?? SOME(7)) ?? 9`. A present outer SOME(NONE) is still present at the first step. A fresh resource optional can donate its payload; a named resource optional needs TAKE for ownership transfer. No implicit move or resource copy is introduced. Borrowed copyable payloads produce independent values when ownership is needed. See optional fallback and defaults for parameter rules.

Closed choices

```
TYPE Reply
    CASE Empty
    CASE Text AS STRING
    CASE Number AS INTEGER
END TYPE
reply = Reply { Text="Ready" }
SELECT TYPE reply
CASE Reply.Empty
    PRINT "No reply"
CASE Reply.Text AS text
    PRINT text
CASE Reply.Number AS number
    PRINT number
END SELECT
```

A choice holds exactly one declared alternative. SELECT TYPE checks the alternatives, with independent case-local copyable payload bindings. Cover all alternatives or provide the supported fallback form. Projecting a payload, such as `reply.Text`, returns an optional; projecting a payload-

free alternative returns a Boolean. Alternative fields cannot be assigned directly: assign a complete new choice instead.

Records and choices may form finite recursive values through supported lists, dictionaries and optionals. A direct inline containment cycle is invalid. Recursive value trees do not imply general shared reference graphs or cyclic back-references.

Record extension and opaque types

```
TYPE Person
    Name AS STRING
END TYPE
TYPE Customer EXTENDS Person
    Visits AS INTEGER = 0
END TYPE
customer = Customer { Name="Ada", Visits=2 }
ShowName(customer)
base = Person(customer)
PRINT base.Name
SUB ShowName(person AS Person)
    PRINT person.Name
END SUB
```

EXTENDS reuses fields and permits checked base access. It does not introduce classes, virtual methods, downcasts or runtime subtype identity. Multiple bases are allowed when layouts have no collisions or repeated ancestors. A default base parameter borrows the actual value; `Person(customer)` explicitly constructs an independent base value. BYREF base access preserves the derived fields and cannot replace the complete base storage.

OPAQUE TYPE is a module boundary: clients can name and own values but cannot construct or extend the record or access hidden fields. Explicit PUBLIC fields are readable outside the defining module, including independently copying copyable fields. They do not grant external mutation, BYREF, SWAP or TAKE. Public field visibility is currently for ordinary opaque records; arbitrary member visibility on ordinary transparent records is not supplied. Chapter 12 shows the module pattern.

7. Lists, sets and dictionaries

Choose a list for an ordered sequence that can grow, a set for unique members, and a dictionary for values addressed by keys. Every collection has statically known types. A list cannot quietly change from integers to unrelated strings.

Lists

```
numbers = [3, 1, 2]
Append(BYREF numbers, 4)
PRINT numbers[0]
PRINT numbers[-1]
PRINT LEN(numbers)
ordered = Sorted(numbers)
PRINT ordered[0]
PRINT numbers[0]
Sort(BYREF numbers)
PRINT numbers[0]
```

List indices start at zero; −1 is the last element. Empty lists need a type context: `DIM names AS LIST OF STRING = []`. LIST() without context constructs an INTEGER list. List literals evaluate elements left to right.

`Sort` and `Reverse` mutate a list and require BYREF. `Sorted` and `Reversed` produce independent values. They also have string forms where applicable; strings do not have mutating Sort/Reverse overloads. Default sorting is stable. Custom comparison functions are covered in chapter 13.

`Insert`, `RemoveAt` and `ExtractAt` use checked positions. `Pop` removes and returns the last value, reporting EmptyCollection if empty. `Remove` removes the first equal value; `Clear` destroys contents while retaining capacity. `Extend` appends a sequence, and `Concat` returns a combined independent sequence. Do not pass the same list as both a BYREF destination and a borrowed source; explicitly copy the source if you need its snapshot.

Slices

```
values = [0, 1, 2, 3, 4]
middle = values[1:4]
PRINT JOIN([STR(middle[0]), STR(middle[-1])], ",")
backwards = values[::-1]
PRINT backwards[0]
values[1:3] = [8, 9, 10]
PRINT LEN(values)
```

Slices use a half-open start:finish interval, an optional stride, omitted bounds and negative indices. Copyable slices are independent values. List slice replacement can change length for a unit stride; extended-slice replacement must meet the reference's length rules. A zero stride is invalid. STRING slices return text; STRING element/slice assignment is rejected.

Direct LIST indexing has a fatal bounds policy. Some named editing and slicing APIs report checked IndexError instead. Do not assume every out-of-range operation is catchable; choose the operation appropriate to your input validation policy.

Sets

```
DIM names AS SET OF STRING = SET()
PRINT Insert(BYREF names, "Ada")
PRINT Insert(BYREF names, "Ada")
PRINT Contains(names, "Ada")
PRINT LEN(names)
```

Insertion reports whether a new member was added. Sets retain insertion order for iteration, although set equality ignores order. Union, Intersection, Difference and SymmetricDifference create new sets. Removal followed by reinsertion places a member last. Resource members are not supported.

Dictionaries

```
DIM counts AS DICT OF STRING TO INTEGER = DICT()
counts["apples"] = 2
counts["apples"] += 1
PRINT Get(counts, "apples")
PRINT Get(counts, "oranges", 0)
IF LET count = Find(counts, "oranges") THEN
    PRINT count
ELSE
    PRINT "Not recorded"
END IF
FOR EACH name, count IN counts
    PRINT name; ":"; count
NEXT
```

Put or bracket assignment inserts/replaces a value. Replacement keeps the key's position; removal/reinsertion moves it last. Get without a fallback reports MissingKey. Get with a fallback evaluates that fallback eagerly, even when the key exists, and does not insert. SetDefault inserts when missing. Find returns an optional and therefore distinguishes a missing key from a stored zero. Keys and Values return independent copyable snapshots. Extract removes an entry and returns its value in an optional; Remove destroys it.

Keys need coherent equality and hashing. NaN is rejected as a key/member, including inside derived composite keys; infinity is allowed and signed zeros hash equally. Resource dictionary values are supported, with ownership rules in chapter 10; resource keys are not.

Query collections with predicates

```
values = [1, 2, 3, 4]
PRINT Any(values, Even)
PRINT All(values, Even)
PRINT CountWhere(values, Even)
FOR EACH value IN Filter(values, Even)
    PRINT value
NEXT
FUNCTION Even(value AS INTEGER) AS BOOLEAN = value MOD 2 = 0
```

A predicate is a function returning BOOLEAN. Any stops on the first true result; All stops on the first false result. CountWhere visits all entries; Filter builds an independent result in source order. List searches also include IndexWhere, LastIndexWhere and FindWhere. Dictionary predicates receive key and value. These predicates must be infallible. Named functions, checked callable values and contextually typed lambdas are supported. The source stays protected while a callback runs, including if it suspends.

Choose a collection for the job

The prelude also supplies generic collection records. They need no import or application package dependency. Choose their types explicitly at empty construction: `DIM jobs AS Queue[STRING] = NewQueue()`. A later Enqueue cannot infer the type of an earlier untyped empty factory. Use **Length** for these records' logical count; LEN remains the operation for core lists, sets, dictionaries and arrays.

Need	Type	Main operations
Add/remove at either end	DequeT	PushFront/PushBack, Front/Back, PopFront/PopBack
First-in, first-out work	QueueT	Enqueue, Front, Dequeue
Keep only a fixed number of recent items	BoundedDequeT	Deque operations with optional eviction
Process smallest priorities first	PriorityQueueT	Push, Peek, PeekPriority, PopPriority
Visit unique keys in sorted order	SortedSetT	InsertValue, Has, At, Range
Look up values with sorted keys	SortedDictionary[K,V]	SetValue, Lookup, LookupRequired
Count occurrences	CounterT	Add, Count, TotalCount
Retain several values per key	MultiMap[K,V]	Add, GetAll, ExtractAt
Track membership in a fixed integer universe	BitSet	Include, Exclude, Flip, Members
Retain a bounded working set	Cache[K,V]	CachePut, CacheFind, CacheExtract

Reset empties a collection without changing its configuration. Copies and copyable snapshots are independent. Insertion takes an owned BYVAL payload: borrowed copyable inputs are copied, fresh values are adopted, and named resources need TAKE. Resource Front/Peek/Lookup-style results borrow the stored owner; removal/extraction transfers it. The entire collection stays protected through consumption, including suspension. Snapshot operations that would copy resources are rejected.

Queues, deques and bounded histories

```
DIM jobs AS Deque[STRING] = NewDeque()
PushBack(BYREF jobs, "build")
PushFront(BYREF jobs, "check")
PRINT PopFront(BYREF jobs)
PRINT PopBack(BYREF jobs)
DIM queue AS Queue[INTEGER] = NewQueue()
Enqueue(BYREF queue, 10)
Enqueue(BYREF queue, 20)
PRINT Dequeue(BYREF queue)
PRINT Length(queue)
```

A deque uses a circular buffer, with amortized constant-time end operations. A queue exposes FIFO access over that storage. Empty reads/removals raise EmptyCollection. Snapshot returns the logical sequence, rather than physical buffer order, and requires copyable payloads.

A bounded deque has a positive capacity. Pushing into a full end returns the evicted value from the opposite end in SOME; a push without eviction returns NONE. Eviction hands ownership to your code, which can inspect it or let it clean up.

```
DIM recent AS BoundedDeque[STRING] = NewBoundedDeque(2)
PushBack(BYREF recent, "first")
PushBack(BYREF recent, "second")
IF LET evicted = PushBack(BYREF recent, "third") THEN
    PRINT "Evicted: "; evicted
END IF
PRINT Front(recent)
PRINT Capacity(recent)
```

Priority queues

```
DIM work AS PriorityQueue[STRING] = NewPriorityQueue()
Push(BYREF work, 20, "later")
```

```
Push(BYREF work, 5, "urgent")
PRINT PeekPriority(work)
PRINT PopPriority(BYREF work)
PRINT PopPriority(BYREF work)
```

This is a min-heap: smaller signed INTEGER priorities come first. Equal-priority order is unspecified, so add your own ordering policy if ties matter. Payloads need no comparison operation. Push/pop take logarithmic time; peeking takes constant time. Resource payloads move during sifting rather than being copied.

Sorted keys

```
DIM names AS SortedSet[STRING] = NewSortedSet()
InsertValue(BYREF names, "Grace")
InsertValue(BYREF names, "Ada")
PRINT At(names, 0)
DIM scores AS SortedDictionary[STRING,INTEGER] = NewSortedDictionary()
SetValue(BYREF scores, "Grace", 2)
SetValue(BYREF scores, "Ada", 3)
PRINT JOIN(KeySnapshot(scores), ",")
PRINT LookupRequired(scores, "Ada")
```

Core SET/DICT retain insertion order; these records keep keys sorted. NewOrderedSet and NewOrderedDictionary are contextual factories for the existing core types. Sorted lookup uses binary search; insertion/removal shifts a vector and is linear. Ranges are half-open and reversed ranges are empty. Lookup returns an optional; LookupRequired reports MissingKey. At uses a checked nonnegative index. Sorted sets also provide merging set algebra; sorted dictionaries provide independent key/value snapshots and a borrowed resource Values view.

Keys follow canonical equality/hash/order laws and reject structural NaNs. Sorted containers do not make an inconsistent custom comparison coherent.

Counters and multiple values per key

```
DIM counts AS Counter[STRING] = NewCounter()
Add(BYREF counts, "red", 3)
Add(BYREF counts, "blue")
PRINT Count(counts, "missing")
PRINT TotalCount(counts)
PRINT RemoveValue(BYREF counts, "red", 2)
PRINT Count(counts, "red")
DIM authors AS MultiMap[STRING,STRING] = NewMultiMap()
Add(BYREF authors, "manual", "Ada")
Add(BYREF authors, "manual", "Grace")
PRINT JOIN(GetAll(authors, "manual"), ",")
PRINT Length(authors)
PRINT TotalCount(authors)
```

Counter Length counts distinct keys; TotalCount counts occurrences. Missing counts are zero; reaching zero removes the key. Amounts must be nonnegative, and invalid amounts or overflow raise CollectionError before mutation. Removal reports the amount actually removed, bounded by the stored count.

MultiMap keeps duplicates in per-key insertion order. GetAll returns an independent copyable list, or empty when missing. Get raises MissingKey; Find returns optional presence. Resource buckets can be borrowed with Get/Find, extracted whole with Extract, or have one owner removed with ExtractAt. RemoveKey destroys its values; removing the last value also removes the key. RemoveOne uses equality and is copyable-only.

Bitsets

```
bits = NewBitSet(10)
Include(BYREF bits, 2)
Include(BYREF bits, 7)
Flip(BYREF bits, 2)
PRINT Has(bits, 2)
PRINT Has(bits, 7)
PRINT Length(bits)
FOR EACH index IN Members(bits)
    PRINT index
NEXT
```

A bitset represents integer membership in `[0, size)`, packed into UINT64 words. Capacity is the universe size; Length counts set bits. Indices are nonnegative and checked with IndexError. Members lists them in ascending order. BitUnion, BitIntersection, BitDifference and BitSymmetricDifference require equal universes; Complement flips valid bits without exposing padding. Reset retains the universe.

Bounded caches: recency, expiry and weight

A cache evicts values to stay within a positive entry bound and, optionally, a weight bound. LRU means least recently used. A successful Find refreshes recency; Peek does not. An insertion result reports Inserted, Evicted and Expired counts.

```
DIM cache AS Cache[STRING,STRING] = NewCache(2)
CachePut(BYREF cache, "a", "alpha")
CachePut(BYREF cache, "b", "beta")
IF LET value = CacheFind(BYREF cache, "a") THEN
    PRINT value
END IF
result = CachePut(BYREF cache, "c", "gamma")
PRINT result.Evicted
IF LET value = CachePeek(cache, "b") THEN
    PRINT value
ELSE
    PRINT "b was evicted"
END IF
```

TTL means time to live, measured since the most recent write. Reads do not extend TTL. Policies are LRU, TTL and LRUWithTTL. TTL policies require a positive Duration; plain LRU forbids a TTL and reads no clock. Weights are caller-chosen accounting units, not measured bytes. maxWeight=0 disables that bound. A single write must satisfy the configured weight policy.

Use NewTimedCache and the At operations with samples from a caller-owned clock when expiry must be deterministic:

```
clock = NewTestClock(InstantFromUnixSeconds(0))
DIM cache AS Cache[STRING,STRING] = NewTimedCache(2, CachePolicy.TTL, Seconds(1),
NowMonotonic(clock))
CachePutAt(BYREF cache, "key", "value", NowMonotonic(clock))
AdvanceTestClock(BYREF clock, Seconds(1))
PRINT ResidentCount(cache)
PRINT PruneExpiredAt(BYREF cache, NowMonotonic(clock))
PRINT ResidentCount(cache)
```

Expiry occurs at age greater than or equal to TTL. ResidentCount/ResidentWeight include expired entries not yet pruned; CacheCount prunes first. External samples must be from the accepted clock domain and mutating samples must not move backward. External caches require At calls; ordinary caches reject them. These mistakes are checked errors. Pruning can occur before a later write fails, so a failed edit is not a rollback transaction.

Caches are ordinary exclusive mutable values, with no implicit lock, timer task, loader or request coalescing. Resource Find/Peek results are loans and Extract transfers ownership. All these generic records hide their storage: serialize logical copyable snapshots or supply a concrete formatter, rather than expecting automatic JSON to expose private representation fields. See the collection-family reference for every operation, cost and checked failure.

8. Dense arrays and buffers

A dense array represents a rectangular coordinate space. Use a list for a growing sequence and an array when dimensions and coordinate bounds matter, such as a board, image grid or numeric table.

Declare coordinates

```
DIM board(1 TO 2, 1 TO 3) AS INTEGER
board(1, 2) = 7
PRINT LEN(board)
PRINT LBOUND(board, 1)
PRINT UBOUND(board, 2)
PRINT board(1, 2)
REDIM PRESERVE board(0 TO 3, 1 TO 4)
PRINT board(1, 2)
PRINT board(0, 1)
```

DIM bounds are inclusive. `DIM values(9) AS INTEGER` allocates ten cells, from 0 through 9. By contrast, ARRAY(length) is a zero-based length constructor. LBOUND/UBOUND dimension numbers start at 1; coordinate values follow the declared bounds. LEN is the total cell count, rather than the length of one dimension.

Cells get valid defaults: numeric zero, FALSE, empty strings, NONE optionals, independent empty containers, and recursively valid record defaults. An enum without a zero member or a choice without an unambiguous default needs a valid initializer. Arrays cannot manufacture valid open files as default cells.

Resize deliberately

REDIM constructs a replacement shape. PRESERVE retains the intersection of the old and new coordinate spaces in every dimension. It does not simply preserve the first N physical cells. New coordinates receive valid defaults; removed cells are destroyed. Resource cells in the intersection move once into the new storage.

Bounds, rank and offset arithmetic are checked, with fatal errors for invalid array coordinates or shapes. Rank belongs to the value, while `AS ARRAY OF T` names its element type. A helper must use the actual bounds/rank it accepts. No borrowed element address may survive a resize. Indexed BYREF access protects the array owner for the call, so a later argument cannot resize that storage.

ToArray accepts a BYVAL list and returns a zero-based rank-one array. Copyable inputs remain independent; named resource lists require TAKE. Chapter 10 shows optional resource cells and initialized resource lists.

Capacity and initialized length

For high-throughput binary work, a packed `LIST OF BYTE` is often appropriate. Capacity is reserved storage; length counts initialized elements. Reserving room does not create bytes that can be read. Use Reserve to avoid repeated growth, Resize for initialized content, Truncate to reduce length, and CopyWithin for checked overlapping copies. Exact overloads and return values are in the buffer reference part.

```
DIM bytes AS LIST OF BYTE = [BYTE(65), BYTE(66), BYTE(67)]
PRINT LEN(bytes)
PRINT DecodeUtf8(bytes)
copy = bytes
copy[0] = BYTE(90)
PRINT DecodeUtf8(bytes)
PRINT DecodeUtf8(copy)
```

Binary bytes are not automatically text. DecodeUtf8 validates the sequence and can report StringError. Use ReadBinaryFile for packed BYTE data; the legacy ReadFileBytes returns a list of INTEGER byte values. Choosing packed data avoids the per-element storage cost of wide integer bytes.

9. Ownership and borrowing, from first principles

When a program creates a list, some memory must hold its elements. When it opens a file, the operating system supplies a handle that eventually needs closing. **Ownership** answers who is responsible for keeping that storage or resource alive and releasing it. **Borrowing** lets other code use it temporarily while the owner remains responsible.

BASIL performs this bookkeeping automatically and checks access before running your program. There is no general garbage collector to wait for. Ordinary owned values are released deterministically as their scopes end, including on checked failure and cancellation. You mainly express intent through assignment and parameter modes, rather than through memory addresses.

Ordinary assignment gives independent values

```
original = [10, 20]
copy = original
copy[0] = 99
PRINT original[0]
PRINT copy[0]
```

Think of `copy = original` as giving `copy` its own value. Nested strings, records and collections follow the same rule: mutating the destination cannot silently change the source. The compiler may eliminate work when behavior stays identical, but you never need to guess whether two ordinary assignments share mutable data.

This differs from languages where assigning a list merely copies a reference to one shared object. It also differs from assuming every assignment behaves like a Rust move. Copyable BASIL values keep independent assignment semantics.

Reading does not require a copy

```
values = [10, 20, 30]
PRINT Total(values)
PRINT values[0]
FUNCTION Total(items AS LIST OF INTEGER) AS INTEGER
    result = 0
    FOR EACH item IN items
        result += item
    NEXT
    RETURN result
END FUNCTION
```

An ordinary parameter gives temporary read-only access to the caller's value. The caller remains the owner; Total does not need an independent list just to read three integers. That access is a borrow. Returning an ordinary owned list from a borrowed input would create an independent result unless the function explicitly declares a borrowed return.

Read-only does not mean globally frozen forever. It means that while this access is active, another access must not mutate, relocate, transfer or destroy the storage in a conflicting way. The rule applies through helpers and suspension. Otherwise an apparently innocent read could use an address invalidated by another operation.

BYREF grants exclusive writable access

```
values = [10, 20]
IncreaseFirst(BYREF values)
```

```
PRINT values[0]
SUB IncreaseFirst(BYREF items AS LIST OF INTEGER)
    items[0] += 1
END SUB
```

The caller explicitly grants writable access. For the duration of the call, the function must be able to rely on the absence of conflicting aliases. An alias is another way to reach the same storage, such as a list and one of its elements.

```
number = 1
Update(BYREF number, BYREF number)
SUB Update(BYREF first AS INTEGER, BYREF second AS INTEGER)
    first += 1
    second += 1
END SUB
```

This is deliberately rejected. Do the two operations sequentially, or redesign the function to accept one writable value. The same principle rejects passing an array element while another argument resizes its parent, or mutating a collection during a protected iteration.

BYREF can address writable list/array elements, fields and existing dictionary values. The helper updates actual stored data; it does not edit a detached copy. Missing dictionary locations fail without insertion. The containing root remains protected through later argument evaluation, the call and suspension.

BYVAL gives the callee an owner

```
values = [3, 1, 2]
PRINT FirstSorted(values)
PRINT values[0]
FUNCTION FirstSorted(BYVAL items AS LIST OF INTEGER) AS INTEGER
    Sort(BYREF items)
    RETURN items[0]
END FUNCTION
```

BYVAL creates a writable local owner. A borrowed copyable argument is snapshotted at evaluation, before later argument effects. Fresh owned results can be adopted directly. Use BYVAL when a function genuinely needs its own mutable copy, and an ordinary parameter when it only reads. Choosing the mode deliberately avoids unnecessary copies without introducing shared mutation.

Parameter mode	Caller intent	Callee access
ordinary	Let this function read my value	Borrowed, read-only
BYREF	Let this function update this storage	Exclusive writable borrow
BYVAL	Give this function an independent local value	Writable owner
TAKE formal	Give this function ownership of a resource	Writable resource owner

Moving with TAKE

Some things cannot sensibly be copied: a file, socket, database connection or user-defined resource. `TAKE value` transfers its ownership. The old binding becomes unavailable until reinitialized. BASIL never infers a resource transfer because a variable appears unused afterward.

Anonymous functions can own independent snapshots of copyable outer values. That is a copy into a closure, rather than a stored borrow or implicit resource transfer; chapter 13 demonstrates when the snapshot is taken. Resources and task/channel capabilities cannot be captured. APIs needing retained resource state, such as logging sinks, take an explicit owned context instead.

TAKE also supports whole writable copyable locals, when explicit transfer is useful. For ordinary data it is usually simpler to let independent-value assignment and compiler copy elimination do their jobs. A TAKE formal itself requires a resource-owning type. Chapter 10 develops resource examples.

Borrowed results and lifetimes

A lifetime is the interval during which storage is valid for an access. BASIL normally infers it. A function returning a view declares the owners from which that view can come:

```
TYPE Label
    Text AS STRING
END TYPE
FUNCTION First(items AS LIST OF Label) AS BORROW Label FROM items
    RETURN items[0]
END FUNCTION
labels = [Label { Text="Ready" }]
PRINT First(labels).Text
```

First borrows a record from labels instead of manufacturing another owner. The result cannot outlive its origin, become writable access, or smuggle out a reference to a local that has already been destroyed. Copyable results can be independently copied. Stable named owners are required for borrowed origins; retain a fresh collection in a local before asking it for a returned view.

`FROM a OR b` declares two possible origins and protects both, even when a particular run returns only one. General stored references, mutable view objects and arbitrary reference graphs remain unsupported. BORROW is a precise advanced contract, not a way to bypass normal ownership checks.

10. Files, transfers and automatic cleanup

A resource is a value with a responsibility that cannot be duplicated implicitly. An open file is the simplest example: copying its handle into two independent owners could make both close the same operating-system resource. BASIL therefore rejects implicit resource copies and releases each owner once.

Use and release a file

```
file = OpenFile("/dev/null")
PRINT GetFileInfo(file).Size
```

OpenFile returns a fresh owner. GetFileInfo borrows it. At scope exit the file closes automatically; there is no manual free. `/dev/null` is used to make this example independent of your personal files. In an application, opening a path can fail with IoError, which you can catch or declare.

```
file = OpenFile("/dev/null")
other = file
```

This is intentionally rejected. If another function only needs to read or use the file, pass an ordinary borrowed parameter. If it must keep ownership, transfer the owner explicitly.

Transfer to a function

```
file = OpenFile("/dev/null")
Inspect(TAKE file)
SUB Inspect(TAKE input AS File) FAILS IoError
    PRINT GetFileInfo(input).Size
END SUB
```

After TAKE, the caller cannot use file until it assigns a new owner. Inspect now owns input and releases it when finished. A fresh result can be passed directly: `Inspect(OpenFile(path))`. A return of a named resource similarly needs TAKE.

Transfer is not rolled back if a later argument fails. Already transferred temporaries are cleaned up; the original binding remains unavailable. Reassignment normally constructs the new value before destroying the old one, but explicitly taking the old owner inside the right-hand side changes that availability.

Compose resources

```
TYPE Reader
    Handle AS File
    Name AS STRING
END TYPE
reader = Reader { Handle=OpenFile("/dev/null"), Name="input" }
PRINT reader.Name
PRINT GetFileInfo(reader.Handle).Size
```

Reader automatically becomes noncopyable because it owns a File. Its fields clean up recursively. If construction fails after some fields have completed, those completed owners are released. No incomplete record is exposed to users.

For extra release behavior, a module may define one nongeneric, nonpublic `SUB __Cleanup(BYREF value AS YourType)`. The compiler invokes it with a complete record, then releases owned fields. Application code cannot call or take a function reference to this reserved hook.

```
TYPE Ticket
    Name AS STRING
END TYPE
SUB __Cleanup(BYREF ticket AS Ticket)
    PRINT "Released "; ticket.Name
END SUB
SUB Work()
    ticket = Ticket { Name="job" }
    PRINT "Working"
END SUB
Work()
PRINT "Done"
```

A cleanup hook makes even this otherwise copyable record a resource. Hooks must handle their own operational failures. Explicit checked Finish/Commit/Close operations are still needed when an application must observe successful completion; automatic cleanup is not a promise that a transaction committed. Cleanup may suspend, with cancellation deferred while it completes. Fatal allocation or runtime traps have no normal application-cleanup guarantee.

Optional owners and containers

```
DIM maybe AS OPTIONAL OF File = SOME(OpenFile("/dev/null"))
IF LET borrowed = maybe THEN
    PRINT GetFileInfo(borrowed).Size
END IF
IF LET owned = TAKE maybe THEN
    PRINT GetFileInfo(owned).Size
END IF
```

The first branch loans the payload read-only and protects maybe from conflicting mutation/transfer. The second consumes the optional and owns its present payload. NONE is a valid initialized absence, unlike an unavailable moved binding.

Lists, arrays, optionals and dictionary values can own resources. Those containers become noncopyable; operations requiring copies of resource elements are rejected. Use Pop/ExtractAt to remove list owners and Extract to remove dictionary values. RemoveAt/Remove instead destroy removed owners. Iteration borrows rather than transferring every visited resource.

```
ready = [OpenFile("/dev/null"), OpenFile("/dev/null")]
files = ToArray(TAKE ready)
PRINT LEN(files)
PRINT GetFileInfo(files(0)).Size
DIM optionalFiles(1) AS OPTIONAL OF File
optionalFiles(0) = SOME(OpenFile("/dev/null"))
REDIM PRESERVE optionalFiles(-1 TO 2)
PRINT LEN(optionalFiles)
```

Optional cells default to NONE. A File itself has no default, so use a list of already initialized owners followed by ToArray when every cell must contain a file. Resource fills cannot duplicate one owner across new cells.

Resource dictionary Get/brackets borrow a view, Find supplies an optional view, and SetDefault adopts a fresh or explicitly transferred fallback before borrowing the stored value. `Values(BYREF dictionary)` permits checked compaction and then a read-only iteration view. These operations do not manufacture independent resource copies. Resource keys and resource sets remain unsupported.

Replace or detach fields

`TAKE owner.Field REPLACING replacement` transfers the previous field and installs a complete replacement atomically. Failure before commitment preserves the old field. SWAP exchanges

initialized writable same-type bindings, nested fields or indexed cells without copies or cleanup; swapping an optional field with NONE is useful for detaching it while keeping its parent complete.

```
TYPE Pair
    Left AS File
    Right AS File
    Label AS STRING
END TYPE
pair = Pair { Left=OpenFile("/dev/null"), Right=OpenFile("/dev/null"), Label="inputs" }
left = TAKE pair.Left
PRINT pair.Label
pair.Left = OpenFile("/dev/null")
PRINT GetFileInfo(pair.Left).Size
```

Exchange indexed values without copying owners

SWAP now accepts nested fields and list/array elements or existing dictionary values, including two checked cells in the same container. The equivalent function-style statement is `Swap(BYREF left, BYREF right)`.

```
items = ["first", "second"]
snapshot = items
Swap(BYREF items[0], BYREF items[1])
PRINT items[0]
PRINT snapshot[0]
SwapAt(BYREF items, 0, 1)
PRINT items[0]
```

The values exchange without cloning or destroying their payloads. Both selectors evaluate once, left to right. Later selector failure before the exchange preserves the stored values, although selector side effects already performed remain. Exchanging a location with itself is valid. A parent and its own overlapping subobject cannot be exchanged; read-only fields, string characters, slices, set keys and task/channel capabilities are not writable exchange locations.

Direct indexed SWAP retains each container's normal bounds policy; a missing dictionary key reports MissingKey without insertion. SwapAt is a checked LIST alternative: both indices must be nonnegative and in range, or it raises IndexError before changing storage. It returns the unchanged list length. These dedicated two-location rules do not permit conflicting BYREF arguments to ordinary functions. See storage exchange for exact selector restrictions.

Static field paths in writable local owners can leave checked holes when their containing records have no whole-record cleanup hooks. Other initialized fields remain usable, but whole-record use requires restoration of every hole. A hook could inspect any field, so hooked parents require atomic replacement instead. Indexed element holes and partial holes through borrowed parameters are rejected. Use the field-move reference for detailed flow rules.

11. Checked errors and recovery

An operation can fail for an ordinary reason: a missing file, malformed JSON or a request rejected by an application rule. Checked errors make these outcomes part of a function's contract. They are separate from successful results, optional absence, cancellation and fatal runtime traps.

Declare and raise an error

```
ERROR InvalidAge
    Value AS INTEGER
END ERROR
FUNCTION ValidateAge(value AS INTEGER) AS INTEGER FAILS InvalidAge
    IF value < 0 THEN
        RETURN ERROR InvalidAge { Value=value }
    END IF
    RETURN value
END FUNCTION
TRY
    PRINT ValidateAge(-3)
CATCH problem AS InvalidAge
    PRINT "Invalid age: "; problem.Value
END TRY
PRINT "Continuing"
```

FAILS lists errors that can leave the function. RETURN ERROR selects failure; ordinary RETURN selects success. Plain calls automatically check failure, even when you discard the successful result. Inline TRY on a call is optional legacy syntax, not a requirement to activate checking.

Catch here or propagate

Use TRY/CATCH when this part of the program can recover, add context or report a useful result. Otherwise declare the escaping error in the enclosing function's FAILS clause. Merely ignoring a return value cannot hide a checked error.

```
ERROR MissingItem
    Name AS STRING
END ERROR
FUNCTION Lookup(name AS STRING) AS STRING FAILS MissingItem
    RETURN ERROR MissingItem { Name=name }
END FUNCTION
FUNCTION Title(name AS STRING) AS STRING FAILS MissingItem
    RETURN UPPER(Lookup(name))
END FUNCTION
TRY
    PRINT Title("report")
CATCH problem AS MissingItem
    PRINT "Missing: "; problem.Name
END TRY
```

Failure skips the remaining successful path to the matching handler or caller. Owned locals and partial temporaries clean up. Errors thrown inside a catch go to outer handlers or propagate, rather than re-entering a sibling catch of that same TRY. `RETURN ERROR problem` forwards a caught error. Catch bindings are read-only owners; independently copy a copyable error to edit it before forwarding.

The implicit top-level boundary reports an unhandled error after cleanup and exits 1. An explicit Main still needs to declare its escaping errors. Keep lower-level functions precise; let the application boundary decide reporting policy.

Error families

```
ERROR OperationFailure
    Message AS STRING
END ERROR
ERROR MissingDocument EXTENDS OperationFailure
    Path AS STRING
END ERROR
TRY
    Load()
CATCH problem AS OperationFailure
    PRINT problem.Message
END TRY
SUB Load() FAILS MissingDocument
    RETURN ERROR MissingDocument { Message="Cannot load", Path="notes.txt" }
END SUB
```

Catching a parent handles descendants and exposes shared fields. Copying or forwarding a parent view preserves the concrete error and its complete payload. The common root Error has no universal Message or Code field. Prefer specific types when they let you handle the cause accurately. Broad-catch warnings show declared error sets and call/callback origins; they do not require UNSAFE.

Callback-dependent failures

A wrapper should preserve a callback's actual checked failures, rather than silently replace them with a generic InvocationError or widen every call to Error.

```
ERROR SigningFailure
    Message AS STRING
END ERROR
SUB Invoke(action AS CALLABLE() FAILS (Error)) FAILS LIKE action
    action()
END SUB
SUB Sign() FAILS SigningFailure
    RETURN ERROR SigningFailure { Message="Rejected" }
END SUB
TRY
    Invoke(Sign)
CATCH problem AS SigningFailure
    PRINT problem.Message
END TRY
```

FAILS LIKE action admits the supplied callback's checked set. The same mechanism can follow callbacks stored in configuration records and containers. `AS LIKE argument` returns an owned value retaining its contained contracts; it is not a borrowed lifetime. `AS Wrapper LIKE argument` carries admitted errors into a wrapper shape. Signature references such as `FAILS LIKE Decode(AS STRING)` compose contracts without executing application code. The reference part explains these advanced forms and their limits.

What a catch cannot recover

Cancellation is a control path for stopping a task; ordinary typed CATCH does not intercept it. Fatal allocation failure, stack exhaustion, checked integer traps and direct bounds traps have no normal recoverable-cleanup promise. A missing optional value is NONE, not automatically an error. Model these distinctions in your application instead of catching Error and assuming every possible failure has become recoverable.

12. Modules, projects and packages

Modules keep related declarations together. A project describes which source is the entry point and which local packages it depends on. Neither requires importing code to execute hidden initialization.

Split a program into files

Save this module as `greetings.bas` beside the entry file:

```
PUBLIC FUNCTION Greeting(name AS STRING) AS STRING
    RETURN "Hello, " + name
END FUNCTION
```

Save the entry as `main.bas`:

```
IMPORT "greetings.bas" AS greetings
PRINT greetings.Greeting("Ada")
```

Build `main.bas`. Quoted paths resolve relative to the importing file. Without AS, the filename stem supplies the alias. Identifiers are case-insensitive; Linux file paths are case-sensitive. Imports do not inject names unqualified or re-export another module's imports. Use the alias explicitly.

Only the designated entry source has executable statements. Imported modules contain declarations and imports. Cyclic source imports and cross-file recursion are supported because declarations are discovered before body checking. Unused user/imported bodies are still checked; emission prunes unreachable code.

Design a module boundary

PUBLIC exposes supported declarations; PRIVATE hides them from qualified external access. Unmodified module declarations are public for compatibility. A public signature cannot expose a private type or private error contract. Entry locals are not mutable module globals; PUBLIC DIM does not create global storage.

Save this module as `counter.bas`:

```
OPAQUE TYPE Counter
    PUBLIC Label AS STRING
    Value AS INTEGER
END TYPE
PUBLIC FUNCTION Create(label AS STRING) AS Counter
    RETURN Counter { Label=label, Value=0 }
END FUNCTION
PUBLIC FUNCTION Current(counter AS Counter) AS INTEGER = counter.Value
PUBLIC SUB Increment(BYREF counter AS Counter)
    counter.Value += 1
END SUB
```

This entry uses the public operations and reads the public label:

```
IMPORT "counter.bas" AS counters
counter = counters.Create("Visits")
counters.Increment(BYREF counter)
PRINT counter.Label
PRINT counters.Current(counter)
```

Clients can read Label but cannot assign it, borrow it BYREF or transfer it out. Value and construction remain module-controlled. The module's own code can mutate when ordinary exclusion rules allow it. Read-only public fields do not imply that all aliases throughout the program are globally immutable.

Create a project manifest

A `basic.toml` in the project directory can select an entry and dependencies:

```
manifest-version = 1
[project]
name = "hello"
version = "0.1.0"
[toolchain]
compiler = ">=0.1.0, <0.2.0"
[target]
kind = "executable"
root = "main.bas"
```

```
build/selfhost/stage3 build --manifest basic.toml
build/selfhost/stage3 run --manifest basic.toml -- --help
```

Manifest paths are relative to the declaring manifest. Arguments after `--` go to the application; run preserves its terminal I/O and exit status. `help build` works without loading a project. The manifest reader supports the documented schema-specific TOML subset, not a general-purpose TOML parser.

Depend on local packages

```
[dependencies]
Utilities = {path = "../utilities", version = "0.1.0"}
Graphics = {path = "../graphics/build/graphics.blib", version = "0.1.0"}
```

`IMPORT Utilities` resolves the declared alias. A source dependency names a project directory; a prebuilt dependency names a .blib. Compiler version ranges are checked transitively, and dependency versions can assert an exact package version. Remote fetching, lockfiles and shared incremental analysis caches are not implemented. Imports do not download dependencies.

For a library, set `kind = "library"`. A .blib contains checked interfaces, precompiled nongeneric code, generic source templates, transitive native inputs and notices. The internal BASIL ABI is versioned; rebuild source when a package is incompatible with the consuming toolchain. Renaming a file does not make its ABI compatible.

Declared system native libraries use the explicit hosted linker bridge and matching SDK inputs. Ordinary static programs use the integrated emitter/linker. See the project reference for archive declarations, dependency identity, output paths and current debugging boundaries.

INTERNAL visibility is deferred and is not supported syntax. Use the implemented PUBLIC/PRIVATE module boundaries rather than relying on an intermediate package-visible access level.

13. Generics, callbacks and custom value behavior

Generics describe one operation over multiple statically known types. Callbacks let a caller supply behavior. Neither requires dynamically typed values or runtime method inheritance.

Generic functions

```
FUNCTION Identity[T](value AS T) AS T = value
PRINT Identity(7)
PRINT Identity("BASIC")
```

T is a type parameter inferred from the supplied argument. The compiler specializes and checks the relevant concrete operation. An unsupported operation for one type is an error, rather than a runtime guess. Explicit bracketed call type arguments are not supported. Generic records use `TYPE Box[T]` and concrete types such as `Box[STRING]`, including named `Box[STRING] {Value="text"}` and positional `Box[STRING](../../docs/book/chapters/%22text%22)` constructors. Opaque records keep the same privacy rules. Generic record contracts and current limits.

Some calls need more context than their arguments provide. An explicit destination type or a checked `AS T` expression can complete supported inference. AS is type ascription, not numeric conversion. It does not change one representation into another. Missing generic information cannot always be inferred from a later use.

Advanced generic overloads may use `WHERE T IS COPYABLE` or `WHERE T IS RESOURCE`. These constraints select the supported ownership category before specialization; they are not a general user-defined trait system. Use them when an implementation genuinely needs independent copies or resource ownership.

Build reusable data with generic records

A generic record lets you reuse a data shape while retaining exact static field types. Define its parameters on TYPE, then bind them at construction or through checked destination context:

```
TYPE Box[T]
    Value AS T
END TYPE
text = Box[STRING] { Value="original" }
copy = text
copy.Value = "changed"
PRINT text.Value
PRINT copy.Value
number = MakeBox(42)
PRINT number.Value
FUNCTION MakeBox[T](value AS T) AS Box[T]
    RETURN Box[T] { Value=value }
END FUNCTION
```

Box[INTEGER] and Box[STRING] are distinct nominal types. Type parameters can appear inside lists, optionals and other records. They remain part of identity even when no field uses them. A destination such as `DIM box AS Box[INTEGER]` can supply the type context for `Box(42)`; later mutation does not retroactively infer an earlier declaration. Ordinary typed constructors also have positional forms, such as `Box[STRING](../../docs/book/chapters/%22text%22)`.

Copies are independent whenever the actual fields are copyable. A Box[File] that stores a File is a resource owner; a type parameter mentioned only in the type's identity does not itself create an owned resource. Read-only functions borrow; owning construction/insertion still requires fresh or explicit TAKE values. Generic records do not erase the difference between copies and transfers.

Finite recursive data can use containers:

```
TYPE Node[T]
    Value AS T
    Children AS LIST OF Node[T] = []
END TYPE
root = Node[INTEGER] { Value=1 }
Append(BYREF root.Children, Node[INTEGER] { Value=2 })
copy = root
copy.Children[0].Value = 9
PRINT root.Children[0].Value
PRINT copy.Children[0].Value
```

Direct inline containment cycles and indefinitely expanding type instantiations are rejected. Generic records may be opaque, keeping the same construction and field privacy rules. One `WHERE T IS COPYABLE` or `WHERE T IS RESOURCE` constraint may follow the record header. A function pattern `Box[T]` is more specific than an otherwise applicable unconstrained `T` pattern; incomparable overloads still need disambiguation.

Generic record extension, generic errors/choices, generic cleanup hooks and generic OPERATOR declarations are not implemented. Existing nongeneric extension can contain generic-record fields. JSON/FORMAT adapters work for eligible concrete records; opaque storage still needs a deliberate formatter or logical snapshot.

Pass a named function

```
FUNCTION Apply(value AS INTEGER, action AS CALLABLE(INTEGER) AS INTEGER) AS INTEGER
    RETURN action(value)
END FUNCTION
FUNCTION DoubleIt(value AS INTEGER) AS INTEGER = value * 2
PRINT Apply(21, DoubleIt)
```

CALLABLE lists parameter types and modes, then an optional successful result type. Without a result it describes a SUB. A failure set uses `FAILS (ErrorType, ...)`. Indirect calls enforce those contracts. Default parameter names/values on the original declaration do not become an unchecked indirect-call feature.

Named functions do not capture locals. Pass explicit data, store independent configuration in supported records, or use an anonymous function when a snapshot of outer copyable data is useful. A named local helper remains noncapturing even though its declaration is nested.

Lambdas: short functions written where needed

A lambda is an anonymous function value. The expression form is `(parameters) => expression`. Parameter types can be explicit or supplied by an expected concrete CALLABLE, as with a collection predicate.

```
minimum = 10
selected = Filter([1, 10, 20], (n) => n >= minimum)
PRINT LEN(selected)
square = (x AS DOUBLE) => x * x
PRINT square(3.0)
```

Filter supplies INTEGER parameter context for n. A stand-alone square needs its parameter type. The successful result is inferred from the expression. `()` is the parameter list for a zero-argument lambda.

Closures own snapshots, rather than shared variables

A closure is a function value together with data it captured from its creation scope. A free local is a name used by the function that is not one of its own parameters or locals. Copyable free locals become independent read-only snapshots when the closure is constructed:

```
values = [10, 20]
first = () => values[0]
copy = first
values[0] = 99
PRINT first()
PRINT copy()
PRINT values[0]
```

Both function values still return 10. The snapshot includes owned strings, collections and other copyable callables. A closure can be returned after its creator exits because it owns its captures; it does not retain dangling borrowed locals. Copying a closure creates an independent value with ordinary cleanup.

For several statements, use an anonymous FUNCTION or SUB block:

```
offset = 2.0
transform = FUNCTION(x AS DOUBLE) AS DOUBLE
    adjusted = x + offset
    IF adjusted < 0.0 THEN RETURN 0.0
    RETURN adjusted * adjusted
END FUNCTION
notify = SUB(message AS STRING)
    PRINT message
END SUB
PRINT transform(3.0)
notify("Done")
```

Block lambdas infer checked escaping failures unless explicitly declared with FAILS. Successful returns must be consistent and complete. Collection predicates still require an infallible callback: a lambda does not weaken the receiving API's contract. Other checked callbacks propagate their real failures as in chapter 11. Fields, list elements and returned CALLABLE values can be invoked directly.

Captured bindings are read-only. They do not provide mutable shared cells, and resources or task/channel capabilities cannot be captured. Pass such owners explicitly to an operation or an owned-context API. For example, this is rejected:

```
file = OpenFile("/dev/null")
inspect = () => GetFileInfo(file).Size
```

TAKE is not an implicit capture modifier. Noncapturing foreign CALLBACK and NOSUSPEND delegates retain their separate restrictions; anonymous adaptation to those kinds and nested named declarations inside anonymous bodies are unsupported. See callables and captures.

Supply a sort comparator

```
values = [1, 3, 2]
Sort(BYREF values, Descending)
FOR EACH value IN values
    PRINT value
NEXT
FUNCTION Descending(a AS INTEGER, b AS INTEGER) AS INTEGER
    IF a > b THEN RETURN -1
    IF a < b THEN RETURN 1
```

```
    RETURN 0
END FUNCTION
```

An INTEGER comparator returns negative, zero or positive according to the desired ordering. CompareResult overloads provide typed Less/Equal/Greater/Unordered outcomes. Comparators are checked and must satisfy their supported contract. The collection reference covers callback signatures and unordered handling.

An ordinary capturing lambda can supply comparison policy too:

```
direction = -1
comparator = (a AS INTEGER, b AS INTEGER) => direction * (a - b)
FOR EACH value IN Sorted([1, 3, 2], Compare=comparator)
    PRINT value
NEXT
```

Here subtraction is safe for the small demonstrated values. For arbitrary INTEGER inputs, branch on comparisons as Descending does instead of risking overflow in `a - b`. The lambda snapshots direction at construction and remains subject to the comparator's infallible, read-only access contract.

Define a type's equality, ordering and hash

Default ordinary-record equality compares all fields. Sometimes a value's application identity uses fewer fields. Canonical OPERATOR declarations make that policy consistent across equality, nested values and hash collections.

```
TYPE Person
    Name AS STRING
    Age AS INTEGER
END TYPE
OPERATOR Equal(a AS Person, b AS Person) AS BOOLEAN = a.Name = b.Name
OPERATOR Hash(value AS Person) AS INTEGER = Hash(value.Name)
OPERATOR Compare(a AS Person, b AS Person) AS CompareResult = Compare(a.Name, b.Name)
a = Person("Ada", 30)
b = Person("Ada", 40)
PRINT a = b
DIM people AS SET OF Person = SET()
Insert(BYREF people, a)
Insert(BYREF people, b)
PRINT LEN(people)
```

These declarations belong in the type's defining module. Merely naming a normal function Equal does not register a capability. Custom Equal disables incompatible generated Hash/Compare; supply coherent operations when needed. Equal keys must hash equally, and ordering must agree with equality. The compiler checks exact signatures and effects but cannot prove your mathematical laws.

Equality, ordering and hash operators borrow operands and cannot transfer them. They cannot perform I/O, inspect clocks/randomness, start tasks, explicitly suspend or invoke unverified foreign/indirect behavior. Resource operators are rejected. A derived record does not virtually inherit a base's canonical operator. The operator reference gives exact effects, unordered behavior and nested dispatch rules.

Define a readable text representation

OPERATOR Display gives a nominal record, choice, error or enum a deliberate text representation. PRINT, interpolation, explicit Display and logging use the same capability. An ordinary FUNCTION Display does not register it.

```
TYPE Person
    Name AS STRING
    Age AS INTEGER
END TYPE
OPERATOR Display(value AS Person) AS STRING
    RETURN f"{value.Name}, age {value.Age}"
END OPERATOR
person = Person("Ada", 36)
PRINT person
PRINT f"Person: {person}"
PRINT Display(person)
PRINT f"[{person:.3}]"
```

The final line prints `[Ada]`: formatting first obtains text, then applies STRING precision/alignment/fill rules. Numeric format specifications are rejected for a displayed nominal value. There is no automatic field dump, implicit concatenation or general conversion to STRING for an unregistered type.

Declare one module-level hook in the exact type's defining module, with one required read-only operand and an owned STRING result. Generic/default/owning parameters, BORROW results and escaping FAILS are not allowed. The hook and its helpers follow the same pure-computation restrictions as other canonical operators. Ordinary extension does not inherit it.

Display also supports resources without copying or transferring them:

```
TYPE Ticket
    Name AS STRING
END TYPE
SUB __Cleanup(BYREF ticket AS Ticket)
    PRINT "Released "; ticket.Name
END SUB
OPERATOR Display(ticket AS Ticket) AS STRING = "Ticket: " + ticket.Name
ticket = Ticket { Name="job" }
PRINT ticket
PRINT ticket.Name
```

The owner remains usable and closes once at scope exit. Display adds no resource equality, hashing or copying. Direct nominal logging arguments are rendered only after severity filtering; repeated message fields reuse that text. Explicit Display/LogArgument calls are ordinary eager expressions. A deferred function retains its actual errors and normal result cleanup. See the public value-operator reference.

14. Coroutines and concurrent tasks

A coroutine is an execution that can suspend and later resume with its local state intact. BASIL's task runtime uses stackful coroutines: an ordinary helper can suspend deep inside a call chain without adding ASYNC/AWAIT syntax to every function. A task adds an owned running operation with completion, errors and cancellation that a parent can observe.

Concurrency means several operations can make progress over overlapping time. Parallelism means executing work simultaneously on multiple CPU cores. BASIL supports both within its placement rules; starting a task does not promise that every library operation will execute in parallel.

Start and join

```
worker = SPAWN Square(12)
IF LET answer = Join(worker) THEN
    PRINT answer
ELSE
    PRINT "Cancelled"
END IF
FUNCTION Square(value AS INTEGER) AS INTEGER = value * value
```

SPAWN returns a noncopyable typed handle. Join waits through task cleanup. A FUNCTION task returns SOME(result) on success and NONE on cancellation. A SUB task returns TRUE/FALSE instead. Operational errors propagate through checked calls rather than becoming cancellation or an empty optional.

Copyable arguments are independent snapshots taken in written argument order. Named arguments and constant defaults work as in ordinary calls. Generic entries infer types from supplied values. No implicit capture of parent locals occurs. Borrowed/BYREF task arguments remain unsupported; channel endpoints have a specific checked scope relationship instead.

Understand scheduling

```
first = SPAWN Compute(10)
second = SPAWN Compute(20)
IF LET value = Join(first) THEN
    PRINT value
END IF
IF LET value = Join(second) THEN
    PRINT value
END IF
FUNCTION Compute(value AS INTEGER) AS INTEGER
    Yield()
    RETURN value * 2
END FUNCTION
```

The printed order is determined here by the parent's joins, not by which task finishes first. Ordinary user function entries and loop backedges contain budgeted scheduling safe points, so CPU loops do not normally need manual Yield. `Sleep(milliseconds AS INTEGER)` parks for a positive interval and yields for nonpositive values. Time units are explicit; Sleep takes integer milliseconds, whereas SELECT AFTER takes a typed Duration.

Verified independent CPU tasks can run on a bounded worker pool. Channel, I/O, resource, foreign and other unverified work stays on the coordinator under the current rules. Native file/DNS/database worker pools are separate. There is no OS thread per task. Blocking native calls and long library transactions can delay other work and cancellation; no hard latency or unrestricted multicore guarantee is implied.

Request cancellation and observe it

```
worker = SPAWN WaitForever()
Cancel(worker)
IF NOT Join(worker) THEN PRINT "Stopped"
SUB WaitForever()
    WHILE TRUE
        Sleep(1000)
    WEND
END SUB
```

Cancel requests stopping; it does not wait. Join confirms the task and dependent cleanup have finished. Cancellation bypasses typed CATCH, and cleanup hooks can defer its acknowledgement while releasing resources. An existing terminal error is retained: Cancel cannot overwrite it. Observe a task's outcome even after a timeout or a COMPLETED notification.

Normal statement-SPAWN children are joined at their invocation's scope exit; failure/cancellation cancels and joins them before dependent locals are destroyed. Retained handle cleanup supplies cancellation/join fallback and error observation. Do not rely on cleanup to finish a protocol that can wait forever: explicitly close producers or cancel outstanding work.

WaitTasks/CancelTasks act on the current invocation's compatibility statement-child scope, not every task in the process. Prefer explicit handles when retaining results or operational errors. Statement SPAWN requires infallible entries; fallible entries need a handle and checked supervision.

Tasks can own resources

```
file = OpenFile("/dev/null")
worker = SPAWN FileSize(TAKE file)
IF LET size = Join(worker) THEN
    PRINT size
END IF
FUNCTION FileSize(input AS File) AS INTEGER FAILS IoError
    RETURN GetFileInfo(input).Size
END FUNCTION
```

The task owns the argument storage, while its ordinary input parameter borrows that storage. A named resource needs TAKE; a fresh resource result can be adopted. Cancellation before entry still releases task-owned arguments.

Tasks can return resources too. A successful Join transfers that result once; another successful extraction reports TaskResultTaken. Copyable results may be joined repeatedly as independent values. A successful resource optional NONE returns SOME(NONE), distinct from cancellation. Unclaimed resource results are released automatically. Repeated failed joins retain the original concrete error.

15. Channels, SELECT and timeouts

A channel coordinates tasks by exchanging typed messages. Use messages instead of trying to share arbitrary writable variables. The channel owns its queue and wait registrations; sender and receiver capabilities authorize specific operations without exposing that internal storage.

Send a sequence and close it

```
DIM jobs AS CHANNEL OF INTEGER = Channel(2)
producer = SPAWN Produce(Sender(jobs))
DO
    IF LET value = Receive(Receiver(jobs)) THEN
        PRINT value
    ELSE
        EXIT DO
    END IF
LOOP
Join(producer)
SUB Produce(output AS SENDER OF INTEGER) FAILS ChannelClosed
    Send(output, 10)
    Send(output, 20)
    Close(output)
END SUB
```

Capacity 2 permits two buffered messages. Capacity zero is a rendezvous: Send waits until a receiver accepts the message. A full buffer parks senders; an empty open channel parks receivers. These operations wait through the scheduler rather than spinning or requiring an OS thread for each wait.

Messages have independent-value semantics. Changing a source list after sending cannot alter the delivered message. Endpoints cannot escape their channel owner's scope or become arbitrary long-lived stored capabilities. The supported message subset uses copyable data; resource-message transfer is not a general channel API.

Close ends sending. It does not discard accepted buffered messages. Receivers get SOME(message) until drained, then NONE. A nested optional message NONE is delivered as SOME(NONE). Sending after closure reports ChannelClosed. Closing one borrowed capability is a deliberate protocol operation, not a side effect of merely ending a local parameter's scope.

Wait for one of several events

```
DIM notices AS CHANNEL OF STRING = Channel(1)
Send(Sender(notices), "Ready")
Close(Sender(notices))
SELECT
CASE SOME(message) = RECEIVE(Receiver(notices))
    PRINT message
CASE NONE = RECEIVE(Receiver(notices))
    PRINT "Closed"
CASE AFTER Seconds(1)
    PRINT "Timed out"
END SELECT
```

Scheduler SELECT commits exactly one case. Each receiver requires paired SOME and NONE cases with the same expression spelling, so the operation covers both message and closure. Current operands are named receiver parameters or Receiver(namedOwner). Case message bindings are read-only and local to that case. SELECT SEND and scheduler CASE ELSE are not implemented.

The runtime unregisters losing waits without consuming their messages. Priority rotates across repeated entries to a SELECT site when multiple cases are ready. A permanently closed receiver

remains ready; stop selecting it when it is no longer useful. Fairness cannot make progress if your program stops entering the selection or performs work that never reaches a scheduling boundary.

Observe completion with a deadline

```
worker = SPAWN Work()
SELECT
CASE COMPLETED(worker)
    PRINT "Ready to join"
CASE AFTER Seconds(1)
    Cancel(worker)
END SELECT
IF Join(worker) THEN PRINT "Joined"
SUB Work()
    PRINT "Working"
END SUB
```

COMPLETED means the task is ready to observe, not that it succeeded. Always Join to handle retained success, failure or cancellation. A timeout winning the SELECT does not erase a child error.

AFTER takes a typed Duration such as Milliseconds(100). Each entry starts a fresh relative timeout; re-entering it is not a fixed periodic cadence. Nonpositive durations are immediately ready. Unrepresentable delays or absolute deadlines raise DateTimeError before registering any waits, so functions using AFTER must catch or declare it. Operand evaluation counts against the new timeout.

Cancellation before commitment consumes no message and removes waits. After a delivery commits, cancellation may prevent the body from observing it; the committed value is still destroyed exactly once rather than put back into the channel. Design protocols around commitment and acknowledgement, rather than assuming cancellation rewinds already accepted work.

16. Console input, files and subprocesses

A utility often reads input, processes text and writes a result. BASIL supplies simple console statements and owned platform resources for larger operations. The calls use the same checked errors and ordinary suspension as the rest of the language.

Read console input

This interactive program asks for a complete line:

Example requirements: interactive program; provide a line on stdin.

```
LINE INPUT "Your name: "; name
PRINT "Hello, "; name
```

LINE INPUT reads a line without interpreting comma-separated fields. INPUT performs checked field conversion into typed destinations. Conversion and field count errors leave the destination set uncommitted, rather than partially updating it. ReadLine returns an optional line and uses NONE for end of input, which makes stream-processing loops convenient.

The following checked example is fed two lines by the book verifier:

```
DO
    IF LET line = ReadLine() THEN
        PRINT UPPER(line)
    ELSE
        EXIT DO
    END IF
LOOP
```

PRINT separates adjacent semicolon items without adding spaces. Commas advance to print zones; they are not CSV output. A trailing separator suppresses the normal newline. Use explicit formatting when generating machine-readable files.

Read and write files

ReadFileText reads a whole UTF-8 file and validates its encoding. ReadBinaryFile returns packed bytes; DecodeUtf8 performs an explicit checked conversion. Large or incremental data can use OpenFile, ReadFileInto and WriteFile instead.

Example requirements: filesystem example; requires an existing notes.txt in the working directory.

```
TRY
    text = ReadFileText("notes.txt")
    PRINT text
CATCH problem AS IoError
    PRINT "Could not read notes.txt: "; problem.Code
CATCH problem AS StringError
    PRINT "The file is not valid UTF-8"
END TRY
```

Relative paths resolve from the running process's working directory, not from the source file. Paths are UTF-8 and cannot contain an interior NUL. Files close on recoverable exits. Calling Close early is useful when a long-lived scope no longer needs a handle; do not construct independent owners around native descriptors.

GetFileInfo provides metadata. ReadDirectory lists entries. MakeDirectory and SetFilePermissions perform checked changes. ReadFileSnapshot and WriteFileAtomic support version-aware editing;

FileChanged lets an editor avoid silently overwriting another writer's work. Treat version conflicts as an application decision to reload, compare or explicitly overwrite.

File open/read/stat/directory/seek operations use bounded workers and park the caller. Writes, flushes and some mutations remain synchronous. A coroutine-friendly API does not mean every underlying filesystem operation is nonblocking.

Run another program

RunProcess captures stdout and stderr while pumping input and output together. It accepts an executable and exact argument strings; it does not insert a shell. An argument containing `*`, spaces or shell punctuation stays an argument. To request shell interpretation you would have to invoke a shell explicitly.

For long-running processes, use StartProcess, ReadProcessInto, WriteProcess, CloseProcessInput, ProcessExited and WaitProcess. TryReadProcessInto distinguishes NONE (not ready) from SOME(0) (EOF). Captured output currently grows without a built-in application output limit; use streaming and an explicit policy when handling potentially large output.

Example requirements: process creation requires /usr/bin/printf on the host.

```
child = StartProcess("/usr/bin/printf", ["Hello from a child"])
PRINT WaitProcess(child)
```

This only demonstrates starting and waiting; use the capture API to consume its output. A Process owns its child leader/group and pipes. Cleanup terminates and acknowledges completion before releasing retained storage; it does not reap other libraries' children. External detached descendants are outside this initial containment model.

Terminal applications

OpenTerminal provides an owned terminal session. EnterTerminalScreen, cursor/color operations and ReadTerminalEvent support full-screen applications. An event timeout returns NONE; EOF is a present event of its defined kind. Unicode scalar positions and terminal display-cell widths differ, so use TerminalTextWidth/TerminalClipText when laying out text. Cleanup restores terminal mode on ordinary recoverable exits. The BASIL IDE is a complete example of these facilities.

Buffer an owned stream

BufferedReader and BufferedWriter own a File or TCPConnection, including TLS. Use TAKE for a named endpoint; a fresh result such as OpenFile or Stdout can be adopted directly. The buffers are resources, so assignment does not implicitly copy them. Their internal byte accumulators do not change that ownership rule.

```
writer = BufferedWriter(OpenFile("greeting.txt", "w"))
WriteLine(BYREF writer, "Hello, λ")
file = Finish(TAKE writer)
Close(file)
reader = BufferedReader(OpenFile("greeting.txt"), Capacity=16)
IF LET line = ReadLine(BYREF reader) THEN
    PRINT line
END IF
IF LET extra = ReadLine(BYREF reader) THEN
    PRINT "Unexpected extra line"
ELSE
    PRINT "EOF"
END IF
```

ReadLine strips LF and a preceding CR. An empty line is SOME(“”); empty EOF is NONE. The final line need not have a newline. MaxBytes defaults to 1 MiB and counts bytes before LF, including CR. A line-limit or UTF-8 failure may already have consumed input. Buffered ReadBytes returns up to

the requested maximum, stopping at EOF; ReadInto may return just one buffered segment. EndOfStream becomes true only after EOF has been observed and lookahead is exhausted.

Always Flush a writer or Finish it before relying on its output. Cleanup discards pending bytes and closes the endpoint; it cannot report a fallible flush. Finish consumes the writer, flushes and returns ownership of the endpoint. Flush drains the buffer; it is not a disk fsync. A failed write has already queued its complete argument: retry Flush to send the unsent suffix, rather than submitting the same argument twice. BufferSize is a flush threshold, not a maximum memory allocation.

Use standard streams as File values

```
writer = BufferedWriter(Stdout())
WriteLine(BYREF writer, "Buffered hello")
Flush(BYREF writer)
reader = BufferedReader(Stdin())
IF LET line = ReadLine(BYREF reader) THEN
    PRINT line
END IF
```

Stdin, Stdout and Stderr return fresh owned duplicates of inherited streams, including redirected files and pipes. Closing a duplicate leaves the original available to PRINT and the other console helpers. The duplicate owners still share underlying position, EOF and file status: use one buffered reader per stream and coordinate writers. Read-only borrowing does not freeze external OS state. File writes remain synchronous; a blocked pipe read may need peer data or EOF before cancellation completes. Endpoint(buffer) borrows its endpoint; bypassing a reader skips lookahead, and bypassing a writer can reorder output. Unbuffer(TAKE reader) returns the endpoint while discarding unread lookahead.

17. JSON, schemas and serialization

Serialization converts application values into a transport representation and back. Prefer typed conversion when you know the data shape; use a dynamic JSON tree when the structure is determined at runtime. In either case, invalid input is a checked error rather than a partly initialized record.

Decode a typed record

```
TYPE Customer
    Id AS INTEGER
    Name AS STRING = "Unknown" JSON NAME "full_name"
    Nick AS OPTIONAL OF STRING = NONE JSON OMITNONE
END TYPE
customer = DecodeJSON("{""Id"":42}") AS Customer
PRINT customer.Name
PRINT EncodeJSON(customer)
```

The AS annotation supplies the requested result type. A known typed destination can supply the same context. Missing fields use explicit defaults, then optional NONE, otherwise fail. Invalid supplied values never silently select a default. OMITNONE leaves an absent optional out of encoded objects.

INTEGER and unsigned values retain exact integer representation. DOUBLE must be finite for standard JSON. Lists/sets become arrays, records become objects, enums use their integers and choices use exactly one named alternative. Nonzero-based or multidimensional arrays include checked bounds metadata. Resource handles have no serializable standard representation.

Work with dynamic values

```
request = JsonObject([
    JsonField("name", "Ada"),
    JsonField("active", TRUE)
])
JsonPut(BYREF request, "visits", 3)
PRINT StringifyJSON(request)
```

JsonValue is a closed data representation, not a universal dynamically typed BASIL variable. ParseJSON parses text into that tree; StringifyJSON writes it. Typed ToJSON/FromJSON bridge between the tree and known BASIL types. JsonObject and JsonArray construct independent values. JsonPut replaces/inserts one object member and preserves previous independent copies. Wrong-kind targets fail with JsonError.

Choose wire directions and names

SERIALIZE supplies shared mapping rules; JSON clauses override the corresponding shared property for JSON. NAME changes wire spelling. READONLY means encode-only, WRITEONLY means decode-only, and EXCLUDE removes both directions; these are wire rules, not permission to mutate BASIL fields. Decoding must still produce a complete valid value. Unknown/disallowed fields fail unless the supplied options permit ignoring them. Duplicate keys/mapped names remain errors.

```
TYPE Customer
    Name AS STRING SERIALIZE NAME "full_name" JSON NAME "name"
END TYPE
customer = Customer { Name="Ada" }
PRINT StringifyJSON(ToValue(customer))
PRINT EncodeJSON(customer)
```

ToValue/FromValue follow shared rules; EncodeJSON/DecodeJSON follow effective JSON rules. This lets one application schema serve multiple codecs without making JSON the meaning of every format's names.

Customize conversion

JsonCodec profiles hold independent per-type converters. Field and TYPE defaults can declare ENCODE WITH and DECODE WITH. Converter representation types must agree. Selection is per direction: JSON field, shared field, supplied profile, JSON type, shared type, standard handling. Representation roots bypass their selected rule once to avoid immediate self-recursion; child values use normal selection.

A converter may accept a second read-only JsonLocation or SerializationLocation to retain path/offset context in a custom error. Its original declared failures propagate through adapters and compiled libraries. Registration currently uses named nongeneric noncapturing functions. Broad mutable global formatter registries and arbitrary resource serialization are not provided.

Attach format-specific metadata

```
TYPE FieldOptions
    Number AS INTEGER
END TYPE
TYPE Message
    Id AS INTEGER FORMAT FieldOptions { Number=1 }
END TYPE
DIM witness AS OPTIONAL OF Message = NONE
options = FormatMetadata(witness, "Id") AS OPTIONAL OF FieldOptions
IF LET field = options THEN
    PRINT field.Number
END IF
```

FORMAT attaches a typed constant record to a type or field. A codec interprets its own options; merely naming an options type does not install a parser. SerializationSchema describes shared mappings; FormatMetadata returns independent typed optional metadata without constructing an uninitialized instance. Arbitrary build-time functions do not execute in these annotations.

Inspect declared types without constructing sample values

TypeSchema, SerializationTypeSchema and JsonTypeSchema return a finite graph of declared types. Use an `OPTIONAL OF T = NONE` witness even when T has required fields or owns resources. The query examines declarations, not a sample object's current contents, and acquires no application resource.

```
TYPE Node
    Name AS STRING
    Children AS LIST OF Node = []
END TYPE
DIM witness AS OPTIONAL OF Node = NONE
graph = JsonTypeSchema(witness)
root = graph.Types[graph.Root]
PRINT root.Name
FOR EACH field IN root.Fields
    PRINT field.Name; ":"; graph.Types[field.DeclaredType].Name
NEXT
childrenType = graph.Types[root.Fields[1].DeclaredType]
PRINT childrenType.ElementType = graph.Root
```

The final comparison is TRUE: the list element refers back to the existing Node descriptor, rather than expanding an infinite tree. Node IDs index this graph only; −1 means no reference. Repeated types share nodes. Generic arguments, including parameters absent from fields, have their own

TypeArguments references. Enum members, choice alternatives and collection element/key/value references describe their respective shapes.

Choose the view deliberately:

Query	Information
TypeSchema	Physical BASIL fields and explicit-default presence
SerializationTypeSchema	Shared names, access rules and declared converter shapes
JsonTypeSchema	Effective declaration-default JSON mappings

Each TypeDescriptor has a display Name, an opaque Identity, Kind, ownership/privacy flags and shape-specific facts. Compare identities as whole values within their documented scope; do not parse them or persist them as versioned schema IDs. TypeFieldInfo distinguishes physical Name, WireName, DeclaredType and each converter direction. HasDefault means an explicit field default, not every fallback that a decoder could supply.

Graph fields are read-only public data, and copied descriptor lists or FORMAT options are independent values. `FormatMetadata(graph, node[, field])` retrieves typed options from the selected declaration. Opaque types expose no fields or field annotations through this query, including otherwise readable PUBLIC fields. Resources, functions, pointers and dynamic JSON explicitly lack a standard static wire shape; a query does not authorize serialization of them.

WireSupported describes a node's standard handling before its root converter; it is not proof that every reachable child is serializable. A wire traversal must select the direction's converter, bypass that representation's root converter once, then visit its children normally. Track direction and bypass state as well as node ID to handle normalization and recursion correctly. Per-instance JsonCodec profiles are not inputs to these declaration-default queries. JSON Schema generation, validation and data synthesis remain separate library work. See the type-metadata reference for full descriptor fields, wire labels, converters and privacy rules.

Embed checked data while building

EMBED reads an input while compiling and creates a read-only module value. The resulting executable does not need the input file or decoder at runtime. Use it for default configuration, templates or other data shipped with the application.

Save this file as `defaults.json` beside the source:

```
{"Host":"localhost"}
```

```
TYPE ServerConfig
    Host AS STRING
    Port AS INTEGER = 8080
END TYPE
EMBED Defaults FROM "defaults.json" AS ServerConfig USING Json
local = Defaults
local.Port = 9000
PRINT Defaults.Host; ":"; Defaults.Port
PRINT local.Port
```

This prints `localhost:8080` then `9000`. Defaults and its nested contents cannot be assigned, transferred or passed BYREF. A normal assignment or BYVAL use produces an independent owned value; read-only calls borrow. PUBLIC/PRIVATE work as for other module declarations, and imported inputs resolve relative to their own source.

Without USING, the supported raw form is STRING. It preserves exact UTF-8 input bytes, including BOM, embedded NUL and line endings:

```
Hello from embedded text!
```

```
EMBED Message FROM "message.txt" AS STRING
PRINT Message;
```

The fixture includes a final newline, which the trailing semicolon preserves without adding another. Raw text embeddings can also serve ordinary constant/default contexts. Values produced by a decoder do not become general constant-expression factories.

USING can name an ordinary visible decoder; no registration or plugin is required:

```
8081
```

```
EMBED Port FROM "port.txt" AS INTEGER USING ReadPort
PRINT Port
FUNCTION ReadPort(text AS STRING) AS INTEGER FAILS NumericError
    RETURN VAL(TRIM(text))
END FUNCTION
```

The first decoder parameter receives STRING content; trailing parameters use their checked defaults and AS supplies result context. The same decoder works at runtime. Json is shorthand for DecodeJson when no visible function named Json overrides it. Decoders can depend on other embedded values; cycles are rejected. Main and application top-level statements do not execute during decoding.

Decoded data can recursively contain records, collections, arrays, optionals, choices and enums. Resources, pointers, function values, task/channel handles and errors as successful data are rejected. Decoders may return their declared errors. The checked decoder and its helpers must avoid I/O, clocks, randomness, task starts, unsafe operations and unknown foreign behavior. The build also isolates execution and enforces bounded input, memory, time and output. A failing decoder is a build diagnostic with the source/input location and original error, not a recoverable failure in the resulting application.

Paths must stay inside the declaring project tree, or the standalone source's directory. Absolute paths, escaping symlinks and unstable/nonregular inputs are rejected. All declarations are checked, including unused ones; unreachable embedded data and decoding machinery are omitted from the executable. Compatible libraries retain public/private embedded values when moved without producer sources/data.

Successful build data may be cached under `build/.basic/embed` beneath the entry source directory. Changed content or decoder inputs invalidate the relevant work; semantic checks remain active. Failures are not cached. Delete that directory to clear the cache; a read-only project can build without it. See the embedding reference for exact limits and failure policies.

The book verifier removes these three fixture input files before executing their compiled programs, checking that the programs no longer depend on those files.

ReadJSON/WriteJSON, JSON Lines and typed HTTP helpers extend the same checked mapping model to files and network bodies. The JSON reference includes exact names, limits and error contracts. YAML, general TOML and XML codecs remain unimplemented; EMBED does not install those parsers. The project manifest's TOML subset is a separate facility.

18. Dates, clocks, UUIDs and randomness

Several values may look like numbers or text while representing very different concepts. Typed dates, durations and identifiers prevent accidentally treating calendar months as fixed seconds or a pseudorandom simulation value as security entropy.

Calendar values and elapsed time

```
day = NewDate(2024, Month.January, 31)
PRINT FormatDate(AddMonths(day, 1, DateOverflow.Constrain))
instant = ParseInstant("2024-01-01T01:00:00+01:00")
PRINT FormatInstant(instant)
PRINT FormatDuration(Nanoseconds(-1))
```

Date and TimeOfDay describe calendar components. LocalDateTime has no implicit timezone, so it is not an instant until an offset or zone resolves it. Instant identifies a point on the Unix timeline. Duration measures exact elapsed time; Period stores calendar months and days. They do not implicitly convert.

Adding a month to January 31 needs a policy: Reject by default or Constrain to the destination month's last day. A calendar day in a zone can differ from 24 elapsed hours. Duration stores wide signed floor seconds plus a normalized nanosecond fraction. Checked total accessors can fail when narrowing that wide value; keeping the Duration avoids unnecessary narrowing.

Zones and clocks

Load an explicit TZif file or construct an immutable zone database snapshot. No ambient process timezone is assumed. MapLocal distinguishes unique, ambiguous and nonexistent local times. ResolveLocal needs an explicit policy for overlaps or gaps and defaults to rejecting both. Offset and zoned equality include their documented representation/rule information; SameInstant compares timeline position.

Use wall time for timestamps and a monotonic clock for elapsed measurement. Wall time can jump after clock adjustment. Monotonic samples and deadlines belong to one Clock resource's domain; mixing domains is a checked error.

```
clock = NewTestClock(ParseInstant("2024-01-01T00:00:00Z"))
start = NowMonotonic(clock)
AdvanceTestClock(BYREF clock, Seconds(2))
PRINT FormatDuration(ElapsedBetween(start, NowMonotonic(clock)))
```

Test clocks make deadline and time arithmetic deterministic. They do not advance the task scheduler's real timeout clock. Sleep and SELECT still follow their documented runtime clocks. The reference covers strict ISO/wire parsers, formatting patterns, offsets, rules snapshots, calendar arithmetic and JSON mappings.

UUIDs

```
id = UUIDv5(UUIDNamespaceDNS(), "www.example.com")
PRINT FormatUUID(id)
PRINT ParseUUID(FormatUUID(id)) = id
```

UUID is an independent opaque value with read-only High/Low halves, equality, ordering, hashing and canonical JSON text. ParseUUID validates text; raw word/byte construction can preserve unknown version/variant bits. Nil and Max are values, not optional absence. UUIDv4 uses OS randomness; UUIDv7 combines time and random data under its documented clock behavior.

Name-based versions use an explicit namespace and name, as in the reproducible example above. The UUID reference lists all supported versions, formats and limits.

Reproducible randomness and secure bytes

```
generator = NewRandom(42, 54)
copy = generator
PRINT NextUInt32(BYREF generator) = NextUInt32(BYREF copy)
PRINT NextInteger(BYREF generator, 1)
```

Explicit PCG generators are independent records. Matching seed, stream and draw order give reproducible results. Copying one copies its sequence position, not shared mutable state. NextInteger uses an exclusive upper bound and checks that it is positive. NextDouble returns a value in [0,1). The legacy RANDOMIZE/RND interface uses a default shared sequence; task scheduling can affect its draw order.

These are simulation/application APIs. For security-sensitive bytes, use the OS source:

```
bytes = SecureRandomBytes(32)
FillSecureRandom(BYREF bytes)
PRINT LEN(bytes)
```

There is no weak fallback if the OS source fails. FillSecureRandom stages a complete replacement; failure/cancellation before commitment preserves the old bytes. Cancellation after commitment does not undo the replacement. Broader public cryptographic APIs remain separate work.

19. URLs, TCP and HTTP

Network programs still use ordinary functions, owned resources and checked errors. The runtime parks tasks on supported socket readiness operations. Your application chooses timeouts, message framing, supervision and how failures are reported.

Build a URL from data

```
DIM parameters AS DICT OF STRING TO STRING = DICT()
parameters["search"] = "Fred & Jane"
url = JoinURLPath("https://api.example.com/v1/?locale=en", "/customers", Query=parameters)
PRINT url
```

The helper appends the path and escapes query data. Query names supplied in the update replace existing values with those names, while unrelated parameters survive. A leading slash in JoinURLPath does not discard the existing base path. ResolveURL instead performs URI reference resolution. Choose the operation whose meaning matches your intent.

ParseURL gives a checked typed value. BuildURL and WithURL helpers construct components; SetURLQueryParameter updates through BYREF after staging a complete result. Raw components preserve encoded delimiters and presence; decoded accessors validate UTF-8. Ordered URLParameter lists preserve repeated names, bare flags and explicit empty values. Dictionary adapters serve simpler one/many-value cases. Typed URL values can be passed directly to HTTP clients.

Make an HTTP request

This example requires working DNS, network access and a peer server:

Example requirements: network program; requires reachable https://example.com and the matching TLS SDK.

```
response = HTTPGet("https://example.com/")
PRINT response.Status
PRINT DecodeUtf8(response.Body)
```

HTTPPost accepts text or bytes; RequestHTTP accepts explicit methods. Responses retain status, headers, body and trailers. Decode body text only when its encoding is appropriate. Typed JSON helpers distinguish HTTP status failure, a bodyless 204 response and a successful decoded value, including nested optional absence.

HTTPClientOptions combines protocol and TLS settings. Redirects are bounded; cross-origin changes strip authorization/cookies and HTTPS downgrade is forbidden by default. HTTPGetPlain rejects HTTPS and permits omitting TLS from a strictly plain-HTTP executable. HTTPS verifies certificates and hostnames by default.

Serve a route

Example requirements: server program; binds localhost port 8080 and runs until externally stopped.

```
router = HTTPRouter()
RouteGet(BYREF router, "/hello/{name}", Hello)
listener = ListenTCP("127.0.0.1", 8080)
ServeHTTP(listener, router)
FUNCTION Hello(request AS HTTPRequest) AS HTTPResponse
    RETURN TextResponse("Hello, " + Get(request.Parameters, "name", "world"))
END FUNCTION
```

Literal path segments take precedence over parameters and final remainder matches. Unknown routes yield 404; unsupported methods on a known path yield 405. HEAD uses GET fallback

without body bytes. Paths are decoded once and validated; do not add another unstructured decode step in a handler.

Buffered handlers return HTTPResponse. Streaming handlers use a connection and BYREF HTTPExchange with ReadHTTPBody, StartHTTPResponse, WriteHTTPBody and FinishHTTPResponse. Consume/drain a body before reusing its connection. Explicit lengths must match bytes written. Abandon a connection after framing errors; the buffered convenience client handles this lifecycle automatically.

UseHTTP adds middleware and HTTPFallback installs fallback handling. These use checked named callbacks. ServeHTTPChecked lets a supervisor observe original typed application/transport failures. Convenience servers recover handler failures, report their concrete names and send suitable error responses when possible. Mutable route slots admit Error, so checked callers must account for that family.

MountDirectory/ServeDirectory implement contained file serving with explicit listing policy, MIME types, conditional responses and supported single ranges. Review the reference's root-containment and hidden-file rules before exposing data.

Work below HTTP

ListenTCP/Accept and connection functions provide owned sockets. A task handling an accepted connection receives TAKE ownership; the connection closes after its handler and dependent operations finish. DNS uses bounded workers. TLS wraps the existing backend behind BASIL-owned options/resources; it does not expose OpenSSL internals as an application ownership model.

HTTP/1.1 is implemented. HTTP/2, WebSockets, automatic connection pools, proxies, cookie jars and transparent decompression are not part of this slice. The full HTTP reference gives framing, streaming, timeout and directory-serving limits; the TLS loopback example shows explicit transport.

20. Typed SQLite data

SQLite is useful for persistent local data and application state. BASIL's database connection is an owned resource. Ordinary queries can suspend while bounded workers execute them; the caller's text and bytes remain valid until completion or cancellation has been acknowledged.

Insert values safely

```
db = OpenDatabase(":memory:")
Query(db, "CREATE TABLE customers (name TEXT, visits INTEGER)")
Query(db, "INSERT INTO customers VALUES (?, ?)", [SqlText("Ada"), SqlInteger(3)])
result = QueryValues(db, "SELECT name, visits FROM customers")
PRINT SqlGetText(result.Rows[0].Values[0])
PRINT SqlGetInteger(result.Rows[0].Values[1])
```

Use placeholders and a typed LIST OF SqlValue to keep SQL structure separate from user data. Do not concatenate input text into SQL. SqlText and SqlInteger construct alternatives of a statically known choice; the parameter list is homogeneous even though its SQL payloads differ. Arbitrary mixed `["Ada", 3]` conversion is not implemented. Dictionary/record parameter adapters are future work.

Preserve NULL and storage classes

SqlValue distinguishes Null, Integer, Real, Text and Blob. NULL is not the same as an empty string, empty blob, integer zero or real zero. SQLite's returned runtime storage class determines the case; a column's declared affinity is not proof that every returned cell has that class.

```
db = OpenDatabase(":memory:")
result = QueryValues(db, "SELECT NULL, '', 0")
PRINT IsSqlNull(result.Rows[0].Values[0])
PRINT LEN(SqlGetText(result.Rows[0].Values[1]))
PRINT SqlGetInteger(result.Rows[0].Values[2])
```

Strict accessors reject the wrong kind rather than quietly converting text to numbers. Check row presence before indexing a result that may be empty. Use the actual typed accessor appropriate to your query and handle SqliteError. Values copied into BASIL results remain valid independently of SQLite's statement buffers.

Use QueryValues for typed results when there are no parameters. The two-argument Query overload is a compatibility text-or-NULL API; passing a LIST OF SqlValue selects the typed Query overload. Do not feed legacy text cells into typed accessors.

Transactions and tasks

The materialized result model is appropriate for bounded queries. Limit rows in SQL when application memory must stay bounded; streaming result objects are not implied by worker execution. A cancelled waiter does not prove the database statement was rolled back. Cancellation and transaction commitment are separate events.

Explicitly begin/commit/rollback using your SQL transaction policy. A user resource can own the database and perform best-effort rollback in __Cleanup, while a checked Commit operation reports application-visible success. The complete transaction example demonstrates this pattern.

Give a task its own connection or transfer an appropriate owner; general borrowed parent-state captures are unsupported. The periodic writer combines a child-owned connection, timeout, Cancel and Join. It observes the retained outcome even if the timeout wins, so operational failure cannot disappear behind cancellation.

21. Binary formats, compression and numerical work

Packed bytes, explicit numeric widths and independent-value collections provide the foundation for binary protocols and scientific utilities. Keep representation choices explicit: a byte offset is different from a Unicode position, and a machine word is not automatically an external wire encoding.

Read fixed-width byte values

```
DIM bytes AS LIST OF BYTE = [BYTE(1), BYTE(2), BYTE(3), BYTE(4)]
PRINT ReadUInt16LE(bytes, 0)
PRINT ReadUInt32LE(bytes, 0)
```

These little-endian readers accept arbitrary byte alignment and validate logical length. Invalid offsets or insufficient bytes report IndexError carrying the original offset. ReadUInt16LE/32LE/64LE return UINT32/UINT32/UINT64 respectively. They read an external representation without exposing raw pointers.

Unsigned types have checked ordinary arithmetic. Named wrapping, shifting, rotating, wide multiplication and layout helpers serve operations that deliberately need machine-width behavior. Use the unsigned reference for exact names and range rules. Do not rely on signed overflow to implement a binary algorithm.

Compress a byte buffer

The import path below is correct for a source saved in the repository root:

```
IMPORT "stdlib/compress/gzip.bas" AS gzip
original = EncodeUtf8("Hello, BASIC!")
packed = gzip.Compress(original)
restored = gzip.Decompress(packed, MaxOutput=1024)
PRINT DecodeUtf8(restored)
```

The imported codecs operate on LIST OF BYTE. DEFLATE, zlib, gzip and LZW encode and decode; bzip2 is decoding-only. LZW selects bit order and literal width. gzip supports checked metadata and individual member decoding. Compression levels choose effort, not identical output to every other implementation.

Always choose an appropriate MaxOutput for untrusted compressed data. The limit counts actual decoded bytes, including concatenated members. Header/checksum/size errors use CompressionError rather than returning a partly successful buffer. Preset dictionaries and reusable streaming codec objects are not exposed yet.

Hash bytes

```
IMPORT "stdlib/sha256.bas" AS sha256
PRINT sha256.HexDigest(EncodeUtf8("abc"))
```

SHA-256 Digest returns 32 bytes; HexDigest returns lowercase hexadecimal text. This is a digest facility, not password hashing, authentication or a complete cryptographic suite. A collection Hash is a separate operation with a different purpose; it is not a cryptographic digest.

Use numerical vectors

The scalar math library includes roots/powers, exponential/logarithmic functions, trigonometry in radians, angle conversion, rounding, fused multiply-add, binary decomposition/scaling and

adjacent-value operations. POW performs exponentiation; there is no ^ power operator. FLOOR/INT return a DOUBLE floor, whereas CINT truncates to INTEGER and checks range. ROUND uses nearest rounding with ties to even. FMOD and REMAINDER use different quotient rules. The included math reference lists every family and its special values.

```
a = [1.0, 2.0, 3.0]
b = [4.0, 5.0, 6.0]
PRINT Dot(a, b)
PRINT Norm([3.0, 4.0])
PRINT FloorDiv(-17, 16)
PRINT EuclideanMod(-17, 16)
```

Dot, Norm, Normalized, Distance, CosineSimilarity, Scaled and component-wise VectorAdd/Subtract/Multiply/Divide accept supported numeric lists and rank-one arrays. Inputs borrow; returned vectors own new list values. Shape mismatches and invalid zero norms are checked. Integral operations retain width/range checks; floating operations retain IEEE NaN/infinity behavior.

For coordinates, FloorDiv and EuclideanMod give floor-based quotient/nonnegative remainder semantics, unlike ordinary truncating integer division. FloorToInteger explicitly converts a floating floor. Use the vector and floating references for empty inputs, precision, nonfinite values and scheduling behavior.

Accumulate bytes and text efficiently

ByteBuffer stores initialized bytes separately from reserved capacity. StringBuilder does the same for valid text. Reserve is a performance hint, not a length change. Clear retains capacity; TakeBytes/TakeString extract the accumulated value and empty the builder. ToBytes/ToString take independent snapshots instead.

```
builder = StringBuilder(Capacity=128)
Append(BYREF builder, "Hello, ")
Append(BYREF builder, "λ")
PRINT Length(builder)
PRINT ByteCount(builder)
PRINT TakeString(BYREF builder)
PRINT Length(builder)
buffer = ByteBuffer(Capacity=16)
Extend(BYREF buffer, EncodeUtf8("AB"))
snapshot = ToBytes(buffer)
WriteAt(BYREF buffer, 0, [BYTE(90)])
PRINT DecodeUtf8(snapshot)
PRINT DecodeUtf8(Bytes(buffer))
```

Bytes(buffer) can lend a read-only view directly to a consumer. That loan keeps the source alive and prevents conflicting mutation or transfer during the call. For example, extending a buffer from its own borrowed view conflicts; take a snapshot first. Builders themselves are copyable independent values. Byte offsets are nonnegative initialized positions, not Python-style negative string indexes. WriteAt replaces existing bytes without extending the buffer; Truncate only shrinks. The buffer reference covers AppendChar, AppendLine, spans and capacity operations.

Read and write binary fields with a cursor

ByteWriter removes manual offset arithmetic when assembling a packet. ByteReader tracks a checked position while parsing it. The default byte order is little endian; select big endian for protocols that require it.

```
writer = ByteWriter(Capacity=32)
WriteUInt32(BYREF writer, UINT32(42), Order=ByteOrder.BigEndian)
WriteInt16(BYREF writer, -7)
```

```
WriteBytes(BYREF writer, EncodeUtf8("OK"))
bytes = TakeBytes(BYREF writer)
reader = ByteReader(TAKE bytes)
PRINT ReadUInt32(BYREF reader, Order=ByteOrder.BigEndian)
PRINT ReadInt16(BYREF reader)
PRINT DecodeUtf8(ReadBytes(BYREF reader, 2))
PRINT Remaining(reader)
TRY
    value = ReadByte(BYREF reader)
CATCH err AS IndexError
    PRINT Position(reader)
END TRY
```

Readers snapshot ordinary input; TAKE adopts a named byte list. Copying a reader forks its position and contents. Seek and Skip are checked. A failed field read leaves the position unchanged; ReadInto also validates its full destination range before mutation. Cursor ReadByte requires a byte and fails at EOF, unlike the OPTIONAL byte returned by a buffered stream. Signed 8/16/32/64, unsigned 16/32/64 and exact binary64 double fields have corresponding read/write calls. Narrow writes check their range before changing output. ReadDouble/WriteDouble preserve bit patterns, including signed zero and NaN payloads.

22. Building complete applications

The features in earlier chapters fit together without a separate application language. A small program can start with top-level statements, grow into modules, and add checked resources or tasks where they help.

A complete word-count utility

This program reads lines until EOF and counts space-separated words without assuming that SPLIT's omitted separator means whitespace. Its input for the checked example is two lines, `red blue` and `red`.

```
DIM counts AS DICT OF STRING TO INTEGER = DICT()
lines = 0
DO
    IF LET line = ReadLine() THEN
        lines += 1
        AddWords(BYREF counts, line)
    ELSE
        EXIT DO
    END IF
LOOP
PRINT "Lines: "; lines
FOR EACH word, count IN counts
    PRINT word; ": "; count
NEXT
SUB AddWords(BYREF counts AS DICT OF STRING TO INTEGER, line AS STRING) FAILS StringError
    FOR EACH word IN SPLIT(LOWER(line), " ")
        IF word <> "" THEN
            counts[word] = Get(counts, word, 0) + 1
        END IF
    NEXT
END SUB
```

Output:

```
Lines: 2
red: 2
blue: 1
```

The dictionary owns the counts; AddWords borrows it exclusively for updates and borrows the line for reading. Each loop's bindings have clear scopes. Input EOF is normal optional absence. Dictionary iteration preserves insertion order, so the first encountered word is reported first.

This is deliberately a space-delimited counter. Punctuation remains part of a word, tabs are not separators and LOWER is not a complete linguistic tokenization policy. Extending it is an application decision: use Regex for a chosen word definition, Sorted(Keys(counts)) for alphabetical output, or a record for options.

Give a command-line tool typed options

```
TYPE Options FORMAT CommandSettings { Name="demo", Description="A small tool" }
    Frames AS INTEGER = 10 FORMAT OptionSettings { Min=0, Max=1000, Help="Work limit" }
    Verbose AS BOOLEAN = FALSE
END TYPE
TRY
    SELECT TYPE ParseArguments(Options {})
    CASE ArgumentResult.Help AS help
        RETURN PrintArgumentHelp(help)
    CASE ArgumentResult.Parsed AS parsed
```

```
        options = ArgumentOptions(parsed) AS Options
        PRINT options.Frames
    END SELECT
CATCH usage AS ArgumentError
    RETURN ReportArgumentError(usage)
END TRY
```

Run the resulting program with `--frames 25`, `--verbose` or `--help`. Help is a normal result; parse errors report usage and return status 2 through the chosen helper. Neither helper exits behind the application's back. Supported record option types are scalar text/integer/real/Boolean and optional forms. Use the dynamic Command API for subcommands, positionals and forwarding.

CommandLineParameters returns application arguments without argv[0]; the lower-level CommandLineArguments includes it. Explicit-list ParseArguments overloads are pure and expect the executable name already excluded.

Add useful logging

Logging records what a running application did and why. `LogInfo("Started")` needs no setup: the default writes UTC timestamp, level and message to stderr at the Info threshold. Levels are Trace, Debug, Info, Warn, Error and Fatal. Fatal is severity only; it does not exit the process.

This example chooses deterministic output for testing:

```
config = DefaultLogConfig()
config.Console = LogConsole.StandardOutput
config.Format = "{level} {message}"
SetLogConfig(config)
LogDebug("Not enabled")
LogInfo("Processed {0} items", 3)
LogWarn("Request {0} took {1:.2f}ms", 42, 3.125)
LogFatal("Still returning normally")
PRINT "Alive"
```

Numbered message fields share scalar f-string format modifiers; repeated fields reuse one computed argument. Double braces escape literal braces. Literal message patterns are checked at compilation; malformed dynamic patterns are reported at runtime. Layouts use named metadata: timestamp/date/time, level, message, file, line, function and logger. Severity calls supply source metadata automatically; explicit WriteLog does not fabricate caller metadata.

Defer expensive disabled messages

Ordinary argument expressions evaluate even when filtered. A zero-argument function value defers its computation until after filtering:

```
config = DefaultLogConfig()
config.Console = LogConsole.StandardOutput
config.Format = "{message}"
SetLogConfig(config)
number = 7
deferred = () => Compute(number)
number = 99
LogDebug("{0}", deferred)
SetLogLevel(LogLevel.Debug)
LogDebug("{0} twice: {0}", deferred)
FUNCTION Compute(value AS INTEGER) AS INTEGER
    PRINT "Computing"
    RETURN value * 2
END FUNCTION
```

The disabled call does not print Computing. The enabled call invokes the function once and produces `14 twice: 14`. The closure captured 7 when constructed, so the later assignment does not

change it. Capture construction is still eager work; reuse an existing closure in a hot loop. LogCallable supplies a function label without invoking the function.

Deferred computation errors retain their original checked types. Severity calls report formatting/destination failures to stderr and continue other destinations. Use checked WriteLog when the caller needs LoggingError after destination attempts; LogArgument constructs typed choices for its heterogeneous formatting-value list. Configuration, Flush and Close also declare LoggingError. Cancellation remains cancellation.

Give services and destinations deliberate owners

CreateLogger(config) returns a scoped resource; CreateLogger(path) appends to a file with console output disabled. LogHandle(owner) gives a copyable access capability, not another owner. A handle does not keep a scoped logger alive or close it when dropped. Closed/stale handles are checked. GetLogConfig returns an independent value; SetLogConfig validates replacement before committing it.

SetDefaultLogger(TAKE owner) transfers ownership to program lifetime. Previously promoted services stay alive for existing handles even when the default selection changes. Program cleanup finishes children and ordinary locals before closing these services. This explicit shared service does not change independent-value semantics for ordinary application data.

LogRecord contains Timestamp, Level, Message, File, Line, FunctionName, Logger and Formatted. AddLogSink can register a stateless handler. For mutable or resource state, CreateLogSink takes an explicit independent owned context, a handler and an optional finishing callback:

```
TYPE CountedOutput
    Count AS INTEGER
END TYPE
config = DefaultLogConfig()
config.Console = LogConsole.None
SetLogConfig(config)
sink = CreateLogSink(CountedOutput { Count=0 },
    SUB(BYREF context AS CountedOutput, record AS LogRecord)
        context.Count += 1
        PRINT context.Count; ": "; record.Message
    END SUB)
AddLogSink(TAKE sink)
LogInfo("first")
LogInfo("second")
```

The sink owns CountedOutput and lends exclusive writable access to the handler. This avoids a mutable closure capture. Named resource contexts require TAKE. The optional finisher runs on checked Flush/Close and automatic service closure; ordinary cleanup then destroys the context. ClearLogSinks releases application sinks; the configured file destination is separate.

Sinks run synchronously in registration order and may suspend. Other tasks' logger operations wait for exclusive service access. Same-service recursive logging reports failure rather than waiting on itself; cycles across different services still require normal deadlock avoidance. There is no logging thread or asynchronous record queue. Unused logging and unused file/formatting adapters are omitted from the executable. See the logging reference for all overloads, failure rules and recorded measurements.

Use the BASIL IDE

```
python3 tools/build_basic_ide.py
./basic-ide hello.bas
```

F1 shows help, F2 saves, F4 builds, F5 builds/runs and Ctrl+Space opens the compiler-owned standard-library browser. F6 navigates procedures/types; Ctrl-F searches and Ctrl-G goes to a line. F12 cycles themes. Ctrl-Q checks unsaved changes. The editor preserves UTF-8, common line endings and file permissions, and reports external save conflicts.

GDB-backed debugging starts with F7. F9 toggles breakpoints, F8 steps into, F10 steps over and F5 continues. GDB must be installed and tracing allowed. Aggregate inspection, task-aware stepping and complete source-level debugging of hosted/package code remain limited. Consult the IDE guide for current keys and boundaries.

Create a desktop application

The GTK package supplies opaque owned windows/widgets and queued application events. Build it with `python3 tools/build_gtk.py build/selfhost/stage3`; declare the dependency alias Gtk in your project. GTK and its development/system libraries are separate host dependencies.

The following is a queued-event application skeleton, requiring that dependency and a graphical display:

Example requirements: GUI program requires a Gtk project dependency, GTK system libraries and a display.

```
IMPORT Gtk
window = Gtk.NewWindow("Hello", 480, 240)
app = Gtk.NewApplication(window)
pump = SPAWN Gtk.Pump(Gtk.EventSource(app))
button = Gtk.NewButton("Click me")
Gtk.SetChild(window, button)
clicked = Gtk.On(app, button, "clicked", 1)
Gtk.Present(window)
WHILE Gtk.IsRunning(app)
    event = Gtk.NextEvent(app)
    IF Gtk.Kind(event) = Gtk.EventKind.CloseRequested THEN
        Gtk.Quit(app)
    ELSEIF Gtk.Command(event) = 1 THEN
        Gtk.SetLabel(button, "Hello, BASIC!")
    END IF
WEND
```

Keep the subscription owner alive while receiving events. The native signal only queues data; application work runs later in ordinary BASIL and can use state, checked failures and tasks. Direct C-stack handlers have stricter NOSUSPEND rules. Widgets own native references rather than independent snapshots of every property; a read-only borrow does not freeze GTK state.

The toolkit includes layout, menus, model-backed lists/tables/trees, dialogs, text buffers, drawing, clipboard and background files. Mixed document snapshots retain original image assets; bounded history supports grouped undo/redo and saved state. Image paste checks the captured revision/selection before committing, so a delayed request cannot overwrite newer edits. Full stylesheet owners and action subscriptions release their registrations on scope exit. The Notepad application and mixed-document editor show these pieces working together; the GTK package reference lists the supported API families.

Graphics and games

The raylib package provides an explicit rendering loop. Build the pinned package using `python3 tools/build_raylib.py build/selfhost/stage3` and declare Raylib as a project dependency. The example requires a display and its native/system inputs:

Example requirements: graphics program requires a Raylib project dependency and a display.

```
IMPORT Raylib
Raylib.InitWindow(640, 400, "BASIC graphics")
Raylib.SetTargetFPS(60)
WHILE NOT Raylib.WindowShouldClose()
    Raylib.BeginDrawing()
    Raylib.ClearBackground(Raylib.Rgba(12, 18, 32))
    Raylib.DrawCircle(320, 200, 70, Raylib.Rgba(64, 220, 190))
    Raylib.EndDrawing()
```

```
WEND
Raylib.CloseWindow()
```

Match drawing begin/end calls, check input each frame and use GetFrameTime for time-based motion. There is one coordinator-owned window. The binding includes procedural 3D cameras/meshes and checked draw states, with details in its reference. The complete Space Invaders project demonstrates options, packages and a game loop. Upstream raylib's entire API is not automatically exposed by this binding.

Create images without a window

Canvas supplies native images, drawing, image codecs and fonts without importing a GUI library. It is useful for charts, thumbnails, reports and generated assets; it does not open a window or start an event loop. Its APIs are available directly.

```
image = NewCanvas(64, 32)
ClearCanvas(BYREF image, RGB(20, 30, 40))
FillRect(BYREF image, 2.0, 2.0, 12.0, 8.0, RGB(255, 0, 0))
DrawBitmapText(BYREF image, "BASIL", 2, 16, RGB(255, 255, 255))
copy = CloneCanvas(image)
SetPixel(BYREF copy, 0, 0, RGB(0, 255, 0))
PRINT GetPixel(image, 0, 0).Red
PRINT GetPixel(copy, 0, 0).Green
png = EncodePNG(image)
restored = DecodePNG(png)
PRINT CanvasWidth(restored)
PRINT CanvasHeight(restored)
```

Coordinates start at the top left, with X increasing right and Y down. A new canvas is transparent black. Colors hold straight RGBA8 channels; RGB supplies opaque alpha, while RGBA accepts it explicitly. GetPixel/SetPixel check bounds; drawing operations clip. Geometry uses DOUBLE pixel-edge coordinates. A Canvas owns its pixel storage: ordinary arguments borrow, BYREF mutates, CloneCanvas makes a new owner, and TAKE transfers a named owner. CanvasBytes makes an independent byte snapshot. Assigning the resource does not silently copy it.

EncodePNG returns bytes. WriteFileAtomic(path, bytes) saves them, or use SaveImage(image, "picture.png") for extension-selected output. LoadImage and DecodeImage inspect signatures. Direct codec calls select only the required format; a dynamic format choice retains its possible codecs. Literal/CONST choices can eliminate unused codec code and tables.

Paths, transforms and image processing

```
image = NewCanvas(16, 16)
path = NewPath()
MoveTo(BYREF path, 1.0, 1.0)
LineTo(BYREF path, 14.0, 1.0)
LineTo(BYREF path, 1.0, 14.0)
ClosePath(BYREF path)
FillPath(BYREF image, path, RGB(255, 0, 0))
small = ResizeCanvas(image, 8, 8, Sampling.Nearest)
PRINT CanvasWidth(small)
PRINT GetPixel(image, 2, 2).Red
```

Paths support lines, quadratic/cubic curves, filling, strokes and dashed strokes. Stroke describes width, cap, join, miter limit and antialiasing. Fill rules select how overlapping contours count as inside. Antialiasing offers Off, Standard and High. DrawImage composites a borrowed source into a mutable destination, choosing BlendMode and encoded-sRGB or linear-light blending. For an overlapping copy inside the same owner use CopyWithinCanvas; separate read/write arguments naming the same owner conflict under ordinary borrowing rules.

ResizeCanvas, CropCanvas, Thumbnail, FlipCanvas, RotateCanvas and WarpCanvas return new owners. Nearest, Bilinear, Area, Bicubic and Lanczos3 sampling offer different quality/cost tradeoffs. Transform helpers compose/invert coordinate maps. AdjustedCanvas, grayscale/inversion, blur, sharpening, edge/emboss and custom 3x3 convolution support processing pipelines. Histogram, AverageColor, AnalyzeCanvas, AlphaBounds and ColorBounds inspect images. The Canvas reference groups these operations; the public API appendix gives their checked signatures and defaults.

Formats and animation

PNG/BMP are lossless image choices; JPEG supports quality-selected baseline output with an explicit background for transparency. GIF and WebP support still and animated images; APNG supplies animated PNG. WebP writing is lossless; reading also supports VP8 lossy input. These are defined format profiles, not a promise to accept every file variant. AVIF and TIFF enum members are reserved: capability queries return false. See the image-format reference for exact limits.

```
frame = NewCanvas(4, 4)
ClearCanvas(BYREF frame, RGB(255, 0, 0))
animation = NewAnimation(4, 4)
AppendAnimationFrame(BYREF animation, frame, 100, 1000)
SetPixel(BYREF frame, 0, 0, RGB(0, 255, 0))
savedFrame = AnimationFrame(animation, 0)
PRINT GetPixel(savedFrame, 0, 0).Red
PRINT CanEncodeImage(ImageFormat.AVIF)
```

AppendAnimationFrame stores an independent snapshot, and AnimationFrame returns a fresh Canvas. Delays retain rational numerator/denominator timing. Use EncodeAnimation/SaveAnimation and DecodeAnimation/LoadAnimation for animation; still-image decoding rejects animations rather than dropping frames. There is no implicit playback task. Coalesced frames account for disposal/blending rules.

CanvasError reports Message, Offset and Format for malformed input, unsupported profiles and configured limits. File operations also declare IoError; helper error contracts remain visible. Choose pixel, encoded-size, frame and work limits for your inputs. They bound logical operations/storage, not total process memory or a wall-clock deadline. Normal scheduling, cancellation and deterministic cleanup apply. New-image operations leave their source unchanged on failure; arbitrary in-place drawing does not promise rollback.

Fonts and prepared text

DrawBitmapText uses a built-in ASCII font and needs no asset. For proportional or multilingual text, load a licensed static OpenType TrueType or CFF1 font. The following example requires your own font file; it is not a portable asset-free program.

Example requirements: requires a user supplied licensed font file.

```
font = LoadFont("fonts/body.ttf")
layout = CreateTextLayout(font, "Hello, café", TextLayoutOptions {})
image = NewCanvas(320, 80)
ClearCanvas(BYREF image, RGB(255, 255, 255))
DrawTextLayout(BYREF image, layout, 10.0, 10.0, RGB(0, 0, 0))
SaveImage(image, "greeting.png")
```

Font and TextLayout are owned resources. A prepared layout owns its outlines and can outlive its source font; repeated drawing does not reshape the text. NewFontSet plus AddFont(BYREF fonts, TAKE font) creates an ordered fallback set. CreateTextLayout accepts either one Font or a FontSet. Measurement, line/glyph queries and hit testing refer back to original scalar positions.

Layout provides Unicode 17 grapheme boundaries, bidi ordering, normalization for shaping, whole-grapheme fallback, wrapping and alignment. GraphemeBoundaries, ResolveBidiLevels and BidiVisualOrder expose text analysis separately. This does not change ordinary STRING indexing

into grapheme indexing. OpenType substitutions, positioning, kerning, marks and supported joining/reordering stages work without FreeType or HarfBuzz at runtime. Variable/CFF2/color fonts, WOFF and hint execution are unsupported. Missing glyphs use a replacement unless strict checking is requested. DiscoverSystemFonts/FindFont help locate fonts, but bundle fonts for repeatable output; discovery locations currently use Linux defaults. The font reference explains format, layout and resource limits.

23. Native interfaces and advanced library design

Most application code can stay within safe BASIL libraries. A native interface is useful for an existing C library, a platform operation or a carefully reviewed performance boundary. It also introduces obligations that static BASIL checks cannot establish for arbitrary foreign code.

Separate the C ABI from BASIL values

EXTERN imports supported scalar C signatures from objects/archives; EXPORT emits C wrappers. The C ABI is the platform's calling convention and representation agreement. BASIL's internal ABI includes its own value/ownership/error contracts. Do not pass a BASIL record or string header to a C function merely because its bytes appear convenient.

Raw C interfaces support the documented scalar subset, not arbitrary structs, varargs, BYREF aggregates or fallible C signatures. BOOLEAN/C _Bool differs from C int and GObject gboolean. A reviewed adapter performs those conversions. The native adapter example builds a complete C producer, generated bridge and safe BASIL wrapper.

Pointers and unsafe authority

POINTER is a non-owning address, distinct from INTEGER. It neither frees storage nor proves that storage is valid. NullPointer and IsNullPointer are safe; reading or converting an address requires an explicit unsafe block.

```
pointer = NullPointer()
PRINT IsNullPointer(pointer)
UNSAFE
    PRINT PointerAddress(pointer)
END UNSAFE
```

This example only inspects a null address. Dereferencing it would be invalid. PointerFromInteger does not create valid storage. Peek/Poke require the caller to establish readable/writable initialized memory, alignment/representation and the absence of conflicting accesses. A pointer does not extend its target's lifetime.

UNSAFE grants specific foreign capabilities; it does not disable type, definite initialization, borrowing or checked-failure rules. An UNSAFE FUNCTION imposes caller obligations but still needs explicit unsafe blocks for its own operations. Unsafe Main and conversion of unsafe functions to safe delegates are rejected.

Callbacks across native frames

CALLBACK describes a generated noncapturing C ABI entry. Its target must be infallible and nonsuspending, cannot start tasks, and cannot allow cancellation to unwind across C. It is passed to C; it is not an ordinary BASIL callable you invoke directly. The current scalar C adapter accepts at most six parameters.

CALLABLE NOSUSPEND retains the BASIL ABI for reviewed adapters. It can have checked BASIL failures, but still cannot suspend, cancel or start tasks transitively. Neither kind captures locals. Foreign-thread entry is rejected rather than automatically attaching a new thread to BASIL. For ordinary GUI applications, queued events avoid placing application code on a native callback stack.

Publish a safe wrapper

A wrapper should own the foreign handle, hide its representation with OPAQUE TYPE, validate lengths and kinds, and expose borrowed access or explicit ownership transfer. Native code must

not retain a borrowed BASIL buffer after return or suspend behind the runtime's back. Converting a foreign address to POINTER does not make retention safe.

The versioned `basic_ffi_v1.h` helpers provide checked UTF-8/packed BYTE construction, matching destruction, supported owned results and rich native errors. The adapter generator declares borrowing, placement, return ownership and local error tags. The GTK extension helpers similarly type-check native GObjects instead of exposing private handle layouts. Follow the adapter reference and its executable example when building a new binding.

Distribute compiled libraries

Use a library project and .blib when consumers are BASIL programs. Use the documented object/archive and C wrappers for a C host; these are different products. Package interfaces preserve failures, callback contracts, ownership, cleanup hooks and suspension effects. The current toolchain preserves checked ownership contracts for native resources and containers. Native File owners, resource records, lists, optionals, dictionary values and arrays can cross compatible BASIL library boundaries with the same borrows and explicit transfers as source calls. Generic templates are checked under consumer types. ABI v15 adds Display and embedded-value support while consuming supported v14 packages. Older readers reject v15 output; rebuild or upgrade with a compatible compiler/SDK when requested by diagnostics.

Save this source module as `storage.bas` beside its entry:

```
PUBLIC FUNCTION OpenInput() AS File FAILS IoError
    RETURN OpenFile("/dev/null")
END FUNCTION
PUBLIC FUNCTION Size(input AS File) AS INTEGER FAILS IoError
    RETURN GetFileInfo(input).Size
END FUNCTION
PUBLIC SUB Consume(TAKE input AS File) FAILS IoError
    PRINT GetFileInfo(input).Size
END SUB
```

```
IMPORT "storage.bas" AS storage
input = storage.OpenInput()
PRINT storage.Size(input)
storage.Consume(TAKE input)
```

The book checks this source-import example. To distribute these declarations, build a library-manifest .blib and select it through a consumer dependency alias as in chapter 12. The package regression suite separately checks source/relocated prebuilt equivalence after removing producer sources. Consumers validate passing modes, borrow origins, effects and cleanup contracts before emitting calls. Missing/mismatched metadata produces a rebuild diagnostic.

Imported BORROW results protect every possible origin through suspension; owned returns transfer one owner. BYVAL and TAKE resource arguments need fresh or explicitly moved owners. Tasks retain their normal cleanup and Join rules. Resource keys/sets, escaping channel owners and raw C resource passing remain unsupported. Compatibility metadata does not authenticate arbitrary foreign machine code. See compiled resource contracts.

Static native inputs must use supported ELF/archive forms. A declared system-library dependency uses the hosted linker bridge and matching SDK. TLS, constructors, IFUNC and arbitrary runtime assumptions are not automatically handled by the integrated freestanding linker. Keep notices/licenses with distributed inputs.

This chapter introduces the trust boundary; it does not replace the foreign-interface contract, compiler ABI reference or binding-specific review.

24. Diagnostics, limits and migration

A compiler rejection often describes a mistaken type, unavailable value or conflicting access before it becomes a runtime problem. Reduce the example until the ownership and data flow are visible, then fix the cause rather than hiding it behind a broader type or unsafe operation.

Common problems

Symptom	Explanation and next step
Condition requires BOOLEAN	Compare explicitly, use a Boolean predicate, or use IF LET for optional presence.
Read before initialization	Assign a complete value on every reaching path. NONE initializes an optional, not every type.
Incompatible numeric types	Convert deliberately and check the target range.
Cannot copy a resource	Borrow for temporary use or TAKE when ownership truly transfers.
Value unavailable after TAKE	Reinitialize before reuse; a transfer is not a copy.
Conflicting argument/borrow	Split operations, snapshot copyable input, or redesign the helper to take one owner.
Mutation during iteration	Build a separate result or finish the protected iteration before editing.
Function can fall through	Return a correctly typed value on each successful path.
Escaping checked error	Catch locally or add the precise FAILS dependency. Discarding a success result does not discard error.
Generic call lacks context	Supply a typed destination or supported AS ascription.
Empty collection factory lacks context	Supply the concrete generic type on the destination; later insertions do not infer it.
Lambda cannot capture a value	Only copyable free locals can be snapshotted; pass resource context explicitly.
Cache clock/mode error	Match ordinary versus At operations, retain one clock domain, and use nondecreasing mutating samples.
Unknown package/incompatible ABI	Check the manifest alias and rebuild with the selected compiler/SDK.
Program hangs on Join	Complete the protocol, close the producer, or cancel and join; scope cleanup cannot invent missing messages.

Unexpected string or index behavior

Positions start at zero. STRING positions count Unicode scalars, not UTF-8 bytes or graphemes. Slices exclude their end; DIM array bounds include it. An omitted SPLIT separator splits into scalars, not words. A dictionary lookup with a fallback evaluates the fallback eagerly. These are deliberate rules, so make boundary examples part of application tests.

Direct list indexing and array coordinate errors have fatal policies. Named APIs such as checked editing operations or packed-byte readers may report IndexError. Read the exact operation's contract rather than assuming TRY catches every invalid index. Floating NaN/infinity are ordinary numeric results until explicitly validated; they are rejected where an operation requires a valid hash key or finite JSON.

Unexpected task behavior

Output from independent tasks can interleave. Use messages and parent-observed results when ordering matters. COMPLETED is readiness; Join observes the outcome. Cancel is a request; Join waits through acknowledgement. A timeout cannot erase an earlier operational error. Resource results transfer once, while repeated copyable joins return independent values.

Native calls and some filesystem operations can delay coordinator progress. Increasing SPAWN count does not make an unverified foreign call run on arbitrary CPU workers. General borrowed

captures and shared writable parent state remain rejected even when a programmer expects a task to finish quickly.

Current implementation boundaries

Linux x86-64 is the supported target. Compiler checks include parser nesting, parameter-word limits, error tags, choice alternatives and target frame encoding. Local metadata grows with the function; older fixed-local limits in historical reports are not current language promises. Native and coroutine stacks remain finite, and stack exhaustion is fatal. Avoid huge automatic frames and unbounded recursion, especially in task entries.

Not implemented: multiple successful return values, local/anonymous records, generic record extension/errors/choices/cleanup hooks, resource or borrowed lambda captures, mutable shared capture cells, general stored references, mutable module globals, resource keys/sets, indexed partial field holes, borrowed or BYREF task arguments, YAML/general TOML/XML codecs, remote package fetching, lockfiles, general incremental semantic caching and unrestricted foreign-thread callbacks. The current checklist owns remaining work.

Historical design files often contain proposed spellings. Do not assume those examples are accepted by today's compiler. The maintained language specification, passing tests and current implementation establish support; requirements express future intent. This book does not silently add unsupported syntax to make an example shorter.

Find an API or report a problem

`basic api` emits the public prelude declaration catalog. Ctrl+Space in the IDE browses those overloads. The generated catalog appendix in this book comes from the compiler used for its build; imported package APIs and compiler intrinsics are not all represented by that catalog.

When reporting a problem, include a minimal source file, exact build/run command, expected and observed behavior, and `basic toolchain` output. Preserve the original diagnostic. For a runtime issue, include input that reproduces it and whether cancellation or native libraries are involved. Do not publish credentials or private input files just to make a report complete.

25. Glossary and topic index

Use this index to return to the teaching chapters. The PDF contents and links provide navigation without relying on page numbers that become stale after edits.

Term	Meaning	Read
ABI	Binary agreement between compiled components about calls and representations	Native libraries
Alias	Another access path to the same storage	Ownership
Array	Dense owned rectangular coordinate space	Arrays
Ascription	Checked type context supplied with AS; not a conversion	Generics
BOM	Initial byte-order mark, handled by an explicit encoding policy	Strings
Canvas	Owned RGBA8 image with checked drawing and codecs	Applications
Byte cursor	Checked position within a binary packet	Binary formats
Borrow	Temporary access while another value retains ownership	Ownership
BYREF	Exclusive writable access to caller storage	Functions
BYVAL	An independent writable local owner	Ownership
Callback	Function supplied as behavior to another operation	Generics
Cache	Bounded key/value working set with recency and/or expiry policy	Collections
Cancellation	Request to stop a task through its cleanup path	Tasks
Channel	Scoped typed message exchange with sender/receiver capabilities	Channels
Choice	A value holding exactly one declared alternative	Data types
Cleanup	Deterministic release of owned state	Resources
Closure	Function value owning independent snapshots of copyable free locals	Generics
Compile	Check source and produce a native artifact	Getting started
Coroutine	An execution that suspends/resumes with its call state	Tasks
Definite initialization	Proof that a value is assigned before every read	Values
Dictionary	Typed key-to-value collection	Collections
Deque	Sequence with insertion/removal at either end	Collections
Duration	Exact elapsed time, distinct from calendar months/days	Time
Display	Explicit canonical text representation of a nominal value	Generics
EMBED	Checked read-only value prepared from input while building	Serialization
Enum	Distinct integer-backed type with qualified named members	Data types
Error family	A checked error type and its descendants	Errors
Exclusive	Protected against other conflicting access	Ownership
FAILS	Declaration of escaping checked failures	Errors
FORMAT	Typed constant options attached to a declaration	Serialization
Generic	Operation checked for statically selected concrete types	Generics
Hash	Value used for collection indexing; not a cryptographic digest	Generics
Instant	A point on the Unix timeline	Time
Join	Wait for and observe a task's retained outcome	Tasks

Term	Meaning	Read
Lifetime	Interval during which storage is valid for access	Ownership
Lambda	Anonymous expression or block function	Generics
List	Ordered growable homogeneous sequence	Collections
LoggerHandle	Copyable access capability for a service owned elsewhere	Applications
LRU	Cache eviction by least recent use	Collections
Module	Source declarations addressed through an import alias	Modules
Move	Explicit ownership transfer, usually TAKE	Resources
NaN	Floating value unequal to every value, including itself	Values
Nominal	Type identity follows its declaration, not just field shape	Data types
Opaque	Publicly nameable type with module-controlled representation	Modules
Optional	NONE or SOME(payload), distinguishing absence from empty data	Data types
Optional fallback	Lazy ?? selection, evaluating the right side only for NONE	Data types
Owner	Value responsible for keeping storage/resource alive and releasing it	Ownership
Period	Calendar months/days, distinct from exact elapsed Duration	Time
Predicate	Function returning BOOLEAN	Collections
Resource	Noncopyable owner with deterministic release responsibility	Resources
Scope	Lexical region controlling visibility and cleanup	Control flow
Shadowing	Explicit DIM local reusing an ordinary/BYVAL parameter name	Functions
SELECT	Value/choice matching or scheduler event selection, according to form	Control flow, Channels
Set	Unique typed members with coherent equality/hash	Collections
Slice	Independent selected subsequence, using half-open bounds	Collections
Snapshot	Independent value captured at a defined evaluation point	Ownership
SPAWN	Start a task with checked argument snapshots/transfers	Tasks
Static type	Type known and checked before running	Values
Suspension	Parking execution until work/event/readiness permits resumption	Tasks
TAKE	Explicit transfer making the source unavailable	Resources
Task	Owned execution with success/error/cancellation outcome	Tasks
TTL	Expiry interval since the cache entry's most recent write	Collections
Type graph	Finite declared-type descriptors with graph-local references	Serialization
Unicode scalar	A text position unit, different from bytes and graphemes	Strings
Unsafe	Explicit authority for operations whose foreign obligations need review	Native libraries

Learning routes

For a command-line utility, read chapters 1-13, then 16, 17 and 22. For a network service, add 14-15 and 19. For a desktop application, learn ownership and errors before the GUI material in 22. For native binding work, read 9-13 and 23, then the exact foreign and adapter contracts linked there.

Library reference

The following references cover public APIs, examples, costs and restrictions. Compiler implementation notes and internal layouts are maintained separately in the contributor documentation.

String library reference

Strings are valid UTF-8 independent values. Default parameters borrow without copying; results own their storage. Positions, lengths and padding widths below count Unicode scalar values, starting at **zero**. A combining sequence or emoji may contain several scalars. Equality and search are exact and case-sensitive. No implicit normalization, locale tailoring, or numeric conversion occurs.

Functions

All integer arguments/results use INTEGER (signed 64-bit). Fallible calls are checked automatically, including calls whose successful results are discarded; inline TRY is optional.

Function	Result and behavior	Declared error
LEN(text), ByteCount(text)	Scalar count; UTF-8 byte count	None
ByteAt(text, offset)	Encoded byte at zero-based byte offset; invalid offset is fatal	None
LEFT(text, count), RIGHT(text, count)	At most count scalars from either end	StringError
MID(text, start, count)	At most count scalars starting at start; start may equal LEN	StringError
INSTR(text, needle)	First scalar position, or −1; empty needle gives 0	None
INSTRFROM(text, needle, start)	First scalar position at/after start; empty needle gives start	StringError
RINSTR(text, needle)	Last scalar position, or −1; empty needle gives LEN	None
CHR(value)	One Unicode scalar encoded as UTF-8, including NUL	StringError
ASC(text)	First Unicode scalar’s integer value	StringError
STR(value)	Canonical integer or shortest round-tripping DOUBLE spelling, no leading space	None
VAL(text)	Strict signed decimal integer after Unicode whitespace trimming	NumericError
SPACE(count), REPEAT(text, count)	Repeated spaces or text; zero gives empty string	StringError
UPPER(text), LOWER(text), CASEFOLD(text)	Full Unicode 17 default mapping; may expand characters	None
UCASE(text), LCASE(text)	Aliases for UPPER and LOWER	None
TRIM(text), LTRIM(text), RTRIM(text)	Remove Unicode White_Space from both/start/end	None
CONTAINS(text, needle)	BOOLEAN substring test	None
STARTSWITH(text, prefix), ENDSWITH(text, suffix)	BOOLEAN boundary test	None
REMOVEPREFIX(text, prefix), REMOVESUFFIX(text, suffix)	Remove one matching boundary; otherwise return independent unchanged value	None
REPLACE(text, old, replacement)	Replace all nonoverlapping matches; empty old inserts at scalar boundaries, including both ends	None
SPLIT(text, separator=““)	LIST OF STRING; empty separator splits Unicode code points, nonempty separators preserve empty fields	StringError (legacy contract)
LINES(text)	LIST OF STRING, splitting line boundaries without retaining separators	None
JOIN(parts, separator)	Join a LIST OF STRING	None
PADLEFT(text, width), PADRIGHT(text, width)	Space padding to width; shorter/negative widths leave text unchanged	StringError

Function	Result and behavior	Declared error
PADLEFTWITH(text, width, fill), PADRIGHTWITH(text, width, fill)	Padding with exactly one scalar	StringError
FORMAT(template, values)	Replace {} with successive strings from LIST OF STRING; {{ and }} escape braces	StringError
CODEPOINTS(text)	LIST OF INTEGER containing Unicode scalar values	None
UNICODEVERSION()	“17.0.0”	None

MID/INSTRFROM reject negative starts and starts beyond LEN. LEFT/RIGHT/MID reject negative counts; excessive nonnegative counts clamp to available text. SPACE/REPEAT reject negative counts. Padding widths and slicing positions are not display-column measurements.

VAL requires the entire trimmed input to be optional ASCII + or - followed by ASCII decimal digits. Empty input, embedded whitespace, fractional/exponent/radix notation, and trailing characters fail. The full signed 64-bit range is accepted. VAL parses INTEGER; STR also formats DOUBLE. Use checked CDBL(text) / VALDOUBLE(text) for floating-point parsing; see the math reference.

LINES recognizes CRLF as one boundary, plus LF, CR, vertical tab, form feed, U+001C–U+001E, NEL, line separator and paragraph separator. Empty input yields an empty list; a trailing line boundary does not add a final empty element. SPLIT with a nonempty separator preserves final empty fields. Its separator is optional and defaults to ““: omitted/empty separators split into Unicode code points, with empty input producing an empty list. Combining marks remain separate entries.

FORMAT requires exactly as many values as {} placeholders, and balanced/escaped braces. Values are strings: use STR explicitly for numeric values. Width/precision/type format specifications are deferred.

Traditional string-result spellings with a trailing dollar are optional **call aliases**, such as LEFT $(text, count), UCASE$(text) and STR$(value). They are recognized for the string-result functions listed above except UNICODEVERSION; they do not introduce typed variable suffixes or arbitrary dollar-named functions. INSTR/ASC/VAL/SPLIT/LINES return other types and have no dollar alias. This is modern BASIC behavior, not complete QuickBASIC compatibility: positions are zero-based, not-found is −1, CHR/ASC use Unicode scalars, and VAL is checked and strict.

Recoverable errors

NumericError and StringError are distinct built-in typed records with an INTEGER Code field. Construction is an ordinary value operation. User functions declare escaping types in FAILS, and RETURN ERROR explicitly forwards a caught error.

Error	Code	Meaning
NumericError	1	Invalid decimal integer syntax
NumericError	2	Integer outside the signed 64-bit range
StringError	1	Invalid Unicode scalar (negative, surrogate, or above U+10FFFF)
StringError	2	Invalid position or negative count
StringError	3	Reserved legacy split error code; empty separators now succeed
StringError	4	ASC called on empty text
StringError	5	Invalid template or mismatched format argument count
StringError	6	Padding fill is not exactly one scalar
StringError	7	Invalid byte sequence supplied to DecodeUtf8/ReadFileText

```
TRY
    PRINT UPPER("Straße")
    PRINT TRY LEFT("café 🌍", 4)
```

```
    PRINT JOIN(TRY SPLIT("one,two,three", ","), " | ")
    PRINT TRY VAL("42")
CATCH err AS StringError
    PRINT "String error code: " + STR(err.Code)
CATCH err AS NumericError
    PRINT "Number error code: " + STR(err.Code)
END TRY
```

Errors inside a catch target an outer matching catch or propagate from the function, never another clause of the same handler. Partial expressions and owned locals are cleaned on recoverable exit. Allocation failure remains fatal, separate from these errors, with no normal application-cleanup guarantee. Integer arithmetic overflow and invalid ByteAt access are also fatal.

Format strings

Implemented by the self-hosted compiler. An `f` or `F` immediately before a string quote enables interpolation:

```
DIM value=12.3456
DIM count=42
PRINT f"The value is: {value:.2f}"
PRINT f"Count: {count:08d}; hex: {count:#08x}"
PRINT f"{{literal braces}} {count+1}"
```

This prints `12.35`, `00000042`, `0x00002a`, literal braces and `43` in the corresponding positions. Ordinary strings retain their existing behavior. Double a quote inside literal text using BASIC's `""` convention. Expressions inside fields can contain quoted strings, calls, qualified module names, indexing, record construction, nested f-strings and checked `TRY` calls.

Each field evaluates exactly once, from left to right. Formatting creates an owned STRING. Existing borrow checks and independent-value semantics apply; formatting cannot retain an element borrow or repair a conflicting mutation. If a field fails through a checked call, earlier owned interpolation results are destroyed before reaching the enclosing handler or propagating.

Specifications

A colon separates an expression from a static specification. The order follows Python's common mini-language:

```
[[fill]align][sign][#][0][width][grouping][.precision][presentation]
```

Part	Behavior
`<`, `>`, `^`	Left, right, center alignment; strings default left, numbers right
`=`	Numeric padding after the sign and base prefix
Fill	One Unicode scalar immediately before alignment, such as 🌟`^10`
`+`, `-`, space	Numeric sign policy; minus-only is the default
`#`	Integer base prefix or retained decimal point/trailing precision zeros
`0`	Numeric zero padding, preserving sign/base prefix
Width	Minimum output width in Unicode scalars, not bytes or graphemes
`,`, `_`	Numeric grouping; commas for decimal, underscores also for other integer bases
`.precision`	STRING scalar limit; DOUBLE fractional digits for `f`, digits after the point for `e`, significant digits for `g`
`s`	STRING
`d`, `b`, `o`, `x`, `X`	INTEGER/unsigned decimal, binary, octal, lower/upper hexadecimal
`f`, `F`, `e`, `E`, `g`, `G`, `%`	Fixed, scientific, general, percentage numeric formatting

INTEGER and unsigned values may also use floating presentations through the ordinary binary64 conversion. BOOLEAN with an empty specification produces `True`/`False`; a nonempty valid numeric specification formats 1/0. With no numeric presentation or precision, DOUBLE uses the language's existing shortest-round-trip `STR` spelling, including its exponent thresholds and `.0` convention. This spelling is not promised identical to Python's `str` in every case.

Decimal precision rounding uses the exact binary64 value and nearest rounding with ties to even; it does not multiply a binary approximation by powers of ten before rounding. Default explicit floating precision is six; `g` precision zero means one significant digit. With a precision but no presentation, general formatting preserves the customary `.0` in fixed notation and uses the corresponding Python-style scientific threshold. Nonfinite values use `inf`/`nan`, uppercase for uppercase presentations. Percentage formatting scales in binary64 before decimal formatting.

Invalid syntax, incompatible value/specification types, unsupported values, and width/precision above 1,000,000 are compile-time errors, including in unused functions. Nested quoted interpolation is scanned with an explicit depth limit of 64. There is no unchecked runtime formatting failure channel. Allocation failure remains fatal, as for other STRING construction.

Specifications are static. Nested dynamic width/precision fields, `!s`/`!r`/`!a` conversions, debug `=`, locale-sensitive `n`, and character `c` are not implemented. Nominal OPERATOR Display supplies text before STRING modifiers are applied. Multiline formatted blocks use `f"""` immediately followed by a newline and support the same checked specifications. A malformed or unsupported specification is rejected instead of silently ignored. Existing checked positional `FORMAT(template,values)` remains available with its original contract.

Multiline strings

Triple double quotes begin a block **immediately followed by a newline**. The closing delimiter may follow content. Ordinary doubled-quote literals keep their meaning; `"""word"""` is still the ordinary literal containing `"word"`.

```
name="Fred"
message=DEDENT TRIM f"""
    Hello {name}.
        Count: {42:06d}
"""
PRINT message
```

This prints `Hello Fred.` followed by an indented `Count: 000042`. Modifiers belong to the expression and also work in arguments, defaults and FORMAT options.

Form	Behavior
`"""` newline … `"""`	Preserve content, quotes, braces, indentation and original LF/CRLF/CR bytes
`DEDENT """`	Remove the common exact space/tab prefix of nonblank content lines
`TRIM """`	Remove boundary newline sequences only
`DEDENT TRIM """` or `TRIM DEDENT """`	Dedent, then trim; duplicate modifiers are errors
`f"""`	Apply existing checked interpolation/format specifications after layout processing

Opening/closing indentation is content. A closing line's spaces survive a raw or TRIM-only literal; DEDENT can remove their matching common prefix. Tabs are exact characters, independent of editor tab width. Whitespace-only lines do not select the common prefix; they lose their matching prefix and retain extra spaces/tabs. If all lines are blank there is no common margin to remove.

TRIM never removes horizontal whitespace or interior newlines and never leaves half a CRLF. Ordinary `TRIM(text)` still removes surrounding whitespace. No backslash escapes are introduced. Six consecutive quotes inside a block represent three literal quotes. In a quote run, each six-quote group is escaped; a remaining three to five quotes closes the block, retaining zero to two preceding quotes. One or two remaining quotes are ordinary content.

Plain blocks treat braces literally. Formatted blocks use `{expression:spec}` and `{{/}}` escapes, as ordinary f-strings do. Each interpolation expression remains single-line; its value may contain newlines/indentation. Layout processing never changes characters supplied by an interpolation. Interpolated expressions retain ordinary checked effects and constant-expression restrictions; interpolation does not turn a runtime call into a constant default.

Collection library reference

Implemented by the primary self-hosted BASIC compiler. Algorithms are first-party BASIC in stdlib/lists.bas, stdlib/hash_collections.bas and stdlib/arrays.bas, with additional generic families for deques, queues, heaps, sorted keys, counters, multimaps, bitsets and caches. Collections accept all implemented copyable types: INTEGER, BOOLEAN, DOUBLE, STRING, arrays, nominal records, supported error values, optionals, and nested lists/sets/dictionaries. Types remain static and homogeneous; there is no dynamically typed catch-all value. Arrays, lists, optionals and dictionary values can also own files/sockets/user resource records under explicit ownership rules; these containers are noncopyable. Resource dictionary Put/Remove/Extract/iteration and checked borrowed Get/SetDefault/Find and bracket reads/writes are supported; resource keys/sets and task/channel capability elements remain unsupported.

Collections grow automatically. Assignment makes independent values; owned elements are copied recursively. Ordinary parameters borrow read-only access, and BYREF borrows exclusive writable access. An ordinary owned return copies a borrowed collection; explicit BORROW returns a checked read-only view tied to one or two declared parameters. Fresh temporary owned results can construct their destination directly. No resource transfer is inferred from last use, and no garbage collector/reference counting is used.

Resource lists support literals, Append, Insert, Reverse, SwapAt, Pop, ExtractAt, RemoveAt, Clear, checked element/field replacement and borrowing. Copying operations, slices and generic operations that require copying an element remain rejected. Pop removes and transfers the last owner; ordinary indexed reads cannot make an independent resource copy.

ToArray takes a BYVAL list and creates a zero-based rank-one array. Copyable inputs remain independent; named resource lists need explicit TAKE. See resource array defaults, borrows and resizing.

Lists

```
DIM numbers = [3, 1, 2, 1]
DIM names AS LIST OF STRING = []
Append(BYREF names, "Ada")
PRINT IndexOf(numbers, 1)
PRINT CountOf(numbers, 1)
DIM ordered = Sorted(numbers)

FOR EACH number IN ordered
    PRINT number
NEXT
```

LIST() creates an empty integer list without type context, or the list type supplied by its destination. Nonempty literals infer one element type; [] requires a LIST type context. Literals evaluate elements left to right, accept line breaks/trailing commas, and clean completed elements/list storage if a later expression fails. Mixed element types are rejected.

Operation	Result / behavior
LEN(items), items[index], items[index] = value	Length, read and replacement; zero-based checked indexing
Append(BYREF items, value)	Append one independent value
Extend(BYREF items, values)	Append in source order; returns new length
Concat(left, right)	Independent combined list
Contains(items, value)	BOOLEAN membership
IndexOf / LastIndexOf(items, value)	First/last index, or −1
CountOf(items, value)	Number of matches

Operation	Result / behavior
Insert(BYREF items, index, value)	Insert before index, including LEN; returns new length; FAILS IndexError
RemoveAt(BYREF items, index)	Remove indexed element; returns 0; FAILS IndexError
Remove(BYREF items, value)	Remove first equal element; returns BOOLEAN indicating removal
ExtractAt(BYREF items,index)	Return/remove the indexed owner; FAILS IndexError; copyable and resource elements
Pop(BYREF items)	Return/remove last element; FAILS EmptyCollection
Slice(items, start, finish)	Independent half-open [start, finish) slice; FAILS IndexError
Reverse(BYREF items)	Reverse in place; returns 0
Sort(BYREF items) / Sorted(items)	Stable ascending sort / independent sorted result
Clear(BYREF items)	Remove all values, retaining capacity

Sorting compares integers numerically and strings in exact Unicode scalar lexicographic order, independent of locale. Search uses exact equality. Insert/RemoveAt/ExtractAt/Slice validate positions before mutation. RemoveAt destroys the removed owner after compacting; ExtractAt transfers it to the result, with no copies of shifted elements. IndexError has an INTEGER Index field. EmptyCollection is a fieldless typed error. Direct indexing retains its existing fatal bounds policy; the listed checked editing/slicing APIs use recoverable failures. Bracket indexing now accepts negative indices, and bracket slices support strides; see sequence reference for implemented behavior and remaining compatibility limits.

Extend(BYREF items, items) conflicts with exclusive access and is rejected; explicitly copy the source if a snapshot is desired. Checked Reserve/Resize/Truncate/CopyWithin are documented in buffers. Named-function predicates and stable comparator callbacks are documented below. Generic records provide statically specialized user/container types. SPLIT, LINES, CODEPOINTS and JOIN are documented in the string library.

Sets

```
DIM names AS SET OF STRING = SET()
Insert(BYREF names, "Ada")
Insert(BYREF names, "Grace")
IF Contains(names, "Ada") THEN
    PRINT "Found"
END IF
```

SET() requires SET type context. Members are unique. LEN, Insert, Contains, Remove, Clear, Keys, Union, Intersection, Difference, SymmetricDifference, assignment/returns and FOR EACH are supported.

Insert returns TRUE for a new member and FALSE for a duplicate. Duplicates keep their position. Remove returns whether a member was removed; removal followed by insertion puts the member last. Keys returns an independent list snapshot in insertion order. Union visits left then unseen right members; intersection/difference preserve left order; symmetric difference visits left-only then right-only members. Whole-set = / == / <> use order-independent value equality.

Dictionaries

```
DIM ages AS DICT OF STRING TO INTEGER = DICT()
Put(BYREF ages, "Ada", 36)
Put(BYREF ages, "Grace", 85)

FOR EACH name, age IN ages
```

```
    PRINT name + ": " + STR(age)
NEXT
```

DICT() requires DICT type context. Supported operations:

Operation	Result / behavior
LEN(table)	Number of entries
Put(BYREF table, key, value)	Insert or replace; TRUE for a new key, FALSE for replacement
Contains(table, key)	Key membership, without insertion
Find(table, key)	Independent copyable optional or checked resource optional view
Get(table, key)	Independent copyable value or checked resource view; typed MissingKey
Get(table, key, fallback)	Stored value or eagerly evaluated fallback; no insertion. Copyable results are independent; resource results borrow from both possible owners
ContainsValue(table, value)	Explicit linear value search
Remove(BYREF table, key)	BOOLEAN indicating removal; releases owned key/value
Extract(BYREF table, key)	Optional independent value, removing the entry if present
Keys(table), Values(table)	Independent insertion-order list snapshots
Clear(BYREF table)	Release entries, retain table storage for reuse

Replacement preserves key position. Remove/reinsert moves the key last. Find distinguishes a missing entry from a present zero or empty string. Optional dictionary values and structural Equal are supported. A list-of-records Entries snapshot remains future work. The supported pair iteration does not depend on an Entries snapshot. See defaults and optionals for checked presence binding, fallback evaluation and error contracts.

Predicate operations

Named functions, callable values and contextual lambdas can supply predicates. A list or set predicate has signature `CALLABLE(T) AS BOOLEAN`; dictionary predicates use `CALLABLE(K,V) AS BOOLEAN`. Parameters borrow read-only. Earlier collection arguments select a matching overloaded function. Predicates must be infallible; they can suspend, with ordinary cancellation and cleanup. Lambda captures are independent readonly snapshots of copyable values; resource captures and arbitrary composed predicate error sets remain unsupported. For example, `Filter(items,(value) => value>=minimum)` captures minimum at construction.

Operation	Collections	Result / evaluation
Any(items, Predicate)	List, set, dict	TRUE at first matching element; FALSE when empty
All(items, Predicate)	List, set, dict	FALSE at first nonmatch; TRUE when empty
CountWhere(items, Predicate)	List, set, dict	Count all matching elements; zero when empty
Filter(items, Predicate)	Copyable list, set, dict	Independent matching elements/entries in source order
IndexWhere / LastIndexWhere(items, Predicate)	List	First/last matching index, or −1; search forward/backward respectively
FindWhere(items, Predicate)	Copyable list	Independent first match wrapped in SOME, or NONE

Each visited element is tested once. Sets/dicts visit insertion order. Filter does not mutate its source, retains dictionary keys, and preserves list duplicates. Read-only searches/counts also accept owning resource lists/dictionary values; Filter and FindWhere reject resource results because they would require copying. The source remains protected against invalidating writes/

transfers during callback execution, including suspension. Cancellation cleans a partially built result and callback/iteration temporaries. Ordinary fatal-allocation policy still applies.

```
values=[1,2,3,4]
PRINT Any(values,Even)
PRINT CountWhere(values,Even)
FOR EACH value IN Filter(values,Even)
    PRINT value
NEXT
FUNCTION Even(value AS INTEGER) AS BOOLEAN = value MOD 2=0
```

Iteration and borrowing

FOR EACH item IN list/set introduces a read-only binding. FOR EACH key, value IN dictionary introduces two; a dictionary may also be traversed by key alone. NEXT may optionally repeat the first binding name. Bindings are scoped to the loop and cannot be reassigned. Rebinding an existing active name is rejected.

Iteration borrows the source for the loop's duration. Mutation/replacement through the source name, another field of its owning record, or a BYREF call is rejected while this borrow is live. This conservative record-wide exclusion also rejects some disjoint-field writes. String bindings borrow their protected element storage without payload copies. Retaining or returning them creates independent values. Temporary collection results can be iterated; their ownership ends on normal loop completion, function exit or recoverable failure. No hidden snapshot repairs a conflicting mutation.

Generic algorithms and value capabilities

The BASIC library uses generic functions, for example:

```
FUNCTION First[T](items AS LIST OF T) AS T
    RETURN items[0]
END FUNCTION

DIM readings = [1.5, 2.5]
PRINT First(readings)
DIM names AS DICT OF STRING TO LIST OF STRING = DICT()
Put(BYREF names, "team", ["Ada", "Grace"])
DIM missing = Get(names, "absent", [])
```

Earlier arguments supply type context for later `[]` and NONE arguments. Generic specializations are cached by template and concrete type arguments. Ordinary recursion reuses a specialization; a recursive call that continually changes type arguments is rejected. Generic bodies are checked when instantiated. No runtime boxing, shared collection storage, or implicit ownership transfer is introduced. Generic record declarations and explicit bracketed type-argument call syntax remain unimplemented. Result AS annotations and known complete-call destinations can supply generic result constraints; see result context.

`Equal(left,right)` and same-type = / == compare copyable values structurally. IEEE equality applies recursively: NaN is unequal even within a value compared with itself, and signed zeros are equal. Records/choices/optionals and sequence contents participate; set/dictionary equality ignores insertion order, capacity and tombstones. Resource and callable equality reject. `Hash(value)` remains a deterministic nonnegative 31-bit INTEGER, with equal signed-zero hashes; it is not cryptographic.

`Compare(left,right)` returns CompareResult.Less/Equal/Greater/Unordered. NaN is unordered; composite comparison is lexicographic. Default sorting uses a separate total comparison with NaNs last, so a sorting tie does not establish equality. Exact comparison and constructor contracts.

NaN members/keys raise InvalidKeyError before lookup or mutation, including empty collections and composite keys. Dictionary values and list elements may hold NaN. Infinity is valid; signed

zeros identify one key. ValidateKey returns zero or the same typed failure. NaN-free types keep infallible specializations without runtime scans; other public key operations compose the error with FAILS LIKE. Get without a fallback additionally reports MissingKey. Existing-key algebra and independent key snapshots preserve validity. A transferred value argument is cleaned if a later key check fails; ownership transfer is not rolled back.

List editing/search/slicing/sorting operations support the implemented copyable element types. Resource containers expose only their documented ownership-safe subset; operations requiring resource copies remain rejected. Sets/dictionaries preserve insertion order for snapshots and iteration. Removing a key releases its owned key/value storage; copying a table with removed slots is safe. Recoverable failures clean up argument temporaries and partially built values. OOM remains fatal. Owned declarations inside conditional/TRY/loop bodies are supported with lexical visibility and per-iteration cleanup. FOR EACH introduces checked read-only borrowed bindings and protects its source root for the loop lifetime.

Runnable examples: nested values and general keys. Validation includes nested ownership, NaN-key rejection, signed-zero key equivalence, arrays and errors as keys, generic recursion, dictionary order-independent equality, malformed/unsafe cases, and repeated successful/failing operations under a 64 MiB address-space limit.

Comparator sorting

`Sorted(items, Compare=comparator)` returns an independent sorted list; `Sort(BYREF items, Compare=comparator)` changes the list in place and returns 0. Both are stable: a comparator result of zero preserves original relative order. The comparator takes two read-only elements and returns INTEGER (negative/zero/positive compatibility) or CompareResult. The callback itself must be infallible. An enum Unordered result produces UnorderedComparisonError; callback sorts build before commit, leaving the input intact on that failure. Stable ties preserve original order. Named/positional and overloaded named comparators are checked with their element and inferred result types. Ordinary named functions and lambdas with checked copyable captures are supported; generic named-function references remain unsupported. The default overload avoids callback dispatch and uses NaNs-last total value ordering.

```
names=["bob","Ada","BOB"]
ordered=Sorted(names,Compare=CompareCaseInsensitive)
Sort(BYREF names,Compare=CompareCaseInsensitive)
FUNCTION Descending(a AS INTEGER,b AS INTEGER) AS CompareResult
    RETURN Compare(b,a)
END FUNCTION
```

`CompareCaseInsensitive` returns CompareResult after comparing full Unicode CASEFOLD results. It does not normalize canonically equivalent text or apply language-specific collation. STRING overloads also exist: `Sorted(text, Compare=comparator)` sorts Unicode code points and returns STRING; strings use the value-returning `Sorted` form. The default STRING overload uses ordinary value ordering. Combining marks are separate elements, so character sorting is not locale-aware text collation. Empty inputs return empty values. Original strings and other independent copies remain unchanged. Comparator callbacks cannot mutate the collection being sorted through another alias; ordinary checked borrowing still applies. Captures are independent readonly snapshots, not aliases granting mutation of the original collection.

Compiler-shipped collection families

These APIs ship in the compiler's embedded first-party BASIC library. No import, generated Integer/String variant, or application `.blib` dependency is required. The algorithms use ordinary specialized records, checked borrows and storage exchange; they are not separate opaque runtime implementations. See core lists/sets/dictionaries and generic records.

Construction and ownership

```
DIM jobs AS Deque[STRING]=NewDeque()
PushBack(BYREF jobs,"build")
PushFront(BYREF jobs,"check")
PRINT PopFront(BYREF jobs)

cache=NewCache(100) AS Cache[STRING,LIST OF INTEGER]
CachePut(BYREF cache,"numbers",[1,2,3])
```

Empty factories need their type arguments from a typed destination or `AS` result annotation. Later mutations do not infer an earlier declaration's type. `Length` is the logical element count for these records; `LEN` remains the core sequence/ hash-container operation. `Reset(BYREF collection)` empties a collection; it does not change its configuration. Ordinary parameters borrow, `BYREF` excludes other access, and copying copyable containers/snapshots creates independent values.

Insertion takes a `BYVAL` payload: it copies a borrowed copyable value once, adopts a fresh value, or adopts an explicitly transferred named owner. There is no last-use resource transfer. Resource payloads make their containing collection noncopyable. Pop, extraction and bounded-deque eviction return owned values; resource Front/Back/Peek/Lookup/Find/Get return checked read-only views. A view excludes mutation, transfer and relocation of its whole owner until consumption finishes, including suspension. Store a snapshot only for copyable payloads.

```
DIM files AS Deque[File]=NewDeque()
PushBack(BYREF files,OpenFile("/dev/null"))
PRINT GetFileInfo(Front(files)).Size
file=PopFront(BYREF files)   ' transfer out; normal scope cleanup closes it
```

Keys must be copyable and support the existing canonical equality/hash/order rules. Structural NaN keys are rejected with `InvalidKeyError`; signed zero is one key. Sorted containers use `Compare` and existing NaN-last ordering, after key validation. Custom Equal/Hash/Compare must agree. Resource keys and stored task/channel capabilities remain unsupported.

Opaque container fields are private implementation state. Automatic JSON/shared serialization cannot expose or manufacture that state: it raises JsonError code4. Encode a logical snapshot, or configure a concrete formatter that decodes through the public factories/operations. This also applies to other opaque types without declared formatters. Generic `OPERATOR` declarations remain unsupported; do not treat derived representation equality/hash as abstract sequence/cache equality. For logical deque/queue comparison, compare their copyable snapshots explicitly.

Sequences and priority queues

Type / factory	Operations	Cost and behavior
`Deque[T]`, `NewDeque()`	Length, PushFront/PushBack, Front/ Back, PopFront/PopBack, Snapshot, Reset	Circular buffer; amortized O(1) insertion/ removal at either end; O(n) growth/ snapshot/reset
`Queue[T]`, `NewQueue()`	Length, Enqueue, Front, Dequeue, Snapshot, Reset	FIFO over a deque

Type / factory	Operations	Cost and behavior
`BoundedDeque[T]`, `NewBoundedDeque(capacity)`	Deque operations plus Capacity	Positive capacity; inserting at a full end returns the owned OPTIONAL eviction from the opposite end; otherwise NONE
`PriorityQueue[T]`, `NewPriorityQueue()`	Length, Push(BYREF q,priority,value), Peek, PeekPriority, PopPriority, Reset	Binary min-heap, signed INTEGER priority; O(log n) push/pop, O(1) peek; equal-priority order unspecified

Deque/queue/heap empty reads and removals raise `EmptyCollection`. Bounded-deque construction rejects nonpositive capacity with `CollectionError`; its pushes also declare CollectionError/EmptyCollection. Snapshots preserve logical sequence order and require copyable payloads. Priorities include INT64_MIN/MAX without negation. Heap payloads need no comparison operation. Growth and sifting exchange complete values rather than cloning their managed fields or resources.

Ordered keys and counting

Existing SET/DICT preserve insertion order; `NewOrderedSet()` and `NewOrderedDictionary()` are contextual factories for those same native types.

Type / factory	Operations
`SortedSet[T]`, `NewSortedSet()`	Length, Has, InsertValue, RemoveValue, At(index), LowerBoundIndex(value), Range(first,finish), Snapshot, Reset; SetUnion/SetIntersection/SetDifference/SetSymmetricDifference
`SortedDictionary[K,V]`, `NewSortedDictionary()`	Length, Has, SetValue(key,value), Lookup(key), LookupRequired(key), RemoveValue(key), Extract(key), KeySnapshot, ValueSnapshot, RangeKeys(first,finish), Reset; resource Values view
`Counter[T]`, `NewCounter()`	Length, Count(key), TotalCount, Add(key,amount=1), RemoveValue(key,amount=1), KeySnapshot, Reset
`MultiMap[K,V]`, `NewMultiMap()`	Length, TotalCount, Add(key,value), Has, Find, Get, GetAll, RemoveOne(key,value), RemoveKey(key), Extract(key), ExtractAt(key,index), KeySnapshot, Reset

Sorted keys use binary search, O(log n) lookup and O(n) insertion/removal in a sorted vector. Ranges are half-open `[first,finish)`; reversed ranges are empty. Sorted-set algebra merges in O(n+m). InsertValue/SetValue return TRUE for a new key, FALSE for an existing one. Lookup returns OPTIONAL; LookupRequired raises MissingKey. At and editing APIs use nonnegative checked indices and IndexError. Key/order errors remain in specialized FAILS contracts.

Counter counts are positive; missing keys read as zero and zero counts remove the key. Add/RemoveValue require nonnegative amounts. Removal returns the amount removed, bounded by the current count. Length counts distinct keys; TotalCount counts occurrences. Overflow/invalid amounts raise CollectionError before mutation.

Multimaps retain duplicate values and per-key insertion order. Length counts keys, TotalCount counts all values; removing the last value removes the key. Get raises MissingKey; GetAll returns an independent list or an empty list and requires copyable values. Find/Get can borrow resource buckets. RemoveOne uses equality and requires copyable values; RemoveKey destroys all values and returns their count. Extract returns an owned optional bucket; ExtractAt removes one owned value. Key snapshots retain native insertion order; sorted-dictionary snapshots are sorted. ValueSnapshot is copyable-only; Values on a resource sorted dictionary is a view.

Bitsets

`NewBitSet(size)` creates a packed UINT64 fixed universe `[0,size)`. Capacity is the universe size; Length counts included bits. Has/Include/Exclude/Flip use checked nonnegative indices and IndexError. Members returns ascending INTEGER indices. Reset clears bits without changing the universe. BitUnion, BitIntersection, BitDifference, BitSymmetricDifference and Complement return independent bitsets. Binary operations require equal universes (CollectionError). Complement

masks unused padding bits. There is no arbitrary 100-million-bit cap; negative sizes are checked, while ordinary allocation/representability limits retain the fatal allocation policy. Storage is `ceil(size/64)*8` payload bytes, plus ordinary list/record headers.

Bounded caches

```
DIM cache AS Cache[STRING,STRING]=NewCache(100,CachePolicy.LRUWithTTL,SOME(Seconds(60)))
result=CachePut(BYREF cache,"key","value",weight=2)
IF LET value=CacheFind(BYREF cache,"key") THEN
    PRINT value
END IF
```

Operation	Behavior
NewCache(maxEntries,policy=LRU,expireAfter=NONE,maxWeight=0)	Finite entry bound; weight0 disables the weight bound; TTL requires a positive Duration, plain LRU forbids one
NewTimedCache(maxEntries,policy,expireAfter,now,maxWeight=0)	TTL/LRUWithTTL with caller-supplied MonotonicInstant samples
CachePut(BYREF cache,key,value,weight=1)	Insert/replace, prune expired entries, then evict to satisfy both bounds; CachePutResult has Inserted, Evicted, Expired
CacheFind(BYREF cache,key)	Prune, return OPTIONAL; refresh recency for LRU/LRUWithTTL
CachePeek(cache,key)	Read-only OPTIONAL; neither prune nor refresh recency
CacheExtract(BYREF cache,key)	Remove/return an owned OPTIONAL
CacheRemove(BYREF cache,key)	Destroy/remove, return presence
PruneExpired(BYREF cache)	Destroy expired entries; return removed count
CacheCount(BYREF cache)	Prune, then return live count
ResidentCount / ResidentWeight	Stored accounting including entries not yet pruned; no clock read
CacheCapacity / CacheWeightLimit	Configuration, no clock read
Reset(BYREF cache)	Release entries; preserve bounds, policy and accepted time domain

Every timed operation has an `At` counterpart with `now AS MonotonicInstant` after key/value and before optional weight. An externally timed cache requires At operations; ordinary caches reject them. LRU reads no clock. Other ordinary caches sample the system monotonic clock. Caller-owned Clock values can supply samples for active/boot/test time without transferring or copying clock authority.

TTL expires at `age >= expireAfter`, measured since the most recent write. Find does not extend TTL. TTL evicts oldest writes; LRU policies evict least recently used entries. Expiry order and recency have separate linked indexes; lookups/edits are amortized O(1), plus entries pruned/evicted. Freed slots are reused. Weights are caller accounting units, not measured memory usage.

Accepted mutating samples must be nondecreasing and from the same clock domain. Wrong domains, backwards samples, invalid mode/weights and weight-total overflow produce checked errors. A Peek validates time but does not advance LastObserved. Expired entries can be pruned before a later rejected write; this is not a full rollback transaction. APIs declare CollectionError/DateTimeError/IoError and key-dependent errors; plain LRU construction itself only declares CollectionError.

These are ordinary exclusive mutable values: no implicit locks, worker thread, timer task, asynchronous loader or request coalescing. Cancellation follows structured scope exit and cleans retained/temporary owners; a cancelled edit is not a recoverable transaction that resumes the same invocation.

Sequence operations and brackets

Implemented in the maintained BASIC compiler. Values retain BASIC's independent ownership semantics. This is not yet complete Python subscript compatibility; remaining boundaries are listed below.

Naming convention

Operation	LIST	STRING
Sort in place	`Sort(BYREF items[, Compare=comparator])`	Use `Sorted`
Independent sorted result	`Sorted(items[, Compare=comparator])`	`Sorted(text[, Compare=comparator])`
Reverse in place	`Reverse(BYREF items)`	Use `Reversed`
Independent reversed result	`Reversed(items)`	`Reversed(text)`

Mutators return the existing INTEGER status 0. Value-returning forms leave their input unchanged. String operations use Unicode code points, preserving UTF-8; combining marks are separate elements. This convention is recorded in instructions.md.

Reads and slices

```
numbers=[0,1,2,3,4]
PRINT numbers[-1]                ' 4
middle=numbers[1:4]              ' [1,2,3]
backwards=numbers[::-1]          ' [4,3,2,1,0]
everyOther=numbers[::2]          ' [0,2,4]
PRINT "aé🌍"[-1]                  ' 🌍
PRINT "aé🌍"[::-1]                ' 🌍éa
```

`[start:stop:step]` uses Python's half-open bounds, sign-dependent defaults, negative indices, clipping and negative steps. A missing negative-step stop is different from an explicit `-1`. Results are independent owned values, including owned records/collections inside sliced lists. Indices/bounds/steps must be INTEGER; existing 64-bit integer limits apply. Step zero raises `SliceError` with Code 1. The implementation avoids overflow for minimum/maximum 64-bit bounds and steps. Reads can chain through fields and other subscripts; temporaries are kept alive until their dependent values are owned.

Subscript computation cannot leave the parent dangling. Named collection handles are refreshed after effectful index/bound evaluation: supported in-place growth and resize remain visible, atomic replacement selects the replacement, and an unrestored move of the parent is rejected. Already borrowed dynamic views protect their roots from invalidation; unnamed copyable parents use independent snapshots when needed. Failure/cancellation releases held owned parents and partial values.

Strings support reads and slices, not element/slice assignment. An invalid STRING index raises checked `IndexError`; mutating STRING elements is rejected. Existing dense ARRAY bounds and coordinate syntax are separate from LIST slices.

List slice assignment

```
numbers=[0,1,2,3,4]
numbers[1:4]=[8,9]               ' [0,8,9,4]
numbers[::2]=[7,6]               ' [7,8,6,4]
numbers[2:2]=[5]                 ' insert without removing
```

A unit-step slice can grow or shrink. An extended slice requires exactly the number of replacement elements selected; a mismatch raises `SliceError` Code 2 before mutation. RHS evaluation precedes slice-bound evaluation. Replacement values are independent snapshots, including self-assignment and overlapping slices. Nested record, list, dictionary and array destinations support the same syntax. Recoverable validation failure leaves the destination unchanged; fatal allocation failure retains the language's separate fatal policy.

Dictionaries

```
DIM visits AS DICT OF STRING TO INTEGER=DICT()
visits["Ada"]=1
PRINT visits["Ada"]
PRINT SetDefault(BYREF visits,"Ada",99)   ' 1; existing value retained
PRINT SetDefault(BYREF visits,"Bob",0)    ' 0; missing key inserted
```

Dictionary brackets use the exact declared key/value types. Missing reads raise checked `MissingKey`, and assignments insert or replace while retaining insertion order. Copyable reads return independent values. Resource reads borrow checked views; resource writes adopt fresh values or explicit TAKE and release any previous owner. Resource Find returns an optional view; use Extract to remove ownership. Resource views cannot outlive/invalidate their dictionary origin. `SetDefault(table,key,fallback)` eagerly evaluates fallback like any ordinary argument, inserts it only when absent, and returns the selected value independently. Zero, FALSE, empty strings and present NONE are all existing values. With OPTIONAL dictionary values, `SetDefault(BYREF table,key)` inserts NONE when absent. A statically nonoptional value type needs an explicit fallback; no implicit NULL is introduced.

List/set conversions

`SET(list)` / `ToSet(list)` remove duplicates using existing hash/equality rules. `LIST(set)` / `ToList(set)` return elements in the set's deterministic insertion order. Both return independent values. Same-kind conversions also copy independently. `LIST(text)` splits into Unicode code points, and `SET(text)` keeps unique code points in first-occurrence order. Unannotated empty constructors still need enough type context to determine their element type.

Nested assignment and compound operators

See assignment rules for nested destinations, RHS-first ordinary assignment, single-evaluation compound updates and conservative borrow exclusions. A slice must be the final destination component. STRING mutation remains rejected.

Remaining compatibility blockers

- Direct LIST out-of-range access/element assignment retains the existing fatal bounds policy. Converting this to checked `IndexError` needs a coordinated effect/FAILS migration, including the compiler itself and native exports.
- Python's dynamic typing, arbitrary-precision integers and shared mutable object aliases are not BASIC language semantics. No such behavior is silently added.

Initialized buffers, bulk I/O and allocation

Plain fallible calls are automatically checked; inline TRY in the API tables is accepted compatibility syntax, not a requirement.
Public ByteBuffer/StringBuilder and stream APIs build on these initialized storage contracts. Compiler boundary/layouts.

List capacity and editing

The self-hosted compiler provides these generic functions for every implemented copyable list element type:

Operation	Result and checked failures
`TRY Reserve(BYREF items,capacity)`	Ensures capacity, returns 0; `IndexError` for negative/ unrepresentable capacity
`TRY Resize(BYREF items,length,fill)`	Returns new length; shrinks with destruction or grows with independent copies of `fill`
`TRY Truncate(BYREF items,length)`	Shrink only, returns length; destroys removed values
`TRY CopyWithin(BYREF items,source,destination,count)`	Returns count; checked initialized ranges; overlap has snapshot semantics

Reserve never changes LEN and never exposes spare storage for reading or writing. Resize requires an explicit initialized fill value, including for records and nested collections. Invalid ranges are checked before mutation. Allocation failure remains fatal. CopyWithin has snapshot semantics, so overlapping owned values remain independent. The packed BYTE overload uses bounded memmove without an intermediate allocation. `Extend` and `Concat` remain the existing bulk append/copy APIs.

Inherited streams

Operation	Contract
`TRY WriteAll(text AS STRING)`	Writes all UTF-8 bytes to stdout; returns byte count; FAILS IoError
`TRY WriteAll(bytes AS LIST OF INTEGER)`	Validates every element in 0..255 before output, writes all, returns count; FAILS IoError
`TRY ReadInto(BYREF bytes,start,count)`	Reads stdin into initialized slots only, returns number read or 0 at EOF; FAILS IoError, IndexError

WriteAll retries interrupted operations and loops over short writes; an error can follow partial external output. ReadInto retries interruption and returns a successful short read without waiting to fill the range. The current platform primitive reads up to 8,192 bytes per call. It preserves LEN and untouched elements. Zero-length reads return zero. Invalid byte-list output returns IoError code 22 before writing; OS failures retain positive errno codes. These dedicated operations do not transfer ownership of the inherited streams.

`Stdin()`, `Stdout()` and `Stderr()` return owned File duplicates (`FAILS IoError`) for normal file and buffered APIs. Cleanup closes the duplicate while preserving the original. Their underlying I/O position is shared; coordinate buffered read-ahead with other consumers. Standard-stream rules and examples.

The compiler now writes its entire executable through bulk WriteAll, avoiding one syscall per image byte. Packed LIST OF BYTE buffers are now supported by WriteAll/ReadInto alongside the compatibility LIST OF INTEGER APIs. ReadInto on packed bytes reads directly into initialized

storage; integer buffers use the bounded conversion chunk. The compiler uses a packed code buffer. See unsigned values and layout.

Optional fallback and parameter defaults

The maintained compiler implements lazy optional fallback with `??`:

```
TYPE Config
    Limit AS INTEGER=20
END TYPE
FUNCTION Run(config AS OPTIONAL OF Config=NONE) AS INTEGER
    DIM config=config ?? Config {Limit=20}
    RETURN config.Limit
END FUNCTION
PRINT Run()
PRINT Run(SOME(Config {Limit=42}))
```

`OPTIONAL OF T ?? T` produces T. A present value wins, including zero, FALSE, an empty STRING or an empty collection. Only absence evaluates the fallback. Nested optionals unwrap one layer. The operator associates to the right and binds below OR, AND, comparisons and arithmetic. Use parentheses when unwrapping multiple optional layers: `(outer ?? SOME(7)) ?? 9`.

Failures from a fallback follow normal checked contracts and cleanup. Its static contract remains checked even when a particular runtime value skips it. A fresh owned optional can donate its payload; a named resource optional needs explicit TAKE. A borrowed copyable value produces an independent value when ownership is needed. No implicit truthiness, ownership transfer or resource copying is added.

Reusing a parameter name

An explicit DIM may shadow an ordinary or BYVAL parameter. Its initializer reads the original parameter; subsequent statements see the new local. The local may have a different type. This works in functions, SUBs and block lambdas. BYREF and TAKE parameters cannot be shadowed. Duplicate locals in one scope remain errors; leaving a nested scope restores the original parameter's visibility.

The original owner and any older borrows keep their identity and lifetime. A new local does not acquire the original parameter's escaping BORROW provenance. Without DIM, an attempted parameter reassignment/type change suggests the explicit DIM declaration instead of silently changing its meaning.

Aggregate defaults

When omission simply means using a default, an optional is unnecessary:

```
TYPE Config
    Limit AS INTEGER=20
    Names AS LIST OF STRING=[]
END TYPE
FUNCTION Run(config AS Config=Config {Limit=20,Names=["default"]}) AS INTEGER
    RETURN config.Limit
END FUNCTION
```

Defaults admit correctly typed scalar constant expressions, records, choices, typed error values, nested lists, empty LIST()/SET()/DICT(), SOME/NONE and scalar static-table constants. Required parameters precede defaults. Arbitrary function calls and mutable bindings are not constant defaults. Generic default recipes are checked when their concrete omitted value is needed; generic bodies retain their ordinary instantiation checks.

Supplied arguments execute once in written order. Only omitted defaults are constructed afterward, in parameter order. Constants and constructor names resolve in the declaration's

module, including moved source-free .blibs and opaque records. Caller locals cannot capture them. Use braces when an ordinary function shares a constructor name; a default cannot silently substitute constructor behavior for that function call. Ordinary supplied arguments retain readonly borrowing. BYVAL defaults are independent owned arguments, just as supplied BYVAL arguments are independent snapshots. BYREF/TAKE defaults remain rejected.

Each required mutable owner has independent storage on every call. Fresh resource literal defaults are constructed and cleaned normally; they do not copy an existing resource. Task defaults use the same typed recipes and owned startup/cleanup rules. NONE can initialize an optional resource without creating a live handle.

Callback-dependent failure contracts

A wrapper should admit the errors of the callback it receives. Declare `FAILS LIKE action` when action is a checked CALLABLE parameter. Chapter 11 shows a complete wrapper and handler. Passing an infallible callback makes that use infallible; passing a callback declaring SigningFailure contributes that error family. The wrapper must separately declare or handle any unrelated error of its own.

Stored callbacks and configuration

FAILS LIKE can follow callbacks stored in supported records, optionals, lists and dictionaries. Inferred construction retains those contracts; a list combines its elements' admitted errors. An explicit wider annotation deliberately widens the checked value. Mutation must fit the storage type, including in loops. Writable references require matching contained contracts.

Copies, caught/rethrown errors and retained task outcomes preserve the checked set and concrete payload. A parent error view does not slice away descendant data. No generic InvocationError wrapper or hidden widening to Error is inserted. Broad-catch warnings describe declared error sets and call/callback origins; an open family is not an enumeration of every future descendant.

Returning values with the caller's contract

`AS LIKE formal` returns an owned value with that formal's representation and contained checked contracts. It is not a BORROW return or a lifetime extension. `AS Wrapper LIKE formal` carries the admitted invocation set into an owned wrapper shape. The body must satisfy that contract; ordinary explicit wide result types still widen it. Repeated positions sharing a callable shape conservatively use their admitted union.

Referring to a known operation

A signature reference such as `FAILS LIKE Decode(AS STRING)` uses an operation's checked declaration without running it. Spell every parameter type, including `BYREF AS Type` or `TAKE AS Type` when required. An optional result annotation, such as `LIKE Decode(AS STRING) AS Customer`, can resolve result-only context. The reference needs a unique visible overload. Value expressions, omitted/default arguments and arbitrary runtime evaluation do not belong in a signature reference.

Cyclic or excessively deep dependencies are rejected. Taking a function with a dependent result as a first-class callable is currently restricted; use an ordinary checked call. C exports and plain C imports require their supported fixed contracts. Compatible BASIL libraries preserve these dependencies through their public interfaces, together with ordinary ownership and cleanup checks.

Local functions, callable values and visibility

Local declarations and expression bodies — implemented

Noncapturing FUNCTION/SUB declarations may appear in an enclosing lexical block. They require declaration before use, can recurse and do not become module exports. Captures and generic-parent local specialization remain restricted. A local helper cannot read an outer local as an implicit capture.

```
PRINT Square(3.0)
FUNCTION Square(X AS DOUBLE) AS DOUBLE = X * X
```

Expression bodies use ordinary checking/ownership/error rules and need no END FUNCTION. SUB has no successful result and allows bare RETURN/fallthrough; returning a value is an error. FUNCTION must return a matching successful value on every reachable successful exit; bare RETURN/fallthrough fails. Both support FAILS and ordinary suspending calls. RETURN ERROR selects failure, not successful return.

Checked function values — implemented

```
FUNCTION Apply(handler AS CALLABLE(INTEGER) AS INTEGER, value AS INTEGER) AS INTEGER
    RETURN handler(value)
END FUNCTION
FUNCTION SquareInt(value AS INTEGER) AS INTEGER = value * value
PRINT Apply(SquareInt, 7)
```

A CALLABLE signature lists parameter types/modes, optional AS success type (no AS means SUB), and optional `FAILS (ErrorType, ...)`. Named noncapturing functions produce callable values; indirect calls enforce access/result/error contracts. Owning copies have independent cleanup. Signatures cannot be compared or hashed. Read-only assignment may widen a failure set; BYREF signatures must match exactly. Callables use required positional arguments; declaration parameter defaults/names are not an unchecked indirect-call facility. See checked-call ABI and layout.

Generic declarations can use type parameters inside signatures. Earlier argument types supply context for overloaded function references; signatures must become concrete before invocation. Nested signatures, declared errors, BYREF/BYVAL modes and borrowed-result origins retain their ordinary checks. For example:

```
FUNCTION Apply[T](value AS T, transform AS CALLABLE(T) AS T) AS T
    RETURN transform(value)
END FUNCTION
```

This does not infer an arbitrary error set: a fallible generic callback needs an explicit checked FAILS contract. Generic `CALLABLE(TAKE T)` remains rejected when T is unresolved; concrete resource-consuming signatures work. Inferring T from an overloaded callback before other arguments provide context remains ambiguous.

Noncapturing foreign handlers

`CALLBACK(...)` selects a generated C ABI entry; `CALLABLE NOSUSPEND(...)` keeps the BASIC ABI for reviewed binding adapters. C CALLBACK must be infallible; BASIC NOSUSPEND may retain declared errors. Both check nonsuspending targets and transitive calls. Neither captures locals. C callback creation binds its coordinator thread before C can invoke it. Unsafe functions cannot become safe delegates. Full contracts and GTK example.

PUBLIC and PRIVATE — implemented scope

Module FUNCTION/SUB, TYPE, ERROR, ENUM and CONST declarations support PUBLIC/PRIVATE. Unmodified declarations remain public for compatibility. PRIVATE is enforced for qualified imports; public signatures cannot leak private nominal types/error sets. Local declarations cannot be exported with PUBLIC. PUBLIC is not C EXPORT or permission for unsynchronized mutation. Imported modules execute no initialization. Mutable module variables remain unimplemented; opaque types can expose explicitly PUBLIC readonly fields under the record rules. PUBLIC DIM does not reinterpret entry locals as globals. Imports defines current name resolution.

Lambdas and captures — implemented subset

```
minimum=10
matches=Filter([1,10,20],(n) => n>=minimum)
square=(x AS DOUBLE) => x*x
PRINT LEN(matches)
PRINT square(3.0)
```

The expression form is `(parameters) => expression`; `()` has no parameters. Use FUNCTION(parameters) [AS result] [FAILS ...] through END FUNCTION for a block, or SUB(parameters) through END SUB for a procedure. Parameter types need AS or a concrete expected CALLABLE. Results and escaping errors infer from ordinary body checking unless declared; inconsistent/missing successful returns are rejected. An inferred lambda can specialize inside a generic parent.

Copyable free locals become independent readonly snapshots at construction. Captured strings/collections/callables own their copies and clean up on ordinary, failure and cancellation exits. Copies of closures are independent values. A later mutation of the original does not change a capture. Explicit local DIM and nested lambda parameters bind local names, rather than capturing matching outer resource variables. Named local helpers still do not capture implicitly.

Resources and task/channel capabilities cannot be captured. Pass an owned context explicitly to a library API, using TAKE for named resources, or pass ordinary checked function arguments. Mutable shared capture cells, borrowed/owning resource captures, anonymous CALLBACK/NOSUSPEND adaptation and nested named declarations inside anonymous bodies are not implemented. No GC/reference counting is added.

Value equality, ordering, hashing and display

Canonical operators define a nominal type's public value behavior. Declare them in the type's defining module. Ordinary functions with the same names do not register operators. Each operation takes required read-only operands of the exact type; it cannot have generic parameters, defaults or escaping checked failures.

Declaration	Result	Used by
OPERATOR Equal(a AS T, b AS T)	BOOLEAN	Equality expressions, searches and key comparison
OPERATOR Compare(a AS T, b AS T)	CompareResult	Ordering, comparisons and default sorting
OPERATOR Hash(value AS T)	INTEGER or UINT64	Hash-based set/dictionary lookup
OPERATOR Display(value AS T)	STRING	PRINT, interpolation, explicit Display and logging

Chapter 13 contains complete examples. Default copyable-record equality compares fields recursively. Custom Equal disables automatically generated Hash and Compare, so supply coherent versions when the type needs them. Compare without Equal derives equality from CompareResult.Equal, but still needs a compatible Hash for hash-key use. Explicit per-sort comparators are a separate policy.

Equal keys must hash identically. Equality and ordering must obey their normal mathematical laws: symmetry, transitivity and consistency with equality. The compiler checks signatures and allowed effects; it cannot prove arbitrary laws in your code. Incorrect laws can produce wrong collection results or poor performance without bypassing memory safety.

Unordered values and keys

CompareResult has Less, Equal, Greater and Unordered alternatives. A custom Unordered result makes ordered relations false. Default sorting with a custom Compare that reports Unordered raises UnorderedComparisonError and preserves the original input. Default DOUBLE sorting separately uses a NaNs-last policy. Sorting ties do not imply IEEE equality.

NaN is rejected in set members and dictionary keys, including nested fields, independently of a custom operator ignoring that field. Signed zero is one key; infinities are valid. Resource equality, ordering and hashing are rejected. Display alone does not make a resource copyable, comparable or hashable.

Predictable effects

Operator bodies can compute, allocate and change their own locals. They cannot perform I/O, inspect clocks/randomness or mutable native state, modify external storage, start tasks, explicitly suspend, or invoke unsafe/unverified foreign operations. These restrictions also apply to helpers they call. Indirect function calls are conservatively restricted; NOSUSPEND alone does not prove purity. The checks apply even when an operator is unused.

Ordinary task scheduling can still occur at compiler-generated safe points. Equality/hash/comparison operations complete their enclosing collection work before acknowledging cancellation. A long-running custom operator can therefore delay cancellation. Display follows ordinary checked call cleanup and can acknowledge cancellation while rendering. Neither policy establishes termination or a hard response-time guarantee.

Displaying nominal values

Display returns independent text while borrowing its operand, including a resource. No automatic field dump, implicit concatenation or arbitrary STRING conversion is added. Width, alignment, fill

and precision apply to the returned text as STRING formats; numeric specifications are rejected. A derived record needs its own Display declaration: ordinary extension does not introduce virtual dispatch.

Direct nominal logging arguments are displayed after severity filtering and once per argument even if a field is repeated. Explicit Display/LogArgument calls are ordinary eager expressions. A deferred zero-argument function runs after filtering and retains its real checked failures. Display does not transfer the operand or extend an escaping resource borrow.

Explicit Display also supports strings, numeric scalars, BOOLEAN and UUID. BOOLEAN conversion uses True/False; existing scalar PRINT spelling is unchanged. Erased root Error does not dynamically discover a descendant's display hook. Compatible BASIL packages preserve declared capabilities; older tools may require a rebuild or upgrade for newly supported declarations.

JSON

Everyday typed values

```
TYPE Customer
    Id AS INTEGER
    Name AS STRING = "Unknown"
    Nick AS OPTIONAL OF STRING = NONE JSON OMITNONE
END TYPE

customer = DecodeJSON("{""Id"":42}") AS Customer
PRINT EncodeJSON(customer)
DIM other AS Customer
other = DecodeJSON("{""Id"":7,""Name"":""Ada""}")
PRINT EncodeJSON(other, JsonCodec(Pretty=TRUE, Indent=2))
```

A complete call's AS annotation or known destination supplies its result type. `ToJSON(value[, codec])` returns JsonValue; `FromJSON(value[, codec]) AS T` converts an existing JsonValue. EncodeJSON writes text; DecodeJSON accepts STRING or packed BYTE list. Failures use the normal checked FAILS/CATCH channel. Inline TRY is optional; allocation failure remains fatal.

Supported standard mappings:

BASIC value	JSON representation
BOOLEAN / STRING	Boolean / UTF-8 string
INTEGER, BYTE, UINT32, UINT64	Exact integer number
DOUBLE	Finite JSON number; NaN/infinity fail
OPTIONAL OF T	null for NONE; otherwise T's representation
LIST OF T / SET OF T	Array, preserving iteration order; decoding a set applies ordinary duplicate elimination
Ordinary record / ERROR record	Object of fields
ENUM	Its integer value; decoding checks declared membership
Closed choice	Exactly one member, e.g. `{"Text":"hi"}` or `{"Empty":null}`
DICT OF K TO V	Ordered object; key rules below
ARRAY OF T	Zero-based rank-one array, or a bounds-preserving object below
JsonValue	Its represented JSON tree, with no original application type identity

Resources cannot be serialized or copied into formatter transport. A function value has no standard JSON representation; an explicitly registered formatter may give it a checked representation. Unsupported active fields fail rather than exposing handles or addresses. JSON EXCLUDE can hide such copyable fields during encoding; decoding still requires valid initialization.

Dictionary STRING keys are ordinary member names. Other keys use their encoded JSON text as the member name, unless their selected converter's representation is STRING, in which case that string is the name. Decoding reverses this rule using the statically known key type and codec. Distinct BASIC keys producing the same name, or distinct names decoding to an equal key, fail with code 3. Thus `{"1":2,"1.0":3}` cannot silently overwrite a DICT OF INTEGER TO INTEGER entry. These non-string-key conventions extend JSON's string-only object names.

A nonzero-based or multidimensional array encodes as `{"$bounds":[lower,upper,...],"$values":[flat row-major values...]}`. Decoding validates signed intervals, rank, shape arithmetic and cell count before constructing an array. A plain JSON array decodes to a zero-based rank-one array. Shape metadata uses standard integers independently of data formatters.

Dynamic construction and edits

`JsonObject()` / `JsonArray()` create independent empty containers. For nested objects, use `JsonObject([JsonField("name",value), ...])`. JsonField returns an ordinary owned JsonMember (Name, Value); typed leaves use ToJSON and retain its checked converter failures. A JsonValue argument is adopted/copied with ordinary BYVAL semantics. Duplicate names replace the value in their first insertion slot. The existing dictionary-taking JsonObject and list-taking JsonArray overloads remain.

`JsonPut(BYREF object,key,value)` inserts/replaces a member and returns TRUE for insertion, FALSE for replacement. Typed inputs use ToJSON. A wrong-kind target raises JsonError code4; conversion failures preserve their declared type and leave the destination unchanged. Explicit BYREF works for named objects, fields and checked indexed locations; root loans exclude invalidation through aliases/tasks. JsonValue arguments are independent BYVAL snapshots, so inserting an object into itself stores its previous value, without a cycle. Mutations already committed before cancellation follow ordinary BYREF rules; there is no rollback transaction.

Defaults and field clauses

Shared SERIALIZE rules also apply; JSON clauses override the corresponding shared property. See shared precedence.

```
TYPE Customer
    Id AS INTEGER
    Name AS STRING = "Unknown" JSON NAME "full_name"
    Password AS STRING = "" JSON WRITEONLY
    ServerValue AS INTEGER = 0 JSON READONLY
    Internal AS INTEGER = 0 JSON EXCLUDE
    Nick AS OPTIONAL OF STRING JSON OMITNONE
    Tags AS LIST OF STRING = []
END TYPE
```

Clause	Effect
NAME “name”	Wire name in both directions; default is original field spelling
READONLY	Encode, exclude from decoding; no change to BASIC mutability
WRITEONLY	Decode, exclude from encoding
EXCLUDE	Exclude both directions
OMITNONE	Omit an absent optional on output; otherwise NONE writes null
ENCODE WITH F / DECODE WITH G	Explicit converter for that field

JSON member names are case sensitive. Conflicting/repeated clauses, duplicate wire names and OMITNONE on a nonoptional field are compilation errors, including on unused types. READONLY and WRITEONLY cannot be combined; EXCLUDE cannot be combined with either. A supplied READONLY/EXCLUDE member is rejected under the standard strict policy; `JsonCodec(IgnoreUnknown=TRUE)` ignores it like any unknown member. Duplicate input keys are rejected even under that policy.

Missing fields use an explicit default, otherwise NONE for OPTIONAL, otherwise a checked missing-field error. Excluded/READONLY fields follow these same initialization rules. Explicit null means NONE only for OPTIONAL. Invalid supplied values and nonoptional null fail; defaults do not hide bad data. OPTIONAL field converters accept/return the payload type; missing/default/null handling precedes them.

Field defaults also apply to ordinary record construction. Current forms are checked compile-time scalar/string/enum expressions, NONE, and empty `[]`, `LIST()`, `SET()` or `DICT()` of the matching field type. Effectful factories and nonempty aggregate defaults are not supported. Explicit constructor arguments run in written order, then omitted defaults in declaration order. Each collection default is independent. Ordinary construction still requires an explicit value or default for every field;

implicit NONE for a missing optional is a decoder rule. Failed decoding destroys partial owners and preserves an existing assignment target.

Type converters and codec profiles

```
TYPE Money JSON ENCODE WITH FormatMoney DECODE WITH ParseMoney
    Cents AS INTEGER
END TYPE
FUNCTION FormatMoney(value AS Money) AS INTEGER = value.Cents
FUNCTION ParseMoney(value AS INTEGER) AS Money
    RETURN Money { Cents=value }
END FUNCTION

codec = WithJsonFormat(JsonCodec(Pretty=TRUE, Indent=2), FormatMoney, ParseMoney)
PRINT EncodeJSON(Money { Cents=1234 }, codec)
```

A pair's original type comes from encoder input/decoder result; encoder output must match decoder input. The representation is another checked BASIC value, not unchecked raw JSON. STRING results are quoted/escaped. A representation's root uses structural handling, avoiding immediate re-entry into the same rule; its nested values use ordinary recursive selection.

Precedence independently in each direction: JSON field override, shared field override, supplied profile rule, JSON type default, shared type default, standard generated handling. Profiles may hold many original types using the same representation type. Replacing a rule changes only that type and leaves the original profile unchanged. There is no mutable global registry. Profiles example.

The compiler retains a codec's admitted errors in its static type and through compiled libraries. Inferred empty codecs must be configured in their initializer or assigned to a new binding; explicit wider JsonCodec annotations admit a wider checked set. Wrapper functions declare `FAILS JsonError, LIKE codec`; factories can retain errors with `AS JsonCodec LIKE Encoder(AS T), LIKE Decoder(AS R)`. Private rules cannot be mutated. Compiled profile contract.

Current registration accepts unambiguous named, nongeneric, noncapturing functions with one read-only argument, or a second read-only JsonLocation/SerializationLocation argument. Reserved __ names, callable variables, overloaded ambiguous names, lambdas and resource representations are not accepted here. General checked function values remain available elsewhere.

JsonError automatically carries the JSON pointer, byte offset and message. Other converter errors retain their declared type and fields without an implicit wrapper or mutation. A context-aware converter can retain the supplied location explicitly:

```
ERROR InvalidDate
    Text AS STRING
    Location AS JsonLocation
END ERROR
```

Its converter can declare `(text AS STRING, location AS JsonLocation)` and return `ERROR InvalidDate { Text=text, Location=location }`. Offsets refer to the start of the supplied value; encoding/FromJSON lack source offsets and use −1. Runnable typed failure. One-argument custom errors are unchanged; they do not acquire hidden location fields. JsonError raised by a converter is normalized to the containing value's wire location. Converter FAILS sets are preserved. An adapter admits the supplied codec's immutable checked set and the selected value schema's converter closure. JSON/shared choices within that schema are conservatively combined because context selects them at runtime; unrelated registrations elsewhere in the source graph do not broaden its contract.

Dynamic data, validation and limits

JsonValue is a closed choice with NullValue, BooleanValue, NumberValue (STRING), StringValue, ArrayValue (LIST OF JsonValue), and ObjectValue (ordered DICT). Use JsonNull/JsonBoolean/JsonNumber/JsonString/JsonArray/JsonObject constructors. JsonNumber validates a string's grammar; the writer revalidates manually constructed NumberValue payloads. Exact number spellings, including large exponents and −0, are retained. Numeric getters reject type/range/fraction mismatches, with integer conversion performed without rounding through DOUBLE.

`Find`, `Get(value,key,fallback)`, `Contains`, `Keys` and `Values` operate on dynamic objects. `JsonAt(value,index)` checks an array position. `JsonGetString/Boolean/Array/ Object/Integer/UInt64/Double`, JsonNumberText and IsJsonNull provide checked access. JsonPointer appends an RFC 6901 escaped member/index segment. ParseJsonDocument additionally retains a Locations map; typed DecodeJSON uses it.

The parser enforces RFC 8259: UTF-8, strict number grammar/whitespace, escaped controls, paired surrogates, no BOM, trailing content/commas or duplicate keys. It does not normalize Unicode. Compact/pretty output preserves values, exact numbers and member order and has no trailing newline. Indent is spaces per level, 0..16. Root depth is zero.

JsonCodec defaults: Pretty=FALSE, Indent=2, MaxDepth=32, MaxInputBytes=8388608, MaxOutputBytes=8388608, IgnoreUnknown=FALSE. Depth is 1..64; byte limits are nonnegative. Limits apply before input conversion/output growth and during recursive conversion. A guarded coroutine may report a stack-budget limit before the requested depth. Output budgets count encoded bytes, including escaping and pretty whitespace; they are not a total heap quota. Long BASIC loops have budgeted cancellation/scheduling checkpoints; native buffer transactions do not suspend.

JsonError fields are Code, Offset (zero-based byte, −1 unavailable), Path (JSON pointer; empty at root), Message. Codes: 1 syntax, 2 UTF-8, 3 duplicate/collision, 4 type/range/representation, 5 missing field, 6 unknown/disallowed field, 7 limit, 8 invalid configuration. Invalid BYTE input currently reports UTF-8 offset 0. Unhandled JsonError diagnostics include the path, offset and message after cleanup.

User-defined formats

Shared serialization defines SERIALIZE clauses, checked typed FORMAT options, SerializationSchema and FormatMetadata. ToValue/FromValue apply shared rules/converters and ignore JSON-specific rules/profiles. They retain the existing JsonValue-compatible transport and JsonError channel. Legacy ValueSchema continues to report the effective JSON mapping. See the runnable custom-codec example. Richer format document models, EMBED evaluation, YAML/TOML/XML and generic record declarations remain separate work.

Files and JSON Lines

API	Behavior
ReadJSON(file[,codec]) AS T	Bounded stream read to EOF, then typed decode
ReadJSONFile(path[,codec]) AS T	Open/read with automatic resource cleanup
WriteJSON(file,value[,codec])	Complete encoding before stream write; I/O failure can leave partial output
WriteJSONFile(path,value[,codec])	Complete encoding, then existing atomic replacement API
ReadJSONLine(file,BYREF reader[,codec]) AS OPTIONAL OF T	One record, NONE at clean EOF
WriteJSONLine(file,value[,codec])	Compact record plus LF, even for a pretty profile

Initialize reader with JsonLinesReader(). It buffers input and exposes Line and Offset. LF/CRLF and an unterminated final record work; blank lines are invalid records. A malformed consumed line can be caught and the next line read. JsonError reports the stream byte offset and record number; custom JsonLocation offsets are relative to that line's JSON document. Input/output limits apply

per record. Do not mix direct reads or different readers on a buffered stream. Independent copies of reader state do not duplicate the file position.

Read helpers inherit worker-backed File reads and cancellation acknowledgement. Writes inherit existing WriteFile/atomic-write behavior, including synchronous regular-file writes. Flush/finishing errors remain explicit operations; no new normal-cleanup guarantee is made for fatal allocation failure.

HTTP and HTTPS

ReadJSONBody(request[,codec]) AS T handles a buffered request. Its streaming overload also takes connection and BYREF exchange and reads within both HTTP and codec limits. It checks application/json or application/*+json and UTF-8 charset; missing/invalid media types return HTTPError 415, oversized bodies 413. Parsing and typed conversion retain JsonError/declared converter failures.

JSONResponse(value[,status]) constructs a response; the full overload takes `(value,status,headers,codec)`. It installs application/json and rejects conflicting framing headers and body-forbidden statuses. ReadJSONResponse(response[,codec]) AS OPTIONAL OF T rejects non-2xx with owned HTTPStatusError {Status,Headers,Body}, returns NONE for 204, and decodes other 2xx bodies (empty 200 is an error). JSON null for OPTIONAL OF U is SOME(NONE), distinct from no response content.

HTTPGetJSON(url) AS OPTIONAL OF T and HTTPPostJSON(url,body) AS OPTIONAL OF T use existing HTTP/TLS clients. The full GET overload takes `(url,BYREF response,codec,options)`; POST adds body after url. Initialize response with HTTPResponse(). It retains status, headers, body and trailers before typed decoding, including on decode/status failure. The codec input limit caps transport body buffering. The ordinary HTTPGet/RequestHTTP + ReadJSONResponse composition also keeps complete response metadata.

`RouteJSON(BYREF router,method,pattern,Handler[,codec])` accepts a named one-argument function: decode its checked input type, call it, encode its return value with 200. A handler returning HTTPResponse controls status/headers directly. It uses existing buffered routes and stores an independent per-route profile; no global registry or unchecked captures. Mutable router slots explicitly admit Error; registration does not infer a narrower lifetime-wide union. ServeHTTPChecked propagates original typed failures. Convenience ServeHTTP/ServeHTTPS handle JsonError as 400 and report other converter/route errors by concrete type name with a 500 response when headers have not started. They are request error boundaries; retain a task running the checked server when the caller needs typed failure observation. HTTP API retains its existing limits. Small server example.

Opaque representations

Opaque types require a declared or profile formatter for encoding/decoding. Automatic JSON/shared record mapping raises JsonError code4 instead of exposing private state or constructing invalid representations. A formatter must use the defining module or public factories. For native collections, serialize a logical snapshot or register a concrete formatter; physical ring/heap/cache state is not a wire schema. Ordinary public records keep generated field mapping.

Shared serialization and custom format metadata

Shared rules and format overrides

```
TYPE Customer
    Id AS INTEGER
    FullName AS STRING SERIALIZE NAME "full_name" JSON NAME "name"
    Password AS STRING = "" SERIALIZE WRITEONLY
    Internal AS STRING = "private" SERIALIZE EXCLUDE
    Nick AS OPTIONAL OF STRING SERIALIZE OMITNONE
END TYPE
```

SERIALIZE accepts NAME, READONLY, WRITEONLY, EXCLUDE, OMITNONE, ENCODE WITH and DECODE WITH. Their direction/default/initialization semantics match the existing JSON clauses. They are schema mapping rules, not changes to BASIC mutability. Defaults are ordinary TYPE field defaults. NAME accepts a string literal or a parenthesized constant STRING expression; parentheses keep a following TYPE field default unambiguous. This also preserves multiline wire names through compiled metadata. TYPE headers accept shared ENCODE WITH / DECODE WITH defaults, including on choices. Case payload declarations do not accept field annotations.

ToValue/FromValue apply shared rules and converters; they ignore JSON-specific rules/profiles. EncodeJSON/DecodeJSON and ToJSON/FromJSON use the effective JSON mapping. Existing programs using only JSON clauses retain their previous behavior.

JSON clauses override the same shared property regardless of source order: NAME replaces NAME; READONLY/WRITEONLY/EXCLUDE replace the shared access group; each converter direction replaces that direction. OMITNONE is additive. There is currently no INCLUDE or explicit undo-OMITNONE clause. Contradictions within one scope, duplicate effective names in either scope and OMITNONE on a nonoptional field are compile errors, even without an adapter call.

For JSON converters, precedence independently by direction is JSON field override, shared field override, supplied JsonCodec rule, JSON type default, shared type default, generated standard handling. A selected encode/decode pair must have matching representation types. Shared converters can accept a second readonly SerializationLocation { Path, Offset }; the existing JsonLocation form also remains accepted. ToValue/FromValue have no parsed text locations, so Offset is −1; typed DecodeJSON supplies the parsed byte offset. Checked errors and partial cleanup use the ordinary call contracts. Allocation failure remains fatal.

Typed format options

```
TYPE PacketField
    Number AS INTEGER
    Label AS STRING = ""
END TYPE
TYPE Customer FORMAT PacketField { Number=100 }
    Id AS INTEGER FORMAT PacketField { Number=1 }
    Name AS STRING FORMAT PacketField { Number=2,Label="customer" }
END TYPE
```

FORMAT attaches an independent typed constant record. It can occur on a TYPE header or ordinary TYPE field, together with SERIALIZE/JSON and other FORMAT annotations. A given options type can occur once per declaration. Identity follows normal imports and nominal types; two modules' identically named options are distinct. The compiler checks the literal, not an open dictionary of unchecked attributes.

Supported option values: checked scalar/string/enum constant expressions, NONE, SOME(constant), nested ordinary record literals, and matching empty lists/sets/dicts. Omitted

options need an ordinary declared field default. Record nesting is bounded at 32. Arbitrary calls, nonempty collection literals, callable values, resources, choice literals and runtime variables are not constant annotation values. Checked integer overflow and constant/default type errors are rejected. No annotation code runs at build time. Type names, field names, private constants and imports use ordinary resolution/access rules.

Custom formats define their own option types and interpret them in normal library code. For example, an XML library could define XmlField with Name/Attribute options; a binary codec could define numeric field identifiers. Merely naming an options type XmlField does not give it compiler privileges. No built-in XmlField, YamlField, TomlField or JsonField metadata type is supplied in this milestone. Existing JSON clauses remain the JSON-specific mapping interface.

Public schema and metadata queries

For recursive field/element references, enums/choices, bound generic arguments and converter representations, use TypeSchema / SerializationTypeSchema / JsonTypeSchema. The older flat queries below remain supported.

```
DIM witness AS OPTIONAL OF Customer=NONE
schema=SerializationSchema(witness)
options=FormatMetadata(witness,"Id") AS OPTIONAL OF PacketField
IF LET field=options THEN
    PRINT field.Number
END IF
```

API	Result
SerializationSchema(witness)	Record name and ordered LIST OF SerializationField
FormatMetadata(witness[,field]) AS OPTIONAL OF M	Owned M or NONE; M inferred from AS or typed destination
ToValue(value)	Checked shared mapping into the existing JsonValue-compatible tree
FromValue(tree) AS T	Checked shared construction of T with defaults and initialization

SerializationField contains Name (physical BASIC spelling), WireName (shared name), TypeName, IsOptional, HasDefault, ReadOnly, WriteOnly, Excluded and OmitNone. Query fields by physical BASIC name, case-insensitively; omitted/empty field means the TYPE header. An unannotated field/type or unrelated metadata type returns NONE. Header metadata is separate from field metadata; codecs can apply their own explicit inheritance/default policies. Each successful query constructs a fresh independent value. Mutating a returned options collection cannot change subsequent queries. The optional type witness never reads or constructs an uninitialized record.

Legacy ValueSchema/ValueField remain available: JsonName and their access flags report the effective JSON mapping. Schema queries expose no pointers, mutable views, erased addresses or general runtime reflection. The compiler generates ordinary typed BASIC construction/access functions; metadata errors use source locations. The defining record's metadata is not automatically executed.

The codec is responsible for interpreting its own options, rejecting unsupported representations and checking format rules such as unique positive field numbers. Recoverable codec errors must be caught or declared normally. Unrelated annotations have no effect on a codec that does not use them.

Boundaries and examples

The runnable application uses a fully user-written numbered-field codec, with strings, exact-number text, Boolean and absent-optional payload tags. It validates field numbers and wire tags; the intentionally small example rejects nested payloads, tabs/newlines in strings and unknown/duplicate fields. It uses no JSON text parser/writer for its wire format. This demonstrates a public extension point, not general binary-format conformance.

The existing ToValue/FromValue transport is still JsonValue and reports JsonError; it is a bounded JSON-compatible value tree, not a lossless universal YAML/TOML/XML document model. Codecs can use their own typed representations and normal checked construction; richer common transport/field visitors remain a checklist item before claiming native temporal/tagged values, arbitrary mapping keys or mixed XML content. Typed metadata does not force custom options or codec documents into JsonValue. Generic user codec functions must declare the custom converter errors they allow to escape, alongside their own failures. Automatic per-specialization generic error-set inference is not implemented; a missing declaration is rejected. Format-specific profiles are ordinary codec-owned configuration; no global registry or arbitrary attribute grammar is introduced.

Recursive type metadata

The maintained compiler exposes a finite graph of **declared types**, without constructing application values. Use an OPTIONAL witness containing NONE, including for records with required fields and resource-owning types.

```
TYPE Node
    Name AS STRING
    Children AS LIST OF Node=[]
END TYPE
DIM witness AS OPTIONAL OF Node=NONE
graph=JsonTypeSchema(witness)
root=graph.Types[graph.Root]
PRINT root.Name
FOR EACH field IN root.Fields
    PRINT field.Name; ":"; graph.Types[field.DeclaredType].Name
NEXT
```

Query	Mapping
`TypeSchema(witness)`	Physical BASIC fields/default presence, without serialization overrides
`SerializationTypeSchema(witness)`	Shared SERIALIZE rules and type/field converters
`JsonTypeSchema(witness)`	Effective declaration-default JSON rules, including shared fallbacks
`FormatMetadata(graph,node[,field]) AS OPTIONAL OF M`	Independent typed FORMAT options or NONE; invalid node fails with JsonError

A TypeGraph has readonly public Root and Types fields. Types is a list of TypeDescriptor. References index that graph; **-1 means no reference**, including a unit choice alternative. Repeated/recursive references share nodes. IDs have no meaning in another graph. Opaque Identity strings distinguish nominal types and instantiated arguments; compare them as whole strings, without parsing them or using them as a persisted versioned schema identifier. Name is a display label, not a lookup key; generic records retain their base name and expose TypeArguments.

TypeDescriptor fields	Meaning
Identity, Name, Kind	Nominal/structural identity, display name, TypeKind
Fields	Declaration-ordered ordinary fields, including flattened base fields
ElementType	List/array/set element or optional payload reference
KeyType, ValueType	Dictionary key/value references
Members	Enum Name and signed 64-bit Value
Alternatives	Choice Name and ValueType (-1 for a unit alternative)
TypeArguments	Bound ordinary-record generic argument references, including phantom arguments
IsOpaque, IsResource	Representation privacy and checked ownership classification
WireSupported, UnsupportedReason, WireRepresentation	Standard wire-shape facts (before root converters) and explicit unsupported/dynamic reasons
EncodeType, DecodeType, HasEncoder, HasDecoder	Declared converter representation references and presence, independently by direction
BitWidth, IsUnsigned	Integer/DOUBLE numeric representation; width 0 when not a numeric scalar

TypeKind cases are Unsupported, BooleanValue, IntegerValue, DoubleValue, ByteValue, UInt32Value, UInt64Value, StringValue, RecordValue, ListValue, ArrayValue, DictionaryValue,

SetValue, OptionalValue, EnumValue, ChoiceValue, DynamicJsonValue, ResourceValue, FunctionValue and PointerValue. Native resources/callables/pointers, dynamic JsonValue, and unclassified categories explicitly lack a static wire shape. Resource records/containers remain nonserializable. A NONE witness never acquires, copies, transfers or destroys an application resource.

TypeFieldInfo contains Name (physical), WireName, DeclaredType, EncodeType, DecodeType, IsOptional, HasDefault, ReadOnly, WriteOnly, Excluded, OmitNone, HasEncoder and HasDecoder. Optionality and ownership reflect bound generic arguments. HasDefault means an explicit TYPE field default; an optional without one can still decode missing input as NONE. The access flags refer to wire directions, not BASIC mutability.

To interpret a wire view, select the direction's root converter when present, then visit its representation reference **with that representation's root converter bypassed once**, matching the codec. Field HasEncoder/HasDecoder distinguishes an explicit field override, including one returning the same declared type: visit that direction's field representation with the same bypass. Without a field override, visit DeclaredType normally. Children of a bypassed representation use ordinary selection again. A traversal's visited key therefore includes direction and bypass state, not only node ID; same-type normalization converters terminate.

WireSupported/WireRepresentation describe standard handling **before** root converters. Physical Fields remain facts even when a converter changes the wire shape. WireSupported=TRUE is not a transitive validation of the graph. After selecting/bypassing root rules, traverse relevant fields/elements/keys/values/cases and reject unsupported nodes, or interpret explicit user FORMAT schema options. Opaque nodes expose no fields/field annotations, even PUBLIC fields; their standard WireSupported remains FALSE. A selected declared converter can expose a known representation without exposing opaque storage. An encoder alone does not establish a decoder. A JsonValue representation stays explicitly dynamic, rather than being inferred from a sample converter result.

BASIC set equality does not guarantee unique JSON array items: exclusions or converters can map different members to identical JSON. SetValue retains the language kind; its wire representation is array without a uniqueness promise.

Standard wire representation labels are boolean/integer/number/string, object, array, nullable, object-encoded-keys, single-case-object and array-or-bounds-object. Choices use exactly one case-name property (a unit case has null). Nontext dictionary keys use the existing JSON key encoding, with the key type's selected mapping. Zero-based vectors use arrays; other dense arrays use `{"$bounds":[lower,upper,...],"$values":[...]}`. Bounds and vector shape are value facts, so the type describes both accepted forms. Converter presence delegates to EncodeType/DecodeType as described above. These are codec protocols, not provider-specific schemas.

FORMAT queries use physical BASIC field names case-insensitively; empty/omitted field selects the type header. Different nominal option types stay distinct; missing annotations return NONE. Returned values and copied descriptor lists are independent. Graphs provide no mutable field access, unchecked value construction, raw addresses or runtime converter registry. Per-instance JsonCodec profiles are not arguments to these declaration-default queries; custom runtime mappings need an explicit schema supplied by their library.

Logging

Logging is synchronous and needs no setup:

```
LogInfo("Hello, world!")
LogWarn("Request {0} took {1:.2f}ms",42,3.125)
```

The default destination is stderr, threshold Info, format `{timestamp} [{level}] {message}`. Levels, from least to most severe, are Trace, Debug, Info, Warn, Error and Fatal. Fatal does not exit. All six functions accept an optional leading `Logger` or `LoggerHandle`.

Numbered message fields use the same scalar format modifiers as f-strings; `{{/}}` escape braces. Repeated fields reuse the value. Literal formats receive compile-time checks; dynamic bad patterns produce logging diagnostics. STRING, BOOLEAN, INTEGER, BYTE, UINT32, UINT64, DOUBLE, Error values (type name) and `LogFieldValue` choices are supported. `LogArgument(value)` constructs a choice for heterogeneous argument lists passed to checked `WriteLog`. Nominal values with OPERATOR Display, including resource records, are borrowed and rendered after filtering; numeric formats reject for their text. Deferred functions may return these values without a manual LogArgument wrapper.

```
minimum=10
deferred=() => minimum*2
SetLogFormat("{level} {message}")
LogDebug("filtered {0}",deferred)
SetLogLevel(LogLevel.Debug)
LogDebug("computed {0} twice: {0}",deferred)
```

Ordinary expressions evaluate normally, even when filtered. A zero-argument callable argument instead runs once, synchronously, after filtering. Its declared errors remain the caller's checked errors. Captures are independent readonly snapshots established when constructing the callable; capture construction is ordinary eager work. Reuse a constructed closure in a hot loop. A named function or noncapturing lambda can borrow static callable data without allocation. `LogCallable(handler,label)` supplies a function label instead of invoking it.

Configuration and handles

`DefaultLogConfig()` returns an independent initialized record with fields Level, Format, Name, Path and Console. Console is `LogConsole.None`, StandardError or StandardOutput. `CreateLogger(config)` creates a scoped resource; `CreateLogger(path)` appends to that file with console output disabled. Resources close automatically. Use `Flush(logger)` or `Close(logger)` where a checked finishing result or early closure matters.

`GetLogConfig`, `SetLogConfig`, `SetLogLevel` and `SetLogFormat` accept a logger owner, handle, or the default logger when omitted. Formats/configuration are validated before replacement; an unsuccessful file replacement keeps the old configuration. `LogEnabled([logger,] level)` also returns false for a service with no destination.

Metadata fields are `{timestamp}`, `{date}`, `{time}` (UTC), `{level}`, `{message}`, `{file}`, `{line}`, `{function}` and `{logger}`. Source positions are inserted automatically by severity calls; function names use normalized compiler spelling. `WriteLog(logger,level,pattern,values)` has no automatic caller metadata.

`DefaultLogger()` and `LogHandle(owner)` return copyable access capabilities. They refer to the service without owning its lifetime; ordinary data still has independent values. `SetDefaultLogger(TAKE owner)` promotes its ownership to the program. A program-owned handle may select an already promoted service. Replacing the selection keeps earlier program-owned services alive for existing handles. Program cleanup finishes children and ordinary locals, then closes these services. Stale/closed handles are checked and cannot access a reused service slot.

Application destinations

`AddLogSink([logger,] handler)` registers a SUB taking `LogRecord`. Its fields are Timestamp (Instant), Level, Message, File, Line, FunctionName, Logger and Formatted. Handlers run in registration order. They may suspend and use ordinary file, database or network APIs. There is no logging thread or asynchronous record queue.

Stateful destinations use `CreateLogSink(context,handler[,finish])`, then transfer the sink to `AddLogSink`. Context is an independent owned argument; named resources require TAKE. The handler takes readonly or BYREF context followed by LogRecord; an optional finish SUB takes the same context/mode. A finish callback runs on checked Flush/Close and automatic service closure; normal resource cleanup then destroys the context. Finish is not a substitute for owned cleanup.

```
TYPE CountedOutput
    Count AS INTEGER
END TYPE
config=DefaultLogConfig()
config.Console=LogConsole.None
SetLogConfig(config)
sink=CreateLogSink(CountedOutput {Count=0},
    SUB(BYREF context AS CountedOutput,record AS LogRecord)
        context.Count+=1
        PRINT context.Count;": ";record.Message
    END SUB)
AddLogSink(TAKE sink)
LogInfo("first")
LogInfo("second")
```

`AddLogSink([logger,] context,handler)` is the direct form without a finisher. `ClearLogSinks([logger])` drops application sinks; the configured Path output is separate. Mutable context access is exclusive throughout suspension. Another task's logger operation queues until the active operation finishes; cancellation unregisters its wait and releases owned values normally. Same-service recursive logging reports an error instead of deadlocking. Application cycles between different services still require normal deadlock avoidance.

Severity calls report format/destination failures to stderr and continue other destinations. Checked `WriteLog` returns LoggingError after attempting destinations; configuration, Flush and Close also declare LoggingError. Deferred computation errors propagate unchanged, including a deferred LoggingError. Cancellation remains a distinct task outcome. Automatic cleanup reports finishing failures and still destroys owned contexts. Allocation failure remains fatal.

Dates, times, durations and clocks

These types and functions are in the standard prelude. No import, native date library or process timezone setting is required. Constructors and parsers validate their inputs. Checked failures are `DateTimeError` with `Kind`, `Message`, `InputByteOffset` (`-1` if inapplicable) and optional `UnderlyingOSCode`. Catch them in `TRY` blocks or declare them with `FAILS DateTimeError` in functions.

```
day = NewDate(2024, Month.January, 31)
PRINT FormatDate(AddMonths(day, 1, DateOverflow.Constrain))
instant = ParseInstant("2024-01-01T01:00:00.000000007+01:00")
PRINT FormatInstant(instant)
PRINT FormatDuration(Nanoseconds(-1))
```

Use	Types and operations
Calendar values	`Date`, `TimeOfDay`, `LocalDateTime`, `YearMonth`, `MonthDay`, `IsoWeekDate`. Use `NewDate`, `NewTime`, `NewLocalDateTime`, `Parts`, `WeekOfYear`, `AddDays`, `AddMonths`, `AddYears` and the typed `Parse`/`Format` functions.
A point on the Unix timeline	`Instant`, with `InstantFromUnixSeconds(seconds, nanosecond)`, scaled constructors, `UnixSeconds`, `NanosecondFraction`, `CompareInstant`, `ElapsedBetween` and `AddDuration`. The civil range is years −999,999,999 through 999,999,999, with astronomical year zero and the proleptic Gregorian calendar.
A local time with a stated offset	`UtcOffset`, `OffsetTime`, `OffsetDateTime`. Use `NewUtcOffset`, `AtOffset`, `ToInstant`, `OffsetOf`, `LocalPart` and `WithOffsetSameInstant`. A local datetime by itself is not an instant.
Exact elapsed time	`Duration`, with `Nanoseconds`, `Microseconds`, `Milliseconds`, `Seconds`, checked `Minutes`/`Hours`, `NewDuration`, arithmetic, `ParseDuration` and `FormatDuration`.
Calendar arithmetic	`Period` stores months and days. `NewPeriod`, `AddPeriod`, `CalendarBetween` and `DateTimeAmount` apply calendar rules first and exact duration second. `DateOverflow.Reject` is the default; `.Constrain` clamps a missing day to the end of the month. A calendar day in a zone can differ from 24 elapsed hours.
Timezone rules	`TimeZone`, `ZonedDateTime`, `ZoneDatabase`. Use `UtcZone`, `FixedOffsetZone`, `NewTransitionZone`, `LoadTimeZoneFile`, `LoadTimeZoneBytes`, `ZoneDatabaseFromEntries`, `FindTimeZone`, `AtZone`, `MapLocal` and `ResolveLocal`.
Clocks and deadlines	`NowInstant()`, `NewSystemClock`, `NewTestClock`, `NowMonotonic`, `ElapsedBetween`, `CompareMonotonic`, `DeadlineAfter`, `Remaining`, `IsExpired`.

`Duration` is one exact value with a signed `INTEGER` floor-seconds component and a normalized nanosecond fraction in `0..999999999`. For example, `Nanoseconds(-1)` has `DurationSeconds` of `-1` and `NanosecondFraction` of `999999999`. Constructors through `Seconds` accept every `INTEGER`; arithmetic can fail if the resulting seconds do not fit. `DurationNanoseconds`, `DurationMicroseconds` and `DurationMilliseconds` check the range of the requested total; the latter two also check precision unless a rounding mode is supplied. `RoundDuration` currently requires its quantum to fit signed nanoseconds.

The old `NanosecondCount` field is removed. Use the checked total accessors, or keep the wide `Duration`. For a sign test, `Compare(delay, Seconds(0))` returns `CompareResult.Less`, `.Equal` or `.Greater` without narrowing the duration. `SELECT CASE AFTER` rejects positive delays/deadlines outside the scheduler's signed nanosecond range with `DateTimeError` before registering any waits; nonpositive delays are ready immediately. TCP, TLS and resolver timeouts retain `IoError` (22 for negative input, 75 for range overflow). Terminal and GTK timeout handling also checks narrowing. `Sleep` still takes integer milliseconds. Rebuild `.blib` files with package ABI v12.

Zones and local time

Load an explicit TZif file, for example `LoadTimeZoneFile("America/Toronto", "/usr/share/zoneinfo/America/Toronto")`, or give `ZoneDatabaseFromEntries` a list of `ZoneDataEntry {Id=..., Data=...}` values. The database owns an immutable snapshot of those entries. There is no implicit zone lookup from

the environment or mutable global database. `MapLocal` reports unique, ambiguous, or nonexistent mappings. `ResolveLocal` takes explicit `LocalResolveOptions` for overlap/gap policy and defaults to rejecting both. Explicitly select how a calendar addition through a clock change is resolved.

TZif v1–v4 and supported POSIX footer rules are read with bounds and consistency checks. Unspecified local time types, missing rules and leap-aware TZif data fail explicitly. Fixed offsets compare by signed seconds. Named `TimeZone` values compare by ID and cached rules digest; `SameZoneRules` compares digests. `ZonedDateTime` equality includes the instant, zone ID and rules digest. `SameInstant` compares only timeline position. Offset datetime equality includes the local value and offset. Periods, mixed amounts and offset/zoned values do not have an implicit ordering. Use an explicit comparison for the intended axis.

Clocks

`NowInstant()` reads wall time. `NewSystemClock()` defaults to Linux `MonotonicBasis.ActiveTime` (does not count suspend); `BootTime` includes it. Elapsed measurements use `ElapsedBetween(NowMonotonic(clock), laterSample)`. Every `Clock` is an owned resource. Samples and deadlines are independent values, tied to the creating clock's domain. Comparing or using them with a different domain fails with `ClockDomainMismatch`. `DeadlineAfter` rejects negative durations; `Remaining` returns zero after expiry.

`NewTestClock(initialWall)` creates a deterministic clock. `AdvanceTestClock` borrows it `BYREF`, advances wall and monotonic time together, and commits neither change if either would overflow. `AdjustTestWallClock` changes only wall time. Test clocks drive these clock and deadline operations; scheduler waits still use the scheduler's system monotonic clock.

Text and storage

Strict ISO-style parsers use ASCII digits and explicit offsets where required. They do not guess missing fields or normalize invalid dates or leap seconds. `ParseOptions` can opt into ASCII whitespace trimming and bounded input. `FormatOptions` offers fixed fraction widths and checked rounding. Compiled patterns support numeric year/month/day/hour/minute/second/fraction and numeric offset fields with ASCII literals. See the calendar example.

`ParseRfc3339` preserves the timestamp's offset designation (including an unknown local offset). `ParseRfc9557` resolves supported named zone and ISO calendar annotations using a supplied snapshot and offset conflict policy. It retains supported noncritical annotations, but has no lossless re-emitter. `ParseHttpDate` and `FormatHttpDate` handle IMF-fixdate; obsolete HTTP date grammars are outside this profile.

Date, clock-of-day, local/offset datetime, instant, UTC offset, offset time, partial-date, ISO-week and duration values have default text codecs for typed JSON. `Period` uses a versioned object with exact month/day strings so mixed signs round-trip. A JsonCodec profile or a field converter can choose another representation. `InstantToStorageJSON`, `PeriodToStorageJSON` and `ZonedToStorageJSON` provide explicit versioned envelopes. Restoring a named zoned value requires a snapshot with the matching rules digest. SQLite uses ordinary typed cells: choose `SqlText(FormatDate(day))` or a documented epoch unit with `SqlInteger`, and decode explicitly. No date type is inferred from SQL text or a numeric cell.

Locale-sensitive text, automatic OS timezone/database discovery, scheduler injection of test clocks, richer database mappings, and native TOML/binary date transport are separate checklist work.

UUID values

UUID is an independent copyable first-party value following RFC 9562. Its implementation is native BASIC, with OS randomness/clocks only at the platform boundary. It needs no UUID, TLS or hashing shared library.

```
id = UUIDv7()
PRINT FormatUUID(id)
PRINT f"id: {id}"
restored = ParseUUID(FormatUUID(id))
PRINT restored = id
```

Words, bytes and text

API	Semantics
`UUID(high,low)`	Two 64-bit words; High is the most significant half. INTEGER/UINT64 combinations are accepted; signed inputs preserve their two's-complement bits.
`UUID(bytes)` / `UUIDFromBytes(bytes)`	Exactly 16 BYTEs in network/big-endian order; UuidError on wrong length
`UUID(text)` / `ParseUUID(text)`	Checked text parsing; UuidError on malformed input
`UUIDBytes(value)`	Independent 16-byte list in network order
`UUIDGuidBytes(value)` / `UUIDFromGuidBytes(bytes)`	Explicit Windows GUID memory order: reverse first 4/2/2 bytes; remaining 8 unchanged
`FormatUUID(value)`	Canonical lowercase `8-4-4-4-12` text; infallible apart from fatal allocation failure
`FormatUUID(value,style)`	UuidFormat.Canonical, Compact, Braced, Parenthesized or Urn
`NilUUID()` / `MaxUUID()`	All-zero/all-one 128-bit values
`IsNilUUID(value)` / `IsMaxUUID(value)`	BOOLEAN tests
`UUIDVersion(value)`	Raw version nibble 0–15, including 0 for Nil and 15 for Max
`UUIDVariant(value)`	UUIDVariantKind.NCS, RFC, Microsoft or Reserved

Parsing accepts canonical 36-character text, compact 32 hex digits, matching braces or parentheses around canonical text, and `urn:uuid:` canonical text. Hex and the URN prefix are case-insensitive. It does not ignore whitespace, trailing junk, embedded NUL or malformed separators. UuidError.Offset is a zero-based UTF-8 byte position where applicable; component/range errors use −1. Code 1 denotes text syntax, 2 binary length, 3 component range, 4 incompatible version/variant and 5 exhausted clock-generator timestamp space. Message describes the failure.

Construction/parsing preserves all 128 bits, including unknown/reserved version or variant bits. Only version-specific constructors impose their format. Nil and Max are values, distinct from optional NONE. High/Low are PUBLIC opaque fields, hence read-only outside the prelude; raw construction remains through the checked factories.

Equality, ordering and Hash use the two unsigned halves. Sets/dictionaries accept UUID keys; sorting follows network byte order. UUIDs support ordinary string-style f-string padding/precision, not integer numeric presentations. C interoperability uses explicit words/bytes; no C struct ABI is inferred.

Generating versions

API	Behavior
`UUIDv4()`	122 random bits from the OS source; IoError on failure
`UUIDv7()`	Current Unix milliseconds plus 74 OS-random bits; UuidError/IoError

API	Behavior
`UUIDv7(milliseconds)`	Checked unsigned 48-bit timestamp and fresh OS randomness
`UUIDv7(milliseconds,entropy)`	Deterministic construction from exactly 16 entropy bytes; replaces timestamp/version/variant bits
`UUIDv3(namespace,name)`	Deterministic MD5-based name UUID
`UUIDv5(namespace,name)`	Deterministic SHA-1-based name UUID
`UUIDv1()` / `UUIDv6()`	Current Gregorian timestamp with a fresh random multicast node/clock sequence; no hardware MAC lookup or shared global state
`UUIDv1(ticks,sequence,node)` / `UUIDv6(ticks,sequence,node)`	Explicit 60-bit Gregorian 100-ns ticks, 14-bit sequence and 48-bit node
`UUIDv8(high,low)`	Explicit custom data; sets RFC variant and version 8, preserving other bits

Namespace helpers: UUIDNamespaceDNS(), UUIDNamespaceURL(), UUIDNamespaceOID() and UUIDNamespaceX500(). String names hash their exact UTF-8 bytes; there is no implicit case folding, normalization or DNS/URL canonicalization. MD5/SHA-1 helpers are private compatibility implementations for UUIDv3/v5, not authentication/password APIs. V2 generation is not supplied; raw v2 values still parse/round-trip. V8 has no generic random/name format: the caller defines its custom data layout.

V7 uses fresh randomness for every call. It sorts by timestamp across different milliseconds; within one millisecond order is random. It follows the supplied/OS clock on rollback, so it does not promise monotonic creation order. There is no hidden process-wide counter. Applications needing strict monotonic v7 generation need a separately specified generator; do not infer it from the time-ordered layout. Use secure random bytes for secret material rather than assuming deterministic or timestamp UUIDs are secret tokens.

Clock history and timestamp extraction

```
generator = NewUUIDClockGenerator()
first = NextUUIDv1(BYREF generator)
second = NextUUIDv6(BYREF generator)
PRINT UUIDGregorianTicks(second)
```

UUIDClockGenerator is opaque and noncopyable to prevent duplicating its sequence state. It is exclusively borrowed with BYREF and cleaned automatically. Resource transfer remains explicit. It keeps a random multicast node and 14-bit sequence. NextUUIDv1AtTicks/NextUUIDv6AtTicks accept explicit Gregorian ticks for controlled clocks. Repeated/backward timestamps advance a logical timestamp by one 100-ns tick; backward observations also advance the sequence modulo 16384. Exhaustion fails before changing the state. No ordinary recoverable failure partially commits an ID. The v6 generator uses RFC-permitted legacy v1 state behavior.

UUIDGregorianTicks returns exact v1/v6 ticks. UUIDClockSequence and UUIDNode inspect those layouts. UUIDUnixMilliseconds handles v1/v6/v7 and floors negative fractional milliseconds. UUIDv1ToV6/UUIDv6ToV1 preserve timestamp, sequence and node. Incompatible versions/variants fail with UuidError. Wall-clock convenience functions retain the existing signed-nanosecond clock range and IoError contract; raw constructors cover the full wire timestamp ranges. These functions are not a civil/timezone library.

Serialization and packages

The shared SERIALIZE default is canonical text, inherited by JSON. UUID record fields therefore encode/decode as strings automatically. Invalid wire UUIDs report JsonError with the enclosing field path/location; standalone ParseUUID reports UuidError. Explicit field converters and JsonCodec rules override the default. For example WithJsonFormat(codec,UUIDBytes,UUIDFromBytes) uses a byte-list representation and retains the decoder's typed UuidError.

Concrete UUID values, rich errors and UUIDClockGenerator owners/BYREF state work through source and moved .blib interfaces. No producer-local semantic type ID is stored in a UUID value. This does not make stored JsonCodec profiles portable; U6 remains a separate contract. Source, name hashes, time/generator helpers, compiler contracts.

```
python3 tools/test_uuid.py build/selfhost/stage3
```

Secure random bytes

SecureRandomBytes(count) returns independent packed BYTE data from the operating system’s cryptographic source. FillSecureRandom(BYREF bytes) replaces an existing initialized buffer with the same length. These operations are separate from PCG/RND and never fall back to a weak pseudorandom generator on failure.

Counts must satisfy the checked capacity range. Zero requests an empty result. IoError reports OS failures, including an unavailable, uninitialized or prohibited source. Allocation failure retains its separate fatal policy. Returned bytes do not form a special locked or automatically erased secure-memory type; protect their storage according to the application’s needs.

A successful result is complete. FillSecureRandom prepares replacement data before committing it, so OS failure or cancellation before commitment preserves the old buffer. Cancellation after commitment does not undo the replacement. Ordinary independent copies and deterministic cleanup still apply.

The operation cooperates with the runtime’s supported scheduling boundaries; compiled/native operations may finish before cancellation is observed at the caller. No background entropy thread or general asynchronous random stream is implied. Chapter 18 contains a checked example and distinguishes reproducible simulation randomness from secure bytes.

URLs, paths and query parameters

Implemented native BASIC URI values, builders and HTTP adapters. Parsing follows RFC 3986; IPv6 zone syntax follows RFC 6874. This is an explicit URI library, not a browser address-bar parser or an authorization policy.

Configured base plus path and query

```
base = "https://api.example.com/v1/?locale=en#results"
DIM parameters AS DICT OF STRING TO STRING = DICT()
Put(BYREF parameters, "search", "Fred & Jane")
Put(BYREF parameters, "page", "2")
url = JoinURLPath(base, "/customers/", Query=parameters)
PRINT url
' https://api.example.com/v1/customers/?locale=en&search=Fred+%26+Jane&page=2#results
```

Query is optional through overloads. A dictionary supplies one value per name; DICT OF STRING TO LIST OF STRING supplies repeated values. An ordered LIST OF URLParameter also represents repeated names, bare flags, empty names and explicit empty values. Dictionaries retain insertion order; lists retain their exact order. An empty multivalue replacement removes existing occurrences of that key.

JoinURLPath **appends** decoded path data, removes duplicate slashes at the join boundary, removes literal dot segments and preserves internal repeated slashes and the final trailing slash. A leading slash does not discard the base path. Empty path leaves it unchanged. A supplied `%`, `?` or `#` is data and is escaped; existing base escapes such as `%2F` retain their delimiter identity. Dot segments may walk above the base path; this is not a filesystem confinement/security boundary.

Base query/fragment survive. Supplied query names replace all existing occurrences of those names; unrelated entries retain relative order and replacement entries follow them. Replacement values retain duplicates. With no supplied entries the original raw query is untouched. Query updates use form encoding: spaces become `+`, literal plus becomes `%2B`. No credentials, signing or network access occurs.

STRING input returns STRING. URL input returns an independent URL. Typed URLs can be passed directly to HTTPGet/HTTPGetPlain, HTTPPost and RequestHTTP/RequestPlainHTTP. ParseHTTPURL also accepts URL for explicit streaming.

Values and components

`ParseURL(text)` / `URL(text)` return an independent opaque URL or checked UrlError. `FormatURL(value)` returns URI text; IsValidURL is a Boolean syntax probe. IsAbsoluteURL checks for a scheme. Relative references, including the empty reference, are valid; syntactic validity does not imply an HTTP destination.

Read-only public field	Meaning
Scheme	Lowercase ASCII scheme, or empty for a relative reference
HasAuthority	Presence of `//`; empty host is allowed generically, e.g. file:///
HasUserInfo / RawUserInfo	Presence of `@`, and escaped userinfo without `@`
Host / HostIsLiteral	ASCII registered name or bracket contents; flag distinguishes IPv6/IPvFuture
HasPort / RawPort	Presence of `:`, and decimal text preserving leading zeros/empty port
RawPath	Escaped path; `%2F` is not a separator
HasQuery / RawQuery	Presence of ?, and escaped text without ?
HasFragment / RawFragment	Presence of #, and escaped text without #

URLAuthority includes userinfo/brackets/port. URLHostname decodes the host, including `%25` in an IPv6 zone. URLPort returns OPTIONAL INTEGER: absent/empty is NONE, otherwise checked range 0..65535. Generic URI parsing preserves larger digit-only ports; the network accessor rejects them. URLUsername/URLPassword return checked optional decoded text, distinguishing absence from an explicit empty value. URLPath and URLFragment decode UTF-8 data. URLPathSegments splits **before** decoding, so `/a%2Fb` has one nonempty segment `a/b`. Empty boundary segments are retained. RedactedURL replaces a present password with `xxxxx` for display; FormatURL includes it.

Raw components round-trip escaped octets, including non-UTF-8 `%FF`. A decoded STRING accessor rejects invalid UTF-8. NUL is valid encoded component data (`%00`) in the generic library; the existing HTTP DecodeURLComponent rejects NUL.

All fields are read-only outside the prelude; factories enforce representation invariants. Assignment, collection storage and task arguments retain ordinary independent-value ownership. A read-only field loan prevents another alias/task from mutating its root. Values do not hold pointers, mutable registries or resources. Equality/hash/order use the stored components; different spellings are not implicitly treated as the same destination. NormalizeURL is explicit.

Creating and updating values

```
url = BuildURL("https", "api.example.com", path="v1/customers", port=8443)
url = WithURLFragment(url, "results")
SetURLQueryParameter(BYREF url, "search", "Fred & Jane")
PRINT FormatURL(url)
```

BuildURL accepts a separate scheme/host, decoded path (default `/`), optional Query and port (-1 means absent). Port values 0..65535 are accepted generically. Host may be bare IPv6 or bracketed; pass any port separately. Scoped IPv6 hosts use escaped `%25` zone syntax. Supplying userinfo/path/query/fragment as a host is rejected.

WithURLPath/Scheme/Host/Port/Fragment/Credentials return independent values. WithURLCredentials has username-only and username/password overloads. WithURLHost also accepts port=-1; changing a host drops the old port unless supplied explicitly. WithoutURLPort/Fragment/Credentials remove those components. WithURLQuery replaces the entire query, including an explicitly empty `?`; WithoutURLQuery removes `?`. MergeURLQuery performs the same key-replacement policy as JoinURLPath. These query functions accept the ordered list and both dictionary forms.

SetURLQueryParameter(BYREF url,name,value) replaces duplicates with one value; AddURLQueryParameter appends a value; RemoveURLQueryParameter removes every match and returns a count. The same operations accept a mutable parameter list. Removal retains existing query-marker presence; WithoutURLQuery removes it explicitly. Construction/decoding completes before committing URL mutation. Recoverable failure leaves the old value unchanged; normal cleanup drops partial values and task owners. Fresh owned builder inputs can be adopted through BYVAL; borrowed inputs copy once for independent results. No move is inferred from last use.

Query lookup and encoding

API	Result
URLParameters(dictionary)	Ordered independent LIST OF URLParameter
ParseURLQuery(text,form=TRUE) / URLQuery(url,form=TRUE)	Decode an ordered query list; checked percent/UTF-8 errors
FormatURLQuery(parameters,form=TRUE)	Encode a list; checked inconsistent bare-flag state
URLQueryValue(parameters-or-url,name)	OPTIONAL STRING; first value, NONE when missing
URLQueryValueOr(parameters-or-url,name,fallback)	First value or fallback; present empty is not missing
URLQueryValues(parameters-or-url,name)	Every value in order
HasURLQueryParameter(parameters-or-url,name)	BOOLEAN presence test

API	Result
EscapeURLComponent(text,form=FALSE)	Encode UTF-8 bytes, preserving only unreserved characters
EscapeURLPath(text)	Encode path data, retaining / and path-reserved characters
UnescapeURLComponent(text,form=FALSE)	Checked percent and UTF-8 decoding

URLParameter has Name AS STRING, Value AS STRING=““, HasEquals AS BOOLEAN=TRUE. Set HasEquals=FALSE for a bare flag; a nonempty Value with that flag is rejected. Empty query text produces no entries. Delimited empty fields such as `&` remain empty-name bare entries. Only `&` separates entries and the first `=` separates a name/value; semicolons are data. Case is significant. With form=FALSE, `+` is data and spaces encode as `%20`. Lookup cannot silently discard malformed values.

The existing EncodeURLComponent/DecodeURLComponent HTTP convenience APIs use the same engine with their existing StringError/HTTPError contracts and NUL policy.

Resolution, normalization and limits

ResolveURL(base,reference) implements RFC 3986 section 5.2. Base must be absolute. STRING/STRING returns text; URL/URL or URL/STRING returns URL. `/customers` replaces the base path; `customers` replaces its last segment. Empty reference inherits the base path/query but not its fragment. Empty `?` replaces a base query. Query and fragment dots are data. Encoded dot segments are not interpreted during resolution. HTTP redirects use this same resolver, followed by existing scheme, downgrade, credential-forwarding and destination checks.

NormalizeURL uppercases escapes, decodes unreserved escapes, lowercases the registered host (preserving case-sensitive IPv6 zone identifiers), removes dot segments and preserves delimiter presence. removeDefaultPort=TRUE additionally removes HTTP 80/HTTPS 443. It does not sort queries, resolve DNS or establish resource identity. Relative paths remain unambiguous after scheme/dot removal.

Parsing accepts ASCII URI syntax. Builders encode decoded UTF-8 path/query/userinfo/ fragment data. Internationalized hosts require externally supplied IDNA A-labels; there is no implicit IDNA/browser host rewriting, backslash repair, whitespace trimming or Unicode normalization. IPv6/IPvFuture and scoped IPv6 are represented; the current HTTP transport accepts unscoped IPv6 and its existing ASCII DNS policy, rejecting userinfo, empty/zero/out-of-range ports and non-HTTP schemes. Generic validity never bypasses transport restrictions or an application’s host allowlist.

UrlError fields: Code, zero-based byte Offset (-1 when inapplicable), Message. Codes: 1 syntax/character policy, 2 percent escape, 3 authority/host, 4 network port range, 5 decoded UTF-8, 6 incompatible builder/operation state. The shared/JSON default encodes URI text; wire errors become JsonError with enclosing path context. Standalone operations retain UrlError.

Algorithms scan bytes linearly, stage query updates, and retain normal automatic/ library checkpoints and cleanup. Existing prebuilt automatic-scheduling limits remain C2/S6; no additional worker or hidden shared mutation is introduced.

```
python3 tools/test_url.py build/selfhost/stage3
python3 tools/check.py related --feature network
```

HTTP and HTTPS

Implemented in BASIC, with ordinary suspending functions and owned TCP connections. The protocol follows RFC 9110 and RFC 9112.

A small server

```
router=HTTPRouter()
RouteGet(BYREF router,"/hello/{name}",Hello)
MountDirectory(BYREF router,"/static/","/var/web")
listener=ListenTCP("127.0.0.1",8080)
WHILE TRUE
    connection=Accept(listener)
    SPAWN ServeHTTP(TAKE connection,router)
WEND

FUNCTION Hello(request AS HTTPRequest) AS HTTPResponse
    RETURN TextResponse("Hello "+Get(request.Parameters,"name",""))
END FUNCTION
```

Each child owns its connection and an independent router snapshot. Scope exit cancels and joins children and closes connections. No handshake or status channel is needed.
`ServeHTTP(listener,router)` also supplies the accept/spawn loop. Serving errors are translated to HTTP responses when possible; disconnected peers simply end that connection.
`ServeHTTPChecked(connection,router,options)` exposes operational/protocol and original application errors to a supervising function. Mutable router slots explicitly admit Error, so a checked server caller must handle or declare that open family. Convenience servers recover handler failures and report concrete error names. Ordinary handler signatures and ownership remain checked statically.

`Route` accepts an explicit method; `RouteGet` and `RoutePost` are conveniences. Literal segments beat `{parameter}` segments, which beat final `{*remainder}`. Duplicate method/pattern shapes are rejected. Unknown paths return 404; wrong methods return 405 with Allow. HEAD falls back to GET and suppresses body bytes. OPTIONS is supported. Paths are percent-decoded once; malformed encodings, dot segments, NUL, backslash and encoded slashes are rejected. Query strings remain separate; `HTTPQueryValues` decodes form-style query parameters.

Buffered handlers receive `request.Body` as LIST OF BYTE and `request.Trailers`. `TextResponse`, `BytesResponse` and `RedirectResponse` construct independent values. Streaming handlers are SUBs taking `(request AS HTTPRequest, connection AS TCPConnection, BYREF exchange AS HTTPExchange)` and declaring `FAILS IoError,HTTPError,IndexError,StringError`. Use `ReadHTTPBody`, `StartHTTPResponse`, `WriteHTTPBody`, `FinishHTTPResponse`; alternatively use `WriteHTTPResponse`. Read chunks until empty before reusing a connection. `ReadHTTPBodyAll` is a bounded convenience. Chunked output is selected when no Content-Length is supplied. Explicit lengths must match actual bytes. Trailers are exposed separately from headers; application-defined trailer fields must permit trailer use. The library rejects core framing/routing/authentication fields rather than merging them into initial headers.

`UseHTTP` registers streaming middleware; writing a response ends dispatch. `exchange.Context` carries string context to subsequent middleware/handlers through `request.Context`. Buffered request bodies are read before middleware runs. `HTTPFallback` installs a streaming fallback. These are checked noncapturing function values, not escaping closures. Route configuration may carry independent string values.

HTTPS and clients

`ServeHTTPS(connection,router,tlsOptions [,httpOptions])` upgrades the same owned connection using TLSServerOptions. A listener overload supplies the accept loop. Certificates and keys use the TLS interfaces documented in networking; https-server.bas is a complete example. ALPN is restricted to http/1.1. Verification remains enabled for HTTPS clients.

```
response=HTTPGet("https://example.com/")
PRINT response.Status
PRINT DecodeUtf8(response.Body)
```

`HTTPPost` accepts STRING or LIST OF BYTE. `RequestHTTP` accepts arbitrary methods. HTTPClientOptions combines HTTPOptions and TLSClientOptions. Redirects are bounded; cross-origin redirects strip authorization/cookies, and HTTPS downgrade is forbidden by default. A redirect limit of zero disables following. `HTTPGetPlain` and `RequestPlainHTTP` reject HTTPS and allow the static linker to omit TLS entirely. Generic scheme-selecting clients necessarily retain both transports.

For explicit streaming/reuse: `ParseHTTPURL`, `ConnectHTTP`, `HTTPExchange(options)`, `StartHTTPRequest`, `WriteHTTPBody`, `FinishHTTPRequest`, `ReadHTTPResponse`, then `ReadHTTPBody` until empty. Finish/drain one exchange before starting another. Errors require abandoning that connection; do not reuse damaged framing state. The buffered client does this automatically. There is no automatic connection pool, proxy, cookie jar, decompression, HTTP/2, WebSocket upgrade or CONNECT tunnel in this slice.

Directory serving and file workers

```
listener=ListenTCP("127.0.0.1",8080)
WHILE TRUE
    connection=Accept(listener)
    SPAWN ServeDirectory(TAKE connection,"/var/web")
WEND
```

The same handler backs MountDirectory. GET/HEAD, index.html, MIME types, weak ETag, If-None-Match, Last-Modified and single byte ranges are supported. Multi-range requests are ignored; If-Range falls back to a full response. Listings are opt-in with `HTTPOptions().DirectoryListings` configured through a local options value; HTML and URL components are escaped. Hidden files are disabled by default. Linux openat2 containment rejects child symlinks and traversal, including symlinks within the root. Unsupported kernels fail explicitly; there is no unsafe path-prefix fallback. Files stream in 32 KiB chunks. Listings are bounded snapshots, not unbounded traversal.

Ordinary open/read/stat/directory/seek APIs now park their coroutine while four bounded filesystem workers execute the operation. Cancellation removes queued work immediately; running kernel operations must acknowledge completion before resources/buffers are released. A stuck kernel operation can therefore delay cancellation. File writes, flush, atomic save and other mutation APIs have not all moved to workers yet.

Limits and protocol policy

HTTPOptions defaults: idle 30 s, headers 10 s, body 30 s, writes 30 s; start line 8192 bytes, headers 32768 bytes/100 fields, body 16 MiB, chunks 65536, requests per connection 1000, redirects 5. Each phase uses its deadline; arbitrary handler CPU work is not preempted. Streaming still enforces body limits. Header values must be UTF-8. Strict CRLF and framing checks reject duplicate Content-Length, TE+CL, unsupported transfer coding, malformed chunks/trailers and premature EOF. Request header folding is rejected; response folding is normalized. Expect: 100-continue is supported. HEAD/204/304 suppress bodies. A partial response is never followed by a second status. Allocation failure remains fatal, as elsewhere in the language.

Typed JSON helpers

Typed URLs and shared parsing

URL values/builders provide ParseURL, component access, JoinURLPath(base,path,Query=parameters), repeated query values and explicit RFC reference resolution. HTTPGet/HTTPPost and request helpers accept typed URL values. ParseHTTPURL and

Host-header adaptation use that parser with the existing stricter HTTP authority/port/transport policy. Redirects now use the shared RFC resolver; encoded delimiters retain their meaning and fragment data is not sent as a target.

Regular expressions

Regex is built into the BASIL toolchain and implemented in ordinary BASIL. No import, package declaration, regex `.blib`, external regex archive, or Rust runtime is required. Compile a reusable pattern for repeated work:

```
pattern=CompileRegex("(?P<name>\p{L}+)(?P<number>\d+)?")
IF LET found=RegexFind(pattern,"🌍Ada42") THEN
    PRINT found.Text
    PRINT found.Start       ' Unicode-scalar position: 1
    PRINT found.ByteStart   ' UTF-8 byte offset: 4
END IF
PRINT RegexIsMatch("^ada$","ADA",Flags="i")
PRINT RegexReplace("(?P<word>\w+)","hello world","${word}!")
PRINT Join(RegexSplit("[,;]","a,b;c"),," | ")
```

API

Function	Successful result
`CompileRegex(pattern,Flags="",Dialect=RegexDialect.Native,Limits={})`	Validated, immutable, shareable Regex
`RegexIsMatch(pattern,text)`	BOOLEAN
`RegexFind(pattern,text)`	OPTIONAL OF RegexMatch; first leftmost match
`RegexFindAll(pattern,text)`	LIST OF RegexMatch; nonoverlapping matches
`RegexReplace(pattern,text,replacement,all=TRUE)`	Independent STRING; FALSE replaces first match
`RegexSplit(pattern,text)`	LIST OF STRING; separators omitted, empty fields retained

Every operation accepts a compiled Regex or a STRING pattern. STRING overloads also accept Flags, Dialect and Limits; Replace places these after `all`. A compiled pattern retains its dialect, flags and limits. All overloads declare `FAILS RegexError`, including compiled-pattern operations: a checked work or output limit can fail during a search. Catch failures locally or declare them in the enclosing function's contract; ordinary calls already perform checked error routing. An unhandled failure at the application entry reports and exits.

`RegexMatch` contains Start/Finish (half-open, zero-based Unicode-scalar offsets), ByteStart/ByteFinish (UTF-8 byte offsets), Text, Groups and Named. Groups is LIST OF OPTIONAL OF STRING and includes group0 for the whole match. Named is DICT OF STRING TO OPTIONAL OF STRING. An unmatched capture is NONE; a matched empty capture is SOME(“”). Results own independent strings and collections and remain valid after the subject or pattern is destroyed.

Dialects

The default `RegexDialect.Native` follows the pinned Rust regex1.13.1 syntax and matching baseline with Unicode16 data: alternation, greedy/lazy/counted repeats, captures, named captures, Unicode properties and simple folding, nested class set operations and inline flags. Flags include i/m/s/u/x/U/R and scoped disabling. Lookaround and backreferences reject. Byte-oriented constructs that could match invalid UTF-8 reject; this is a STRING API, not Rust's arbitrary-byte API.

`RegexDialect.EcmaScript` selects the **ECMAScript2025 Unicode-mode pattern profile with Unicode16 properties**. It supports backreferences and positive or negative lookahead/lookbehind, Unicode property escapes, surrogate-pair escapes, named captures, and disjoint-alternative duplicate names. Matching is stateless; Flags supports i/m/s/u, with Unicode mode always active. Flags d/g/v/y reject explicitly. JavaScript RegExp object state, v-mode string sets and isolated UTF-16 surrogates are outside the profile. BASIL subjects are valid UTF-8.

```
PRINT RegexIsMatch("^([a-z]+)-\1$","blue-blue",Dialect=RegexDialect.EcmaScript)
PRINT RegexIsMatch("^(?!admin$)[a-z]+$","user",Dialect=RegexDialect.EcmaScript)
```

Dialect selects pattern grammar/matching. Both dialects use BASIL's Rust-style replacement expansion: `$1`, `$name`, `${name}`, `$$`; missing/unmatched groups expand to empty text. Use braces to separate a name from following identifier characters. Split omits capture separators. These functions do not implement JavaScript String.replace/split conventions. Empty-match iteration advances on Unicode-scalar boundaries and suppresses an empty match immediately following a nonempty match at the same ending position. Normalization and locale collation are separate facilities.

Limits and typed failures

RegexLimits field	Default
MaxWork	100,000,000
MaxCompileWork	10,000,000
MaxMemoryBytes	67,108,864
MaxPatternBytes	1,048,576
MaxInstructions	262,144
MaxDepth	256
MaxResults	1,000,000

All limits must be positive. Work is logical engine work, not milliseconds. Memory accounting conservatively covers estimated engine-owned program, scratch and output storage, including temporary construction; it is not a process RSS ceiling. Immutable shared Unicode tables are toolchain data. Compile work accumulates through the parser/compiler; search work resets once per public operation, including all candidate starts, matches and assertions. An optional DFA optimization stops within its own bounded quota and falls back to the complete NFA. Allocation failure retains BASIL's fatal policy.

RegexError carries Code, Message and Offset (a UTF-8 pattern byte offset, or −1 when unavailable). RegexSyntaxError identifies invalid syntax; RegexUnsupportedError identifies unsupported profile features (Code3). RegexLimitError has Code2, Limit AS RegexLimitKind and Maximum. The enum contains Work, Memory, Pattern, Instructions, Depth and Results. Invalid configuration uses the RegexError base with Code4; internal invariant diagnostics use Code5. A limit failure never becomes FALSE, NONE, or a partial successful result.

```
TRY
    limits=RegexLimits {MaxWork=10000}
    PRINT RegexIsMatch("^(a+)+
\1b$",Repeat("a",100),Dialect=RegexDialect.EcmaScript,Limits=limits)
CATCH exhausted AS RegexLimitError
    PRINT exhausted.Message
END TRY
```

Ownership and scheduling

Regex is an opaque resource. Ordinary parameters borrow it; explicit TAKE transfers a named owner. Cleanup releases it automatically. Matches and captures own independent data. Compilation and matching participate in normal scheduling and cancellation, including calls through compiled BASIL libraries. No background thread or application polling protocol is required. C-ABI callbacks retain the usual nonsuspending foreign-call rules.

Native BASIC compression

The first-party codecs are ordinary BASIC modules under stdlib/compress. Import only the format needed; unused code is removed by the ordinary linker. There is no zlib/libbz2/Go dependency in the production implementation. Go and Python are independent acceptance oracles.

The algorithm scope follows Go's compress packages:

Module	Encoding	Decoding	Format
flate.bas	Yes	Yes	RFC 1951 DEFLATE
zlib.bas	Yes	Yes	RFC 1950, including Adler-32
gzip.bas	Yes	Yes	RFC 1952, header/body checksums and sizes
lzw.bas	Yes	Yes	Late-change LZW; 2–8 literal bits, 12-bit dictionary, LSB/MSB
bzip2.bas	No	Yes	Nonrandomized bzip2 blocks and concatenated streams

These are buffer APIs, not a claim of Go's complete streaming Reader/Writer API. Preset DEFLATE/zlib dictionaries and reusable streaming state are not yet exposed. TIFF early-change LZW and deprecated randomized bzip2 are rejected; Go also rejects the latter. No bzip2 encoder is implied.

API

```
IMPORT "../stdlib/compress/gzip.bas" AS gzip
IMPORT "../stdlib/compress/common.bas" AS compression

original=EncodeUtf8("Hello, BASIC!")
packed=gzip.Compress(original)
restored=gzip.Decompress(packed,MaxOutput=1024)
PRINT DecodeUtf8(restored)
```

Paths above are relative to an example source in `examples/`. All data arguments and results are `LIST OF BYTE`, preserving arbitrary bytes without UTF-8 coercion. Inputs are borrowed for the call. Results are independent owners; failures leave inputs unchanged and clean temporary/partial owners normally.

`Compress(data, Level=1)` applies to flate/zlib/gzip. Levels: 0 stored blocks, 1 fast bounded matching, 2–9 progressively longer bounded hash-chain search, −1 default search, −2 literal-only Huffman encoding. Compression levels are effort choices, not a promise of identical Go output or compression ratio. The encoder currently emits fixed Huffman codes or stored blocks; the decoder accepts fixed, dynamic and stored blocks. Level 1 uses four-byte hashing, sparse match-position updates, and checked word comparisons. Higher levels retain bounded chains. Incompressible output falls back to stored blocks.

`Decompress(data, MaxOutput=67108864)` applies to flate/zlib/gzip/bzip2. The limit counts actual output bytes, including all concatenated members. Zero permits empty output. Negative limits and limits above 1 GiB fail. Size checks precede growth; advertised sizes do not grant unbounded allocation. Malformed headers, invalid Huffman trees/history references, truncation, checksum/size mismatches and trailing garbage fail through `compression.CompressionError` with `Format`, `Offset`, `Message`. Some shared failures name `bits` or `huffman`; no native pointer or unchecked success escapes. gzip metadata decoding also declares `StringError` for its Latin-1-to-UTF-8 conversion operations.

`lzw.Compress(data, Order=lzw.BitOrder.LSB, LiteralWidth=8)` and `lzw.Decompress(data, Order=..., LiteralWidth=..., MaxOutput=...)` select the packing convention. The decoder requires the terminal code and rejects trailing bytes. Encoder input bytes must fit the specified literal width. Dictionary resets, width transitions and the special repeated-prefix code are checked.

`gzip.Compress` additionally accepts `Metadata=SOME(gzip.Header {...})`. Header fields are Name, Comment, Extra (byte list), Modified (unsigned seconds in an INTEGER), OS (0–255). Name/comment use Latin-1 without NUL; extra <=65535 bytes. Default output is deterministic: no timestamp, name or comment, OS 255. `gzip.ReadMember(data, start=0, MaxOutput=..., MaxHeader=65536)` returns `Member {Header, Bytes, EndOffset}` for explicitly processing one member. Normal `Decompress` concatenates all members. Optional extra/name/comment/header-CRC fields are validated and header bytes are bounded separately from output. For a valid header exceeding the default bound, use ReadMember with a larger explicit MaxHeader; the normal multi-member convenience keeps its 64 KiB bound.

`flate.Decode(data, start=0, MaxOutput=...)` returns `compression.Decoded {Bytes, EndOffset}` for implementing enclosing formats. `flate.Decompress` requires the entire input to contain exactly one raw stream. Offsets are byte positions at or just after the failing input operation, not bit-precise locations.

Runnable buffer showcase exercises all four encoders with UTF-8 and arbitrary binary bytes, gzip metadata and both LZW orders. Bzip2 decoder interoperability is covered by the acceptance corpus.

Coordinate and vector arithmetic

These operations borrow their inputs and return independent results. Chapter 21 contains runnable examples. Ordinary INTEGER division and MOD truncate toward zero; choose the explicit coordinate helpers for other mathematical policies.

Coordinate helper	Result and checked errors
FloorDiv(value, divisor)	INTEGER floor quotient; NumericError for zero or unrepresentable quotient
EuclideanMod(value, divisor)	Nonnegative INTEGER remainder below mathematical abs(divisor); NumericError for zero divisor
FloorToInteger(value)	DOUBLE floor converted to INTEGER; NumericError for nonfinite/out-of-range input

For example FloorDiv(17,-16) is −2, while EuclideanMod(17,-16) is 1. With a negative divisor, these policies do not necessarily describe the same quotient/remainder pair. Minimum INTEGER divided by −1 is unrepresentable; its Euclidean remainder is zero. These helpers report recoverable NumericError rather than relying on fatal arithmetic traps.

Numeric sequences

Supported element types are DOUBLE, INTEGER, BYTE, UINT32 and UINT64. Operands are lists or rank-one arrays. Paired operands use the same element/container type; arrays with different lower bounds pair by position. Other ranks raise VectorRankError. There are no string or Boolean coercions.

API	Result	Checked conditions
Dot(left, right)	DOUBLE, INTEGER, or UINT64 for unsigned inputs	Matching dimensions; integral arithmetic range
Norm(values)	DOUBLE Euclidean length	Valid array rank
Normalized(values)	LIST OF DOUBLE	Nonzero norm
Distance(left, right)	DOUBLE Euclidean distance	Matching dimensions
CosineSimilarity(left, right)	DOUBLE, rounded/clamped to [−1,1]	Matching dimensions and nonzero norms
VectorAdd/Subtract/Multiply/Divide	LIST with input element type	Matching dimensions; integral range/division
Scaled(values, factor)	LIST with input element type	Factor has the same scalar type; integral range

VectorDimensionError carries LeftCount and RightCount. VectorRankError carries Rank. Empty dot, norm and distance are zero; elementwise empty results are empty. Normalized and cosine reject zero norms, including empty vectors, with ZeroVectorError. Shape validation precedes arithmetic. Recoverable failure preserves source values and an existing assignment destination.

BYTE/UINT32 dot products widen to UINT64 before multiplication. Other component operations keep their input width. Integral overflow/underflow or unrepresentable quotients raise NumericError code 2; integer division by zero is code 1. An earlier intermediate overflow fails even if later terms could cancel it.

Floating arithmetic retains IEEE NaN/infinity. DOUBLE dot starts at +0.0 and uses source-order multiplication/addition without reassociation or fused contraction. Integral norm/normalization/cosine coordinates convert to DOUBLE and can lose precision above 2^53. Integral distance first computes each exact difference magnitude, then converts it: adjacent large integers still have distance 1.

Norm/distance return NaN for a relevant NaN component/difference, otherwise infinity when any is infinite. Equal infinities have NaN distance. Normalization with infinite scale yields NaN for

infinite components and signed zero for finite ones; NaN scale yields NaN components. Nonfinite cosine data yields NaN, but a zero vector still raises ZeroVectorError first. Use explicit finite validation where the application needs it. Value-returning operations produce new lists even for array inputs; use ToArray when an array is required. Large synchronous operations may delay scheduling and cancellation.

Unsigned numbers and bit operations

BYTE, UINT32 and UINT64 are distinct static types with ranges 0..255, 0..4,294,967,295 and 0..18,446,744,073,709,551,615. Packed lists use one, four and eight bytes per element respectively. Choose packed BYTE data for binary I/O; spare reserved capacity is never initialized content.

Hexadecimal `&H`, binary `&B` and octal `&O` literals have UINT64 type. Decimal whole literals retain INTEGER type. There are no implicit mixed numeric conversions. BYTE/UINT32/UINT64 constructors check range with the ordinary fatal policy. CBYTE/CUINT32/CUINT64 supply supported recoverable INTEGER/UINT64 conversions through NumericError. CINT(UINT64) checks signed range; CINT of BYTE/UINT32 is infallible. CDBL converts to binary64 with its normal rounding.

Operations	Behavior
BitAnd, BitOr, BitXor, BitNot	UINT64 bitwise operations
ShiftLeft, ShiftRight	Shift UINT64 bits; right shift fills with zeros
RotateLeft, RotateRight	Rotate within a UINT64 word
LeadingZeros, TrailingZeros, BitCount	INTEGER counts; zero has 64 leading/trailing zeros
WrappingAdd/Subtract/Multiply	Explicit arithmetic modulo 2^64
MultiplyHigh	High half of a UINT64 multiplication
MultiplyWide	WideProduct with High and Low UINT64 fields

Use matching UINT64 arguments for these named helpers, constructing them explicitly from narrower values when needed. Infix `&`, `|` and XOR also accept matching INTEGER/BYTE/UINT32/UINT64 operands directly. MultiplyWide's record result is one successful value; it does not imply multiple-result syntax.

Shift/rotate counts are INTEGER in 0..63; invalid counts are fatal. Left shifts discard bits outside the word. UInt64Bits/UInt64FromBits reinterpret numerical bits between INTEGER and UINT64 without exposing addresses. UInt64Max returns the maximum unsigned value. Ordinary unsigned arithmetic checks overflow and division in its declared width; unary minus is rejected.

Lists, sets, dictionaries, optionals, records, arrays, sorting, hashing and formatting support unsigned scalars under their usual rules. Copies remain independent, resources keep their ownership restrictions, and an indexed BYREF or SWAP keeps its containing owner protected. Packed storage does not change those guarantees.

Floating point and math

Implemented by the primary BASIC compiler; the frozen Rust seed remains unchanged. `DOUBLE` is IEEE-754 binary64. Decimal literals include `1.5`, `.5`, `1.`, and `6.02e23`. An exponent makes a literal floating point even without a decimal point. Decimal conversion rounds to nearest, ties to even, including subnormals. A literal that overflows is rejected; underflow can round to signed zero. `SINGLE` and decimal/fixed-point types are not implemented.

```
DIM measurements = [1.5, 2.5, 3.5]
DIM total = 0.0
FOR EACH value IN measurements
    total = total + value
NEXT
PRINT total / CDBL(LEN(measurements))

PRINT SQRT(2.0)
PRINT POW(2.0, 10.0)
PRINT IsNaN(SQRT(-1.0)) ' TRUE; math itself does not raise an error.
TRY
    PRINT Finite(EXP(1000.0))
CATCH problem AS NonFiniteError
    PRINT "Expected a finite value: ";problem.Value
END TRY
```

There are no implicit INTEGER/DOUBLE conversions, including in collections and overload resolution. `CDBL(integer)` converts with nearest-even rounding. `CDBL(double)` preserves the input bits. `TRY CINT(double)` truncates toward zero and reports `NumericError { Code=2 }` for nonfinite or out-of-range values. `TRY CDBL(text)` / `TRY VALDOUBLE(text)` trim Unicode whitespace and parse the entire ASCII numeric spelling; malformed text reports NumericError code 1, finite overflow code 2. Signed `inf`, `infinity`, and `nan` are accepted. `STR(double)` and PRINT use a shortest round-tripping decimal spelling. Negative zero prints `-0.0`.

Arithmetic `+ - * /`, unary signs, and comparisons accept matching DOUBLE operands. Arithmetic follows IEEE behavior, including infinities, NaNs, gradual underflow, and signed zero. Division by floating zero does not become an integer arithmetic trap. NaN compares unequal to every value, including itself; all ordered comparisons involving NaN are false. `MOD` remains an integer operation; use FMOD or REMAINDER for doubles. `POW` provides exponentiation; a `^` operator is not implemented.

Defaults can contain DOUBLE literals and constant arithmetic/comparisons. Floating division by zero in a default has the same IEEE result as runtime arithmetic. Mixed numeric types are rejected in defaults as elsewhere.

Functions

Names are case-insensitive. Unless noted, arguments and numeric results are DOUBLE. Floating math functions are infallible: numeric domain failures return NaN; overflow returns signed infinity; underflow rounds normally, possibly to signed zero. There is no mandatory result-validation pass. Algorithms retain the input classification needed for correct special cases and safe range reduction. Allocation failure is fatal and separate.

Use `Finite(value AS DOUBLE) AS DOUBLE FAILS NonFiniteError` to opt into finite-result validation. It returns a finite input bit-for-bit, including signed zeros/subnormals, or returns `NonFiniteError {Value AS DOUBLE}` with the rejected bits. It uses ordinary CATCH/FAILS propagation and automatic cleanup, including in tasks. IsFinite is the non-failing predicate. Finite checks only its supplied result: finite underflow and 1.0/Infinity() pass. No hardware trap modes are changed.

Family	Functions	Contract
Constants/classification	PI(), TAU(), E(), Infinity(), NaN(), IsNaN(x), IsInfinite(x), IsFinite(x)	Infallible; IsInfinite accepts either infinity sign
Explicit validation	Finite(x)	NonFiniteError for NaN or either infinity
Magnitude/sign	ABS(x), COPYSIGN(x,sign)	Infallible
Sign as INTEGER	SGN(x), SIGN(x)	DOUBLE NaN fails MathError code 1; INTEGER overloads infallible
INTEGER magnitude	ABS(integer)	NumericError code 2 for minimum INTEGER
Rounding, DOUBLE result	TRUNC/FIX, FLOOR/INT, CEIL/CEILING, ROUND(x)	Infallible; ROUND ties to even
Decimal-place rounding	ROUND(x,places AS INTEGER)	Infallible; exact decimal quantization of binary input, ties to even; overflow to signed infinity
Bounds	MIN(a,b), MAX(a,b), CLAMP(x,lower,upper), also INTEGER overloads	DOUBLE CLAMP returns NaN for reversed/NaN bounds; INTEGER CLAMP fails MathError code 1 for reversed bounds
Roots/power	SQR/SQRT(x), POW(base,exponent), EXP2(x), HYPOT(x,y), CBRT(x)	Infallible
Exponential/log	EXP, EXPM1, LOG/LN, LOG(x,base), LOG2, LOG10, LOG1P	Infallible
Trigonometric, radians	SIN, COS, TAN, ASIN, ACOS, ATAN/ATN, ATAN2(y,x)	Infallible
Hyperbolic	SINH, COSH, ACOSH, ATANH, TANH, ASINH	Infallible
Angle conversion	DEGREES, RADIANS	Infallible
Fused operation	FMA(x,y,z)	Infallible; one final binary64 rounding for finite inputs; invalid infinite operations return NaN
Remainders	FMOD(x,y), REMAINDER(x,y)	Infallible; truncating versus nearest-even quotient
Binary scaling	LDEXP(x,exponent AS INTEGER)	Infallible; overflow to signed infinity
Binary decomposition	FREXP(x) returns FloatParts { Mantissa AS DOUBLE, Exponent AS INTEGER }	Infallible
Adjacent values	NEXTAFTER(x,toward), ULP(x)	Infallible
Bit serialization	DoubleBits(x) returns INTEGER; DoubleFromBits(integer)	Infallible; numeric encodings, never addresses

Important special cases:

Operation	Result
SQR/SQRT(negative), LOG/LN/LOG2/LOG10(negative)	NaN
LOG/LN/LOG2/LOG10(+0 or −0), LOG1P(-1)	Negative infinity
LOG1P(x < −1), LOG(x,base <= 0 or base = 1)	NaN
ASIN/ACOS outside [−1,1], ACOSH below 1	NaN
ATANH(+1), ATANH(-1), ATANH outside [−1,1]	Positive infinity, negative infinity, NaN respectively
Finite negative POW base with nonintegral exponent	NaN
POW(+0,negative), POW(-0,negative odd integer)	Positive infinity, negative infinity respectively
POW(x,0), POW(1,x), including NaN arguments	1.0

Operation	Result
SIN/COS/TAN(infinity)	NaN
FMOD/REMAINDER with zero divisor or infinite dividend	NaN
HYPOT(infinity,NaN)	Positive infinity

NaNs otherwise propagate; no payload/sign is promised for computed NaNs. Explicit bit conversion and successful/rejected Finite values preserve their input bits. Legacy inline TRY is accepted only for fallible calls: remove it from calls to the now-infallible floating math functions. TRY blocks remain useful around Finite and numeric conversions. MathError now covers INTEGER-result sign and INTEGER bound errors; it no longer reports floating numeric domain or overflow results.

MIN/MAX propagate NaNs and select the appropriate signed zero. CBRT handles negative numbers. Trigonometric reduction covers the full finite binary64 range. No application cleanup is promised after fatal allocation failure.

Collections

DOUBLE works in records, optionals and collections. Scalar and recursive equality follow IEEE rules: NaN is unequal even inside a value compared with itself. Compare returns CompareResult.Unordered; default stable sorting uses separate NaNs-last ordering. NaN set members/dictionary keys are rejected through checked InvalidKeyError, including composite keys; list elements and dictionary values may contain NaN. Infinity is valid and signed zeros are equal with equal hashes. Comparison, sorting, key and error contracts.

BASIC terminal, file and process services

APIs

- `OpenTerminal`, `EnterTerminalScreen`, `GetTerminalSize`, `WriteTerminal`, `ReadTerminalEvent`, `TerminalCursor`, `TerminalColor`, `ShowTerminalCursor`, `SuspendTerminal`, `ResumeTerminal`, `Close`.
- `ReadTerminalEvent(terminal, duration)` returns `OPTIONAL OF TerminalEvent`: NONE is timeout; SOME with Kind 0 is EOF. Text, keys, resize and paste-boundary events have distinct kinds. Escape ambiguity has a bounded decoding timeout.
- `TerminalTextWidth` and `TerminalClipText` use checked cell ranges and pinned Unicode widths. They are scalar-width facilities, not a grapheme shaping engine.
- `Stdin()`, `Stdout()`, `Stderr()`, `OpenFile(path, mode="r")`, `Close`, `Flush`, `ReadFileInto`, `WriteFile`, `GetFileInfo`, `ReadFileSnapshot`, `WriteFileAtomic`, `ReadDirectory`, `MakeDirectory`, `SetFilePermissions`.
- `StartProcess(executable, arguments, directory=".", interactive=FALSE)`, `RunProcess`, `RunInteractive`, `ProcessExited`, `WaitProcess`, `TerminateProcess`, `WriteProcess`, `CloseProcessInput`, `ReadProcessInto`, `TryReadProcessInto`.
- `OpenPseudoTerminal`, `PseudoTerminalName`, `TryReadPseudoTerminalInto`, `WritePseudoTerminal` support debugger inferior terminal I/O.

Standard streams as File values

`Stdin()`, `Stdout()` and `Stderr()` each return a fresh owned `File`, with `FAILS IoError`. They duplicate the currently inherited stream handle, preserving redirection to terminals, pipes or files. Closing or destroying that File closes only its duplicate; the original process stream remains available to PRINT, INPUT, ReadByte, WriteAll and WriteErrorText. Named handles require explicit TAKE when transferred into a buffer, task or another owner. Fresh results can be passed directly, for example `BufferedWriter(Stdout())`.

These are independent **handle owners**, not independent byte streams: duplicates share the underlying stream position, EOF and file status flags. Use one buffered reader per input stream; read-ahead can otherwise consume bytes another reader expects. Separate writers do not provide implicit synchronization or message atomicity. The factories do not change blocking flags. Read-only borrowing of a File does not freeze this external OS state.

```
writer = BufferedWriter(Stdout())
WriteLine(BYREF writer, "Hello, λ")
Flush(BYREF writer)
WriteFile(Stderr(), "diagnostic" + CHR(10))
```

Buffer cleanup discards pending output. Call `Flush(BYREF writer)` or `Finish(TAKE writer)` to send it and observe write failures. That buffer Flush does not call fsync; `Flush(file)` does and may fail on a pipe or terminal. ReadFileInto and buffered reads use the existing bounded File workers. A blocking pipe read can require peer data/EOF before cancellation can acknowledge the kernel operation; cleanup retains its storage and handle until then. File writes remain synchronous.

Processes, scheduling and failures

`RunProcess` accepts exact UTF-8 argv elements and never inserts a shell. It pumps input, stdout and stderr together, preventing the usual full-pipe deadlock. Outputs are byte lists; UTF-8 decoding is explicit. Nonblocking reads return NONE for not-ready and SOME(0) for EOF. Captured output is currently unbounded growable storage; application limits/streaming are follow-up work. Environment inheritance reads the process's initial Linux environment; there is no BASIC environment mutation API yet.

Terminal input/output, process pipes, pidfd completion and resize self-pipe waits park the current coroutine. The sole scheduler thread owns wait registrations. Events and timers only mark/wake

tasks; BASIC resumes after polling returns. Readiness callbacks never re-enter BASIC application code inside the native poll loop. Multiple waits detach deterministically on completion, timeout and cancellation. SQLite's bounded worker completion eventfd participates in the same idle wait (native component ABI 2), so terminal/pipe I/O cannot hide completed database work.

Regular file open/read/stat/directory/seek use a bounded worker pool and acknowledge cancellation before releasing retained buffers/handles. Writes/fsync and some mutations remain synchronous and can block the coordinator. This implementation does not claim that all filesystem work is asynchronous. See worker integration.

Recoverable syscall failures use `IoError` with positive Linux errno. File version conflicts use `FileChanged`. Allocation failure is fatal with a best-effort diagnostic, with no normal application cleanup guarantee. Native terminal cleanup restores original descriptor flags, termios and resize signal action, closes its self-pipe, and leaves the alternate screen. Abnormal external termination can still prevent application cleanup.

Commands, projects and libraries

Verified scope: Linux x86-64, local source projects and prebuilt BASIC libraries. Tooling requirements also describe future remote resolution/locking. Package format and ABI specify this release.

Standard-library declaration catalog: `basic api` works without a manifest or native SDK and supplies IDE discovery from parsed declarations.

Build and run

```
build/selfhost/stage3 --help
build/selfhost/stage3 help build
build/selfhost/stage3 api
build/selfhost/stage3 build hello.bas -o build/hello
build/selfhost/stage3 run --manifest examples/space-invaders/basic.toml -- --help
```

`build` without a source reads `basic.toml` in the working directory. `--manifest` selects another manifest; relative paths inside it resolve from its own directory. Default outputs are `<project>/build/<name>` or `<name>.blib` for libraries. An explicit `-o/--output` resolves from the invoking directory. `run` accepts only executables, inherits terminal input/output, waits and returns the child's exit status. Arguments after `--` go to the program, including `--help`.

Help is generated from the same public command declaration as parsing and works before any manifest/source access. Usage errors print relevant usage to stderr and exit 2; build/data errors exit 1. Help/version go to stdout. The compiler version is `0.1.0`; package, manifest and ABI versions are separate.

Compatibility remains: `[--emit-object|--emit-library|--debug|--debug-lines] source.bas > binary`, and no arguments compiles stdin to stdout. A legacy `.bas` invocation emits bytes; `build` writes the output file and prints its path. The frozen compact checkpoint builds the maintained core; that core builds the public front end. Stage 2 and stage 3 compare the same public front end.

Manifest

```
manifest-version = 1
[project]
name = "my-app"
version = "0.1.0"
[toolchain]
compiler = ">=0.1.0, <0.2.0"
[target]
kind = "executable" # or "library"
root = "src/main.bas"
[dependencies]
Raylib = {path = "../raylib/build/raylib.blib", version = "0.1.0"}
```

`IMPORT Raylib` resolves the declaring project's dependency alias. Alternatively, `path` names a source project directory containing its own `basic.toml`. Quoted source imports retain their existing relative-file behavior. Source projects compile from current sources each build; no stale binary cache replaces edits. Dependency aliases are ASCII BASIC identifiers, compared without case. Unresolved aliases, wrong target kinds, ambiguous aliases and conflicting builds of the same package name are diagnosed. Optional dependency versions assert an exact package version. Compiler constraints accept exact `MAJOR.MINOR.PATCH` or comma-separated `=`, `<`, `<=`, `>`, `>=` comparisons, checked transitively.

The manifest reader implements a **project TOML subset**, not the planned general TOML library: strings/escapes, integers, Booleans, arrays, inline tables, dotted keys, headers and comments. Arrays of tables, multiline strings, date/time and floating scalar literals are rejected. Unknown

schema fields are errors; other platforms' native tables are ignored. Manifest limit: 2 MiB. Source/import namespace rules remain those of the language.
Library manifests may declare:

```
[project]
name = "wrapper"
version = "0.1.0"
licenses = ["LICENSE.txt"]
[target]
kind = "library"
root = "src/wrapper.bas"
[native.linux-x86_64]
archives = ["vendor/libwrapper.a", "vendor/libupstream.a"]
system-libraries = ["SDL2", "GL", "m", "dl", "pthread"]
```

Declared/referenced archives and notices travel inside `.blib`. Native identity currently means archive basename; equal hashes merge, unequal bytes under one identity fail. This deliberately rejects ambiguous builds rather than picking one. Prebuilt transitive dependencies need no original paths or source projects. BASIC runtime state/helpers link once; archive members are selected by references. Generic templates remain checked source input for consumer specialization. Nongeneric bodies are precompiled, with checked import stubs in their interfaces. Current ABI v15 retains v14 suspension/cancellation effects, supported resource cleanup hooks, checked callables and compatible profile contracts. Native resource owners and owning containers retain source-level borrow/transfer rules across compiled libraries; see compiled resource interfaces. Scope-bound task/channel capabilities retain their source restrictions: channel owners cannot escape, and task handles have no concrete public signature spelling. Foreign C interfaces and anonymous CALLBACK/NOSUSPEND adaptation remain separate restricted boundaries.

The GTK 4 .blib and demo use this path with declared GTK/GLib system dependencies and a bundled static adapter/readiness archive.

Extracted verified interfaces/archive blobs live under the invoking project's `build/.basic/packages/<id>/`. This is build output, not a remote download cache. Git/registry dependencies, lockfiles and shared incremental compilation caches remain future work. Put mutually dependent library modules in one project; cycles between independently packaged libraries are currently rejected. Library imports must be relative within their project; cross-project imports use the declared bare alias. Absolute/cross-project quoted imports are rejected when packaging rather than persisting producer-machine paths.

Public argument API

`Command` declares `Options`, `Positionals` and child `Commands`. `CommandOption` has `Name`, optional one-character `Short`, `Description`, `Kind`, `Required`, optional typed `Default`, and `Global`. `CommandPositional` has name, description, kind and required status. `ArgumentKind` is Text/IntegerValue/RealValue/BooleanValue. Use `CommandFlag` for a Boolean with default FALSE. Default constructors are `ArgumentText`, `ArgumentInteger`, `ArgumentReal`, `ArgumentBoolean`.

`ParseArguments(command)` reads process arguments and skips the executable. The explicit `ParseArguments(command, arguments)` overload remains pure and expects the executable to have already been excluded. `CommandLineParameters()` returns that independent argument list; raw `CommandLineArguments()` still includes index zero. Process access declares IoError; explicit parsing does not.

`TextOption`, `IntegerOption`, `RealOption`, `BooleanOption` supply ordinary typed defaults without boxing at call sites. For example, `IntegerOption("frames", Default=0, Min=0, Max=360000, Help="Frame limit")`. Integer bounds are inclusive and validated for defaults and supplied values. `TextArgument`, `IntegerArgument`, `RealArgument`, `BooleanArgument` read the Options/Positionals dictionaries and raise ArgumentError for missing/mismatched values; they never silently turn a typo into a default or FALSE.

The parse result is `ArgumentResult.Parsed` or `.Help`; invalid input raises `ArgumentError {Message, CommandPath, Usage}` (and checked StringError). `ParsedArguments` holds CommandPath, Options/Positionals and Forwarded with ordinary independent-value semantics. `RenderHelp` remains pure. `PrintArgumentHelp(help)` prints and returns 0; `ReportArgumentError(error)` prints to stderr and returns 2. Neither exits the process; the caller decides whether to return, continue, or perform additional cleanup.

For ordinary applications, declare a record instead of a command tree:

```
TYPE Options FORMAT CommandSettings {Name="demo",Description="A typed command"}
    Frames AS INTEGER=0 FORMAT OptionSettings {Min=0,Max=360000,Help="Frame limit"}
    Hidden AS BOOLEAN=FALSE
    Output AS OPTIONAL OF STRING=NONE FORMAT OptionSettings {Name="output",Short="o"}
END TYPE
TRY
    SELECT TYPE ParseArguments(Options {})
    CASE ArgumentResult.Help AS help
        RETURN PrintArgumentHelp(help)
    CASE ArgumentResult.Parsed AS parsed
        options=ArgumentOptions(parsed) AS Options
        PRINT options.Frames
    END SELECT
CATCH usage AS ArgumentError
    RETURN ReportArgumentError(usage)
END TRY
```

Defaults are ordinary record values; the supplied prototype must be initialized. Field types currently supported: STRING, INTEGER, DOUBLE, BOOLEAN and their OPTIONAL forms. Absent optional flags produce NONE; explicit empty text remains SOME(“”). Options default to the lowercase physical field name. OptionSettings supports Name, Short, Help, Required and inclusive integer Min/Max. The TYPE header requires CommandSettings; format options retain ordinary nominal module identity. Shared wire names are respected internally; read/write-excluded fields and unsupported nested/resource fields are rejected. Custom converting serialization contracts are not a supported CLI type-conversion mechanism.

`ArgumentOptions(parsed) AS T` constructs a complete independent owner through public checked serialization adapters. Dynamic command trees may also use it for a selected typed options record (as the compiler does); missing required/nondefaulted fields and wrong scalar types remain errors. Help is still a normal result, so no fabricated default record is exposed as a successful parse. SELECT TYPE checks coverage and binds the selected fresh result without copying its payload. Subcommands/positionals/forwarding remain the dynamic Command API; the typed record convenience declares options only.

Measured project-build optimizations preserve full checks on every invocation; persistent incremental/prelude caching remains planned.

Compiler and SDK identity

`basic toolchain` prints JSON identifying the actual running compiler/hash, target, BASIC ABI and resolved native SDK root. It works before loading a project and can help diagnose an incomplete SDK. Incompatible .blib diagnostics show producer and consumer schema/target/ABI plus the command to rebuild the producer from source. A .blib file is not a substitute for the source needed by that rebuild command.

Build-time embedded values

`EMBED` reads a UTF-8 file while building and makes a readonly module value. The executable needs neither the input file nor the decoder. PUBLIC/PRIVATE work as for other module declarations. Implementation reference.

```
TYPE ServerConfig
    Host AS STRING
    Port AS INTEGER = 8080
END TYPE

EMBED Defaults FROM "config/defaults.json" AS ServerConfig USING Json
EMBED DefaultText FROM "config/defaults.json" AS STRING
EMBED Port FROM "config/port.txt" AS INTEGER USING ReadPort

PRINT Defaults.Host; ":"; Defaults.Port
local = Defaults
local.Port = 9000
PRINT Defaults.Port       ' Still the original value.

FUNCTION ReadPort(text AS STRING) AS INTEGER FAILS NumericError
    RETURN VAL(TRIM(text))
END FUNCTION
```

Paths are relative to the declaring source, including imported modules. A path must stay inside its declared project tree, or its source directory for a standalone module; absolute paths and escaping symlinks are rejected. Inputs must be stable regular files. STRING preserves UTF-8 bytes, NUL, a BOM and line endings exactly. A decoder applies its own text/format policy.

`USING` names an ordinary visible function. Its first argument receives STRING content; trailing arguments use their checked defaults. The AS type supplies ordinary result context, including inferred generic result types. The same function works at runtime. `Json` selects `DecodeJson` when no visible function named Json exists. Other user decoders need no registration or compiler plugin. YAML/TOML/XML codecs are separate pending work.

All declarations are checked, including unused inputs. Decoders may read other embedded values; dependencies are ordered before execution and cycles are errors. Application top-level statements and Main do not run during decoding.

The declaration and its nested data cannot be assigned, transferred or passed BYREF. Ordinary readonly calls borrow it. Assignment/owned return/BYVAL use the normal independent-value rules; changing a copy cannot change the embedded value. Immutable STRING bytes can safely remain shared. Records, lists, sets, dictionaries, dense arrays, optionals, choices and enums are supported recursively. Resources, pointers, function values, task/channel handles and errors as successful data are rejected. A decoder can still fail with its declared typed errors.

Raw STRING embeddings also work in ordinary constant/default contexts. A value produced by USING is a readonly runtime value; it does not become a general constant-expression factory. Exported `.blib` values preserve these rules after moving the package and deleting producer sources/data. Generic bodies may refer to private embedded values in their own module.

Deterministic decoding and failures

Local construction, computation and cleanup are allowed. The decoder's checked call/callback/cleanup closure must avoid I/O, clocks, randomness, tasks, unsafe operations and unknown foreign behavior. Prebuilt BASIC decoders require a versioned build-effects contract; legacy packages work normally but must be rebuilt to serve as opaque build-time decoders. Scheduler EFFECTS alone is not evidence of deterministic behavior.

A Linux x86-64 child receives the authorized content and has bounded memory, CPU, wall time and output. Filesystem/network/process access and executable memory are denied. Unsupported

kernel isolation fails the build. Normal decoder failures clean owned values, then report the declaration's source line/column, input path, concrete error and primitive error fields. JSON offsets additionally map to input line/scalar-column alongside path/offset/message fields. Diagnostic I/O does not replace the primary failure. Nonprimitive error fields are not dumped; user Display/serialization hooks do not run while reporting failures. Forced termination/fatal allocation does not promise normal decoder cleanup.

Current build policy: 8 MiB per input; 32 MiB aggregate input including an eight-byte length per declaration; 32 MiB evaluator image; 64 MiB value packet; 1,048,576 value nodes; nesting depth128; 16 KiB diagnostics; 256 MiB child virtual memory; 5 CPU seconds and 10 wall seconds. Exceeding a limit is a build diagnostic, not a recoverable error in the resulting application.

Caches and output

`build/.basic/embed` beneath the entry source directory holds optional evaluator images and successful checked packets. Keys include compiler/SDK, target/policy, source/module/project identities, type schemas, decoder/default code and compiled inputs. Changed typed content reuses a compatible evaluator but reruns decoding. Semantic/effect checks and packet validation still run on hits. Damaged entries miss/rebuild; a malformed executable with otherwise valid cache metadata is explicitly rejected by the job boundary. Failures are never cached. Readonly projects build without a writable cache. Remove this directory to clear it.

Text encoding

STRING remains valid UTF-8. These built-in functions explicitly convert between STRING and LIST OF BYTE. No import, native codec library, locale setting or runtime mapping file is required. Existing EncodeUtf8/DecodeUtf8 keep their contracts; file/socket/database APIs do not silently choose a legacy encoding.

```
bytes=EncodeText("café",TextEncoding.Windows1252)
PRINT DecodeText(bytes,TextEncoding.Windows1252)
```

Read encoded files with ReadBinaryFile, decode their bytes, then encode text before writing bytes. Conversion adds no NUL terminator.

Profiles

TextEncoding member	Precise profile
UTF8	Shortest-form scalar UTF-8
UTF16LE / UTF16BE	Explicit-endian UTF-16, checked surrogate pairs
UTF32LE / UTF32BE	Explicit-endian scalar UTF-32
ASCII	US-ASCII, 0…127
Latin1	ISO-8859-1, including C1 controls
Windows1252	Pinned vendor table; undefined positions reject
CP437	Pinned IBM/DOS map; control bytes retain control meanings
ShiftJIS	Pinned JIS X 0208 profile, distinct from CP932
Big5	Pinned Big5, distinct from CP950/HKSCS
EUCKR	KS X 1001, including modern Hangul composition fallback
GB18030_2000	Versioned compatibility mapping
GB18030_2022	Revised assignments, including reciprocal four-byte changes

Both GB versions represent every Unicode scalar, including private-use, unassigned and noncharacter values. Their byte mappings differ. The 2022 profile preserves strict scalar round trips; ICU/web fallback aliases are separate behavior. See exact profiles, pinned 2022 derivation and Unicode's mapping-change account. No detection, normalization, transliteration, newline conversion or best-fit fallback is implicit. CP932/CP950/HKSCS/CP949, ISO-2022 and UTF-7 are outside this API.

Functions

Function	Result / purpose
EncodeText(text, encoding, errors, bom, maxOutputBytes)	LIST OF BYTE
DecodeText(bytes, encoding, errors, bom, maxOutputBytes)	Valid UTF-8 STRING
EncodeTextWithInfo / DecodeTextWithInfo	EncodedText / DecodedText
NewTextEncoder / NewTextDecoder	Independent opaque streaming state
EncodeTextChunk(BYREF encoder, text, final=FALSE)	Output byte chunk
DecodeTextChunk(BYREF decoder, bytes, final=FALSE)	Output string chunk
FinishTextEncoder / FinishTextDecoder	Finish once, validating pending input
EncoderInfo / DecoderInfo	Independent TextConversionInfo snapshot

Encoding is required. Defaults: errors=TextErrorPolicy.Strict, decode bom=TextDecodeBom.Preserve, encode bom=TextEncodeBom.Omit, maxOutputBytes=67108864

(64 MiB per call/chunk). The limit accepts 0…1 GiB and counts actual output bytes, including emitted BOMs. It does not cap the stream.

EncodedText.Bytes and DecodedText.Text hold the output; .Info contains InputBytes, OutputBytes, Replacements, FirstReplacementOffset (-1 if none), and Finished. Streaming counters are cumulative and use checked INTEGER byte offsets.

```
result=EncodeTextWithInfo("a€😀",TextEncoding.ASCII,TextErrorPolicy.Replace)
PRINT DecodeUtf8(result.Bytes)
PRINT result.Info.Replacements
```

Replacement is opt-in: decode uses U+FFFD per malformed span; legacy encode uses `?` once per unrepresentable scalar. Strict encoding rejects lossy aliases, including Shift-JIS yen/overline aliases. Unicode encoding of a valid STRING is lossless. Duplicate byte spellings may canonicalize when re-encoded.

Errors and BOMs

TextEncodingError fields: Encoding, Operation, Offset, Length, Reason, Message. TextOperation is Decode or Encode. Reasons: InvalidSequence, Unrepresentable, MissingBom, UnexpectedBom, InvalidOption, OutputLimit, AlreadyFinished. Offset/Length identify external source bytes on decode and input UTF-8 bytes on encode, not BASIC scalar positions. Offsets remain absolute across stream chunks. Constructors declare TextEncodingError. Conversion/chunk/finish functions also retain IndexError, NumericError and StringError contracts from checked helpers; they do not mislabel implementation failures as malformed data.

```
TRY
    text=DecodeText([BYTE(255)],TextEncoding.UTF8)
CATCH err AS TextEncodingError
    PRINT err.Offset
    PRINT err.Length
END TRY
```

Decode Preserve retains U+FEFF. Consume removes a matching initial BOM when present; Require demands it. Consume/Require reject opposite byte order even in replacement mode; they never change the chosen encoding. Under UTF16LE, FF FE 00 00 is BOM plus NUL; under UTF32LE it is just BOM. Encode Emit writes one initial BOM, including for empty text, and retains existing U+FEFF content. Consume/Require/Emit reject non-Unicode profiles.

Streaming and ownership

```
decoder=NewTextDecoder(TextEncoding.UTF16LE)
PRINT DecodeTextChunk(BYREF decoder,[BYTE(65)])
PRINT DecodeTextChunk(BYREF decoder,[BYTE(0)],final=TRUE)
PRINT DecoderInfo(decoder).Finished
```

A decoder retains at most seven pending bytes for split BOMs, surrogate pairs, regional sequences or Hangul composition. Encoders take complete STRING chunks and emit a BOM at most once. final=TRUE finishes; explicit Finish supplies empty input. Reusing a finished converter, including a second Finish, fails.

Converters are copyable opaque values: copies fork independently. Returned buffers and metadata snapshots are independent. BYREF chunk operations exclusively borrow the state. Every recoverable chunk failure leaves it unchanged and returns no partial chunk. Earlier chunks/external writes are not rolled back. Allocation failure follows the language's fatal allocation policy. Ordinary cleanup and compiler scheduling/cancellation apply; no codec thread, hidden shared mutable state, GC or reference counting is introduced.

Byte buffers, text builders and streams

These APIs are available without an import. They use ordinary checked calls, independent values and deterministic cleanup. See low-level capacity/I/O and implementation contracts.

```
builder = StringBuilder(Capacity=256)
Append(BYREF builder, "Hello, ")
AppendLine(BYREF builder, "Fred")
Append(BYREF builder, f"Count: {42:04d}")
text = TakeString(BYREF builder)

packet = ByteWriter(Capacity=64)
WriteUInt32(BYREF packet, UINT32(ByteCount(text)), Order=ByteOrder.BigEndian)
WriteBytes(BYREF packet, EncodeUtf8(text))
bytes = TakeBytes(BYREF packet)
reader = ByteReader(TAKE bytes)
count = ReadUInt32(BYREF reader, Order=ByteOrder.BigEndian)
PRINT DecodeUtf8(ReadBytes(BYREF reader, CINT(count)))
```

Accumulation

Operation	Contract
`ByteBuffer(Capacity=0)`, `StringBuilder(Capacity=0)`	Empty value; reserve at least this many **bytes**. Invalid capacity raises IndexError; allocation failure is fatal.
`ByteBuffer(bytes)`	BYVAL packed-byte input: independent snapshot, or explicit `TAKE bytes` to adopt it.
`StringBuilder(text)`	Infallible valid UTF-8 input, including embedded NUL.
`Length(buffer)`, `Length(builder)`	Initialized bytes / Unicode scalar count. `ByteCount(builder)` returns UTF-8 bytes.
`Capacity(value)`, `Reserve(BYREF value,capacity)`	Byte capacity, including reserved storage; Reserve leaves length/ content unchanged and returns0.
`Append(BYREF buffer,BYTE(...))`	Append one byte; return byte length.
`Extend(BYREF buffer,bytes[,start,count])`	Append a packed-byte list or checked span; return byte length. All range checks precede mutation.
`WriteAt(BYREF buffer,offset,bytes)`	Replace an initialized byte span; no growth. Return count or IndexError before mutation.
`Append(BYREF builder,text)`, `AppendChar(BYREF builder,scalar)`	Append STRING / checked Unicode scalar (StringError when invalid); return scalar length.
`AppendLine(BYREF builder,text="")`	Append text plus LF; return scalar length. Interpolate values before appending them.
`Clear(BYREF value)`	Empty the accumulator while retaining capacity.
`Truncate(BYREF buffer,length)`	Shrink bytes only; checked IndexError.
`ToBytes(buffer)`, `ToString(builder)`	Independent snapshots. Later source/output mutation and destruction are independent.
`TakeBytes(BYREF buffer)`, `TakeString(BYREF builder)`	Explicitly extract storage and empty the accumulator. Byte extraction swaps in an empty list; text extraction donates UTF-8 storage to an owned STRING header. Extracted capacity is no longer retained by the builder.
`Bytes(buffer)`	Checked read-only borrowed byte-list view. Use it directly in a consumer to avoid a snapshot. Storing a copyable result creates an independent value.

Builders are copyable opaque records. Their storage is private; StringBuilder accepts valid STRINGs/scalars, never arbitrary bytes. Decode external data explicitly with DecodeUtf8 or DecodeText. Positions/counts in byte APIs are bytes, not Python-style negative indices. A borrowed

view retains its source against mutation, relocation, transfer and cleanup through the whole consumer call, including later arguments and suspension. Read-only access is not global immutability. Self-appending a borrowed view is rejected; take a snapshot first.

Binary cursors

`ByteReader(bytes)` takes a BYVAL snapshot (or explicit TAKE); `ByteReader(buffer)` takes a buffer snapshot. `ByteWriter(Capacity=0)` uses ByteBuffer storage.

Reader	Writer / representation
`ReadByte(BYREF reader)`	`WriteByte(BYREF writer,value AS BYTE)`: unsigned8
`ReadInt8(BYREF reader)`	`WriteInt8(BYREF writer,value)`: signed8, INTEGER result/input
`ReadInt16/32/64(BYREF reader[,Order])`	`WriteInt16/32/64(BYREF writer,value[,Order])`: two's complement, INTEGER result/input
`ReadUInt16/32/64(BYREF reader[,Order])`	`WriteUInt16/32/64(BYREF writer,value[,Order])`: UINT32 for16/32, UINT64 for64
`ReadDouble(BYREF reader[,Order])`	`WriteDouble(BYREF writer,value[,Order])`: exact IEEE binary64 bits, preserving signed zero and NaN payloads
`ReadBytes(BYREF reader,count)`	`WriteBytes(BYREF writer,bytes)`: independent byte results / appended input
`ReadInto(BYREF reader,BYREF bytes,start,count)`	Copy exactly count bytes into initialized destination storage

Order defaults to `ByteOrder.LittleEndian`; `ByteOrder.BigEndian` is explicit. Eight-bit operations have no Order argument. Narrow signed writes and UInt16 validate their range with NumericError before changing output. All reads validate the complete requested range with IndexError **before** changing cursor/output. They do not return partial fields. `Position`, `Remaining`, `Length`, `Seek(BYREF reader,position)` and nonnegative `Skip(BYREF reader,count)` inspect/move the cursor; invalid Seek/Skip leave it unchanged. Writers support Length, Capacity, Reserve, Clear, ToBytes, Bytes and TakeBytes with the same contracts as ByteBuffer. Readers/writers remain independent copyable values. Existing zero-argument `ReadByte()` and INTEGER `WriteByte(value)` still access inherited streams.

Buffered files and sockets

```
writer = BufferedWriter(OpenFile("greeting.txt", "w"))
WriteLine(BYREF writer, "Hello, λ")
file = Finish(TAKE writer)

reader = BufferedReader(OpenFile("greeting.txt"), Capacity=4096)
WHILE TRUE
    IF LET line = ReadLine(BYREF reader) THEN
        PRINT line
    ELSE
        EXIT WHILE
    END IF
WEND
```

`BufferedReader(endpoint,Capacity=8192)` and `BufferedWriter(endpoint,Capacity=8192)` own a File or TCPConnection, including TLS connections. Named endpoints require TAKE; fresh owned results are adopted. Nonpositive capacities raise IndexError and clean the consumed endpoint. The resulting records are noncopyable resources, including inside owning collections or source-free BASIC libraries. Their Endpoint view is borrowed and cannot become another owner. Existing read workers/socket readiness/TLS retry leases and structured cancellation remain in force; buffering adds no thread per stream. File writes/flushes retain the existing synchronous platform boundary. The generic record can hold/release another resource T, but supplied I/O operations require the File/TCPConnection adapters; a public custom-stream protocol is not supplied.

Inherited streams also use these APIs: `BufferedReader(Stdin())`, `BufferedWriter(Stdout())` and `BufferedWriter(Stderr())`. Each factory returns an owned File duplicate; closing the buffer leaves the process's original stream open. Explicit Flush/Finish is still required for pending output. Duplicates share stream position, so coordinate read-ahead with any other reads from stdin. See standard-stream ownership and I/O limits.

Reader operations fail with IoError/IndexError (also StringError for ReadLine):

Operation	Result
`ReadInto(BYREF reader,BYREF bytes,start,count)`	Checked initialized range; up to count bytes from one buffered segment;0 at EOF or for count0.
`ReadByte(BYREF reader)`	OPTIONAL BYTE: SOME for a byte, NONE at EOF.
`ReadBytes(BYREF reader,maximum)`	Up to maximum bytes, filling across segments until the request is satisfied or EOF.
`ReadLine(BYREF reader,MaxBytes=1048576)`	OPTIONAL STRING: strip LF and its preceding CR. An empty line is SOME(“”); empty EOF is NONE. A final unterminated line is returned. Limit counts bytes before LF, including CR; overflow is IoError75.
`BufferedBytes`, `Capacity`, `EndOfStream`	Unread lookahead count, buffer size and observed EOF with no remaining lookahead. EOF is learned by a read, not predicted.
`Endpoint(reader)`	Checked read-only endpoint loan. Direct endpoint I/O bypasses lookahead and can change external stream position; use deliberately.
`Unbuffer(TAKE reader)`	Return endpoint ownership, explicitly discarding unread lookahead.

Buffered reads are stream operations: an I/O, line-limit or UTF-8 failure may follow consumed input. They do not roll back external stream position. Zero-size ReadInto/ReadBytes does not refill. Returned bytes/text are independent.

Writer operations:

Operation	Contract
`Write(BYREF writer,text)`, `WriteLine(BYREF writer,text="")`	Queue UTF-8 bytes, optionally LF; flush when pending bytes reach BufferSize; return accepted byte count. FAILS IoError.
`WriteBytes(BYREF writer,bytes)`	Same for packed bytes; also FAILS IndexError.
`BufferedBytes(writer)`, `BufferSize(writer)`	Unsent byte count / configured flush threshold. Large input is queued whole before output, so this threshold is not a memory limit.
`PendingBytes(writer)`	Independent snapshot of the unsent suffix; checked IndexError contract.
`Flush(BYREF writer)`	Send pending bytes, return the count pending on entry. A successful short write commits progress before another wait/cancellation/failure. This is buffer draining, not file fsync. FAILS IoError.
`Endpoint(writer)`	Checked read-only endpoint loan. Bypassing the buffer can reorder output; flush before direct endpoint writes.
`Finish(TAKE writer)`	Flush and return endpoint ownership. On failure, clean the consumed writer/endpoint and propagate IoError.

After a failed Write/WriteLine/WriteBytes, its complete argument has already been queued. Retry **Flush**, rather than resubmitting that argument: the unsent suffix is retained and committed bytes are not replayed. Explicit Flush/Finish observes operational failures. Automatic cleanup discards pending output and closes the endpoint; it does not attempt fallible flushing or claim that pending data was saved. Cancellation remains a distinct task outcome and releases buffers/waits/endpoints deterministically. Constructors and all recoverable paths follow the same ownership rules as source-compiled resources.

Native images and drawing

Canvas is part of BASIL's embedded standard library. Use its public names directly, without an import, a `.blib`, or an external image/font library. The implementation is maintained BASIL source; programs emit only reachable code and immutable tables. See the complete API, format profiles, fonts/text, compiler contracts, and measured integration.

```
image = NewCanvas(64, 32)
ClearCanvas(BYREF image, RGB(20, 30, 40))
SetPixel(BYREF image, 0, 0, RGBA(255, 0, 0, 128))
copy = CloneCanvas(image)
SetPixel(BYREF copy, 0, 0, RGB(0, 255, 0))
PRINT GetPixel(image, 0, 0).Red
PRINT GetPixel(copy, 0, 0).Green
PRINT CanvasWidth(DecodePNG(EncodePNG(image)))
```

Run the asset-free drawing example to produce a PNG, BMP and JPEG, reload the PNG and create a thumbnail. File output can use `SaveImage(image, "picture.png")`; `LoadImage(path)` identifies supported input by signature. Direct `DecodePNG`/`EncodePNG` and corresponding codec APIs avoid dynamic format dispatch. `EncodeImage(image)` defaults to PNG; a literal or CONST format can select one writer. Filename-driven saving and generic loading retain their possible codecs. `WriteFileAtomic(path, EncodePNG(image))` explicitly selects a single writer for a small file-producing program.

Ownership and coordinates

`Canvas` is an opaque resource owning straight, row-major RGBA8 pixels, four bytes per pixel without row padding. Coordinates start at the top left; X grows right, Y grows down. Pixel coordinates and rectangles use zero-based integer positions; geometry uses pixel-edge DOUBLE coordinates. Colors have BYTE Red/Green/Blue/Alpha fields; `RGB` supplies alpha 255. `NewCanvas` initializes every channel to zero, including alpha. `CanvasFromRGBA` checks the exact byte count and independently owns its input unless explicitly transferred.

Ordinary parameters borrow. Mutation uses `BYREF`; the checker excludes aliases that could invalidate the borrow, including across suspension. Assignment cannot implicitly copy a Canvas. `CloneCanvas` creates an independent owner and `CanvasBytes` returns an independent byte snapshot. Scope, failure and cancellation cleanup release owned storage automatically. A task can receive an explicit `TAKE image`; resource-containing lists and source-free libraries use the same ownership rules. There is no hidden reference counting or shared mutation.

`GetPixel`/`SetPixel` reject out-of-range coordinates. Drawing clips to the target. `CopyWithinCanvas` handles overlapping self-copies; passing the same owner as a separate source and mutable destination is an invalid conflicting borrow. Operations returning a new image leave the source unchanged on failure. In-place operations do not promise general rollback; prepared text validates its outlines before drawing.

Available operations

Area	Principal entry points
Pixels and snapshots	NewCanvas, CanvasFromRGBA, CanvasWidth, CanvasHeight, GetPixel, SetPixel, ClearCanvas, CanvasBytes, CloneCanvas
Sampling and geometry	SamplePixel, ResizeCanvas, CropCanvas, Thumbnail, FlipCanvas, TransposeCanvas, RotateCanvas90, RotateCanvas, WarpCanvas, FitCanvas
Transforms	IdentityTransform, TranslationTransform, ScalingTransform, RotationTransform, ComposeTransforms, InvertTransform, TransformPoint
Compositing	CompositeColor, DrawImage, CopyWithinCanvas, MaskedCanvas; 24 BlendMode values, EncodedSRGB/LinearLight

Area	Principal entry points
Drawing	FillRect, FillRoundedRect, FillEllipse, FillCircle, FillPolygon, DrawLine, DrawPolyline, DrawRect, DrawCircle, DrawEllipse
Paths	NewPath, MoveTo, LineTo, QuadraticTo, CubicTo, ClosePath, TransformPath, FillPath, DrawPath, DrawDashedPath, CreatePathMask
Filters and analysis	AdjustedCanvas, GrayscaleCanvas, InvertedCanvas, BoxBlurCanvas, GaussianBlurCanvas, SharpenCanvas, EdgeCanvas, EmbossCanvas, ConvolveCanvas3x3, Histogram, AverageColor, AnalyzeCanvas, AlphaBounds, ColorBounds
Images and animation	DecodeImage, EncodeImage, LoadImage, SaveImage, NewAnimation, AppendAnimationFrame, AnimationFrame, AnimationFrameDelay, DecodeAnimation, EncodeAnimation, LoadAnimation, SaveAnimation
Text	DrawBitmapText, LoadFont, LoadFontBytes, NewFontSet, CreateTextLayout, DrawTextLayout; see text guide

Nearest, Bilinear, Area, Bicubic and Lanczos3 sampling are available. Filtering works in premultiplied color to avoid transparent-edge fringes, then produces straight RGBA8. Pure permutations/lossless copies preserve hidden RGB at alpha 0. Antialiasing offers Off, Standard (4×4), and High (8×8) coverage; strokes support caps, joins, dashes and checked miter limits.

`CanvasError` carries Message, Offset (default −1), and Format (default empty). Malformed/unsupported input and configured limits are recoverable errors. File operations also preserve IoError; numeric/index/string helper failures retain their declared types. A function must catch or declare escaping errors, as with other BASIL APIs. Allocation failure remains fatal. Pixel, encoded-size and work limits are logical bounds, not a whole-process memory reservation or deadline. See the API for each signature and default.

Native image format profiles

All production codecs are BASIL source. These profiles describe implemented behavior, not universal support for every variant of a format. Exact signatures and limit defaults are in the API reference.

Format	Reading	Writing
PNG	Standard color types/bit depths, palette/transparency, filters, Adam7 interlace; CRC and bounded zlib decoding	RGBA8, lossless
BMP	Selected uncompressed 24/32-bit and supported mask profiles	V5 RGBA with alpha
JPEG	8-bit Huffman sequential and progressive; supported grayscale/YCbCr scan arrangements, subsampling and restart intervals	Baseline grayscale or 4:4:4 color, quality selection; explicit background for transparent input
GIF	GIF87a/89a palettes, interlace, transparency, LZW, bounded rectangles, disposal and loop metadata	Palette quantization, transparency, LZW, still and animation
APNG	Coalesced frames with rational delays, offsets, blending and disposal	PNG/APNG animation with explicit frame timing
WebP	VP8 lossy, VP8L lossless, supported alpha and animation containers	Lossless VP8L, still and animation

Use `CanEncodeImage`/`CanDecodeImage` and animation capability queries for format-level availability. AVIF/TIFF enum values are reserved and return false; unsupported profiles raise CanvasError. No lossy WebP writer, progressive JPEG writer, arithmetic/lossless/hierarchical/CMYK JPEG, color management, HDR, EXIF orientation processing or metadata-preserving editor is claimed.

`DecodeImage` checks signatures. Its still-image APIs reject animation rather than silently throwing away frames; use `DecodeAnimation`/`LoadAnimation` for GIF/APNG/animated WebP. An ordinary still image can become a one-frame animation with zero delay. `EncodeImage` defaults to PNG; `EncodeAnimation` defaults to APNG. Save APIs infer supported extensions when the format is Auto. Encoding finishes before atomic file replacement.

`Animation` owns coalesced RGBA8 snapshots. `AppendAnimationFrame` borrows its Canvas and stores an independent snapshot. `AnimationFrame` returns a fresh Canvas. Delays retain their numerator/denominator; `AnimationFrameDelay` exposes a rounded typed Duration. Zero delay is valid. There is no implicit playback task or display thread. Play-count interpretation is normalized by the container adapters. `DrawAnimationFrame` composites a frame without first cloning it.

The default canvas pixel limit is 16,777,216; dimensions allow configurable limits up to 67,108,864 pixels, with overflow checks before multiplication/allocation. Generic encoded input defaults to 64 MiB. Animation decoding defaults to 1,000 frames and 256 MiB decoded/work storage; retained animations permit at most 10,000 frames. Codec-specific plane, coefficient and dictionary budgets are checked separately. Large padded JPEG MCU planes count toward its work limit. These limits exclude some allocator capacity and caller-owned input, and are not a promise about peak process RSS.

Native fonts and text

`DrawBitmapText` supplies a small public-domain ASCII font without a font file. Other scalars render as a replacement unless strict glyph checking is requested. Tabs/newlines and integer scaling are supported. Font-file APIs read static OpenType TrueType glyf or CFF1 outlines, including TTC/OTC collection face indexes. They support cmap formats 4/12, proportional metrics, compound glyphs and CFF Type2/CID subroutines. Font instruction bytecode is checked/skipped, never executed. Rendering is grayscale and unhinted, without display-dependent subpixel color.

`LoadFont(path)` and `LoadFontBytes(bytes)` return opaque owning Font values. `FontHasGlyph`, `GetFontMetrics`, `GetKerning`, `MeasureText` and `DrawText` borrow them. Missing glyphs default to glyph zero; StrictGlyphs requests errors before drawing. A Font cannot be implicitly copied. `AddFont(BYREF fonts, TAKE font)` transfers it into an ordered FontSet; `AddFontFile` is a convenience for loading and adding. Automatic cleanup drops the whole set.

`CreateTextLayout(fonts, text, TextLayoutOptions {...})` returns an independent opaque TextLayout owning prepared outlines. The single-Font overload uses the same checked shaping machinery without cloning font bytes. The originating font or set can leave scope before drawing the layout. `DrawTextLayout` borrows it; repeated draws do not reshape or clone the outline buffer. Text measurement, line/glyph queries and hit testing report original scalar positions, rather than normalized or visual-order indexes.

Layout includes Unicode 17 extended grapheme boundaries, bidi resolution, font-aware canonical normalization, whole-grapheme fallback, tabs/newlines, word/grapheme wrapping, alignment, line spacing and per-line shaping. Public `GraphemeBoundaries`, `ResolveBidiLevels` and `BidiVisualOrder` expose the relevant text analysis. Normalization is currently internal to shaping; it is not a new standalone public normalization API.

The OpenType reader implements GSUB 1–8 and GPOS 1–9 with contextual/extension lookups, ligatures, pair/class kerning, marks and cursive attachments, plus GDEF filtering. Joining/reordering stages cover the scripts in the independent corpus: Latin, Arabic, Hebrew, Devanagari, Bengali, Gujarati, Gurmukhi, Tamil, Telugu, Kannada, Malayalam, Oriya, Sinhala, Thai, Lao, Khmer and Myanmar. Passing that corpus does not establish exhaustive typographic conformance for every font or language. Variable/CFF2/color fonts, WOFF/WOFF2 and hint execution are unsupported.

`ReadFontInfo`, `DiscoverFonts`, `DiscoverSystemFonts`, `FindFont` and explicit substitution flags provide bounded metadata discovery. The current Linux system defaults are `/usr/share/fonts` and `/usr/local/share/fonts`, isolated in linux_fonts.bas. Pass an explicit directory list for other locations; bundle licensed fonts when exact cross-machine appearance matters. Discovery warnings are available separately.

Default font input is bounded to 32 MiB (configurable to 128 MiB). Layout bounds input/shaped glyphs to 65,536 and outlines to a configurable maximum of 1,048,576 edges. Lookup recursion/work and normalization expansion have separate bounds. Malformed fonts and exhausted limits raise CanvasError; file errors retain IoError. Prepared layout construction fails before target rasterization.

The pinned Unicode 17 corpus covers 766 grapheme vectors, 91,707 bidi vectors and 20,034 normalization vectors (200,340 checks). Synthetic fonts exercise malformed tables, outlines, collections, attachments, GSUB/GPOS primitives and atomic preparation failures. Real-font controls compare CFF rasterization with Pillow and glyph identities/positions with HarfBuzz. Source-free library tests return Fonts and prepared layouts, destroy local source fonts, transfer Fonts into sets, and render the returned layout afterwards. See the acceptance for the actual tested inputs and results.

Language coverage

This edition describes the maintained Linux x86-64 implementation. The table below points from each language area to its teaching chapter. The language specification defines exact semantics; unsupported features are identified in the troubleshooting chapter. Build provenance and detailed verification records accompany the editable book source.

Teaching examples: **124 executed with exact expected output; 4 intentionally rejected**. 8 excerpts/interactive/external examples are labelled with their requirements and are not claimed as executed. Library reference excerpts have separate regression coverage.

Implemented-language coverage

Specification area	Teaching chapter
Status and authority	1. Your first program
Source, names and modules	12. Modules, projects and packages
Program entry and statements	1. Your first program
Types, literals and operators	2. Values, variables and expressions
Locals, initialization and assignment	9. Ownership and borrowing, from first principles
Functions, calls and generics	13. Generics, callbacks and custom value behavior
Records, choices, enums and optionals	6. Records, alternatives and missing values
Strings, collections and arrays	7. Lists, sets and dictionaries
Structured control flow	3. Decisions and repetition
Errors, cancellation and ownership	11. Checked errors and recovery
Tasks, channels and SELECT	15. Channels, SELECT and timeouts
JSON and typed data adapters	17. JSON, schemas and serialization
Library and platform scope	23. Native interfaces and advanced library design
Logging	22. Building complete applications
Commands and project packages	12. Modules, projects and packages
Implementation limits	24. Diagnostics, limits and migration
Planned and unresolved behavior	24. Diagnostics, limits and migration
Dates, times, durations and clocks	18. Dates, clocks, UUIDs and randomness
UUID and secure randomness	18. Dates, clocks, UUIDs and randomness
URL values and builders	19. URLs, TCP and HTTP
Embedded values	17. JSON, schemas and serialization
External text encodings	5. Text and formatting
Buffers, builders and byte streams	21. Binary formats, compression and numerical work
Images and text rendering	22. Building complete applications

Public prelude declaration catalog

Generated from `basic api` using this build's compiler. Signatures preserve overloads, parameter modes, defaults and declared failures. This is declaration lookup: it does not enumerate record fields, every intrinsic, or imported package APIs, and it does not replace the explanations and examples in the guide.

A

```
FUNCTION ABS(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION ABS(value AS INTEGER) AS INTEGER FAILS NumericError
```

```
FUNCTION AbsDuration(value AS Duration) AS Duration FAILS DateTimeError
```

```
FUNCTION Accept(listener AS TCPListener) AS TCPConnection FAILS IoError
```

```
FUNCTION Accept(listener AS TCPListener,timeout AS Duration) AS TCPConnection FAILS IoError
```

```
FUNCTION ACOS(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION ACOSH(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION Add[T](BYREF c AS Counter[T],key AS T,amount AS INTEGER=1) AS INTEGER FAILS
CollectionError,LIKE ValidateKey(AS T)
```

```
SUB Add[K,V](BYREF m AS MultiMap[K,V],key AS K,BYVAL value AS V) FAILS
CollectionError,MissingKey,LIKE ValidateKey(AS K)
```

```
FUNCTION AddDays(value AS Date,days AS INTEGER) AS Date FAILS DateTimeError
```

```
FUNCTION AddDuration(left AS Duration,right AS Duration) AS Duration FAILS DateTimeError
```

```
FUNCTION AddDuration(value AS Instant,amount AS Duration) AS Instant FAILS DateTimeError
```

```
FUNCTION AddDuration(value AS OffsetDateTime,amount AS Duration) AS OffsetDateTime FAILS
DateTimeError
```

```
FUNCTION AddDuration(value AS ZonedDateTime,amount AS Duration) AS ZonedDateTime FAILS
DateTimeError
```

```
SUB AddFont(BYREF fonts AS FontSet,BYVAL font AS Font) FAILS CanvasError
```

```
SUB AddFontFile(BYREF fonts AS FontSet,path AS STRING,maxBytes AS INTEGER=33554432,faceIndex
AS INTEGER=0) FAILS CanvasError,IndexError,NumericError,IoError
```

```
SUB AddLogSink(BYVAL sink AS LogSink) FAILS LoggingError
```

```
SUB AddLogSink(handler AS CALLABLE(LogRecord) FAILS (Error)) FAILS LoggingError
```

```
SUB AddLogSink(logger AS Logger,BYVAL sink AS LogSink) FAILS LoggingError
```

```
SUB AddLogSink(logger AS Logger,handler AS CALLABLE(LogRecord) FAILS (Error)) FAILS
LoggingError
```

```
SUB AddLogSink(logger AS LoggerHandle,BYVAL sink AS LogSink) FAILS LoggingError
```

```
SUB AddLogSink(logger AS LoggerHandle,handler AS CALLABLE(LogRecord) FAILS (Error)) FAILS
LoggingError
```

```
SUB AddLogSink[T](BYVAL context AS T,handler AS CALLABLE(BYREF T,LogRecord) FAILS (Error))
FAILS LoggingError
```

```
SUB AddLogSink[T](BYVAL context AS T,handler AS CALLABLE(T,LogRecord) FAILS (Error)) FAILS
LoggingError
```

```
SUB AddLogSink[T](logger AS Logger,BYVAL context AS T,handler AS CALLABLE(BYREF T,LogRecord)
FAILS (Error)) FAILS LoggingError
```

```
SUB AddLogSink[T](logger AS Logger,BYVAL context AS T,handler AS CALLABLE(T,LogRecord) FAILS
(Error)) FAILS LoggingError
```

```
SUB AddLogSink[T](logger AS LoggerHandle,BYVAL context AS T,handler AS CALLABLE(BYREF
T,LogRecord) FAILS (Error)) FAILS LoggingError
```

```
SUB AddLogSink[T](logger AS LoggerHandle,BYVAL context AS T,handler AS CALLABLE(T,LogRecord)
FAILS (Error)) FAILS LoggingError
```

```
FUNCTION AddMonths(value AS Date,months AS INTEGER,overflow AS
DateOverflow=DateOverflow.Reject) AS Date FAILS DateTimeError
```

```
FUNCTION AddNominalDuration(value AS LocalDateTime,amount AS Duration) AS LocalDateTime FAILS
DateTimeError
```

```
FUNCTION AddPeriod(value AS Date,amount AS Period,overflow AS
DateOverflow=DateOverflow.Reject) AS Date FAILS DateTimeError
```

```
FUNCTION AddPeriod(value AS LocalDateTime,amount AS Period,overflow AS
DateOverflow=DateOverflow.Reject) AS LocalDateTime FAILS DateTimeError
```

```
FUNCTION AddPeriod(value AS ZonedDateTime,amount AS Period,options AS CalendarAddOptions) AS
ZonedDateTime FAILS DateTimeError
```

```
FUNCTION AddPeriods(left AS Period,right AS Period) AS Period FAILS DateTimeError
```

```
SUB AddURLQueryParameter(BYREF parameters AS LIST OF URLParameter,name AS STRING,value AS
STRING)
```

```
SUB AddURLQueryParameter(BYREF value AS URL,name AS STRING,text AS STRING) FAILS UrlError
```

```
FUNCTION AddWeeks(value AS Date,weeks AS INTEGER) AS Date FAILS DateTimeError
```

```
FUNCTION AddYears(value AS Date,years AS INTEGER,overflow AS
DateOverflow=DateOverflow.Reject) AS Date FAILS DateTimeError
```

```
FUNCTION AdjustedCanvas(canvas AS Canvas,brightness AS DOUBLE=1.0,contrast AS
DOUBLE=1.0,saturation AS DOUBLE=1.0,gamma AS DOUBLE=1.0) AS Canvas FAILS
CanvasError,IndexError,NumericError
```

```
SUB AdjustTestWallClock(BYREF clock AS Clock,adjustment AS Duration) FAILS DateTimeError
```

```
SUB AdvanceTestClock(BYREF clock AS Clock,elapsed AS Duration) FAILS DateTimeError
```

```
FUNCTION All[K,V](items AS DICT OF K TO V,Predicate AS CALLABLE(K,V) AS BOOLEAN) AS BOOLEAN
```

```
FUNCTION All[T](items AS LIST OF T,Predicate AS CALLABLE(T) AS BOOLEAN) AS BOOLEAN
```

```
FUNCTION All[T](items AS SET OF T,Predicate AS CALLABLE(T) AS BOOLEAN) AS BOOLEAN
```

```
FUNCTION AlphaBounds(canvas AS Canvas,threshold AS INTEGER=0) AS OPTIONAL OF PixelRect FAILS
CanvasError,IndexError,NumericError
```

```
FUNCTION AnalyzeCanvas(canvas AS Canvas) AS ImageStatistics FAILS CanvasError,IndexError
```

```
FUNCTION AnalyzeCanvasRegion(canvas AS Canvas,region AS PixelRect) AS ImageStatistics FAILS
CanvasError,IndexError
```

```
FUNCTION AnimationDelayDenominator(animation AS Animation,index AS INTEGER) AS INTEGER FAILS
CanvasError,IndexError
```

```
FUNCTION AnimationDelayNumerator(animation AS Animation,index AS INTEGER) AS INTEGER FAILS
CanvasError,IndexError
```

```
FUNCTION AnimationFrame(animation AS Animation,index AS INTEGER) AS Canvas FAILS
CanvasError,IndexError
```

```
FUNCTION AnimationFrameCount(animation AS Animation) AS INTEGER
```

```
FUNCTION AnimationFrameDelay(animation AS Animation,index AS INTEGER) AS Duration FAILS
CanvasError,IndexError
```

```
FUNCTION AnimationHeight(animation AS Animation) AS INTEGER
```

```
FUNCTION AnimationPlays(animation AS Animation) AS INTEGER
```

```
FUNCTION AnimationWidth(animation AS Animation) AS INTEGER
```

```
FUNCTION Any[K,V](items AS DICT OF K TO V,Predicate AS CALLABLE(K,V) AS BOOLEAN) AS BOOLEAN
```

```
FUNCTION Any[T](items AS LIST OF T,Predicate AS CALLABLE(T) AS BOOLEAN) AS BOOLEAN
```

```
FUNCTION Any[T](items AS SET OF T,Predicate AS CALLABLE(T) AS BOOLEAN) AS BOOLEAN
```

```
FUNCTION Append(BYREF buffer AS ByteBuffer,value AS BYTE) AS INTEGER
```

```
FUNCTION Append(BYREF builder AS StringBuilder,text AS STRING) AS INTEGER
```

```
FUNCTION Append[T](BYREF items AS LIST OF T,BYVAL value AS T) AS INTEGER
```

```
SUB AppendAnimationFrame(BYREF animation AS Animation,canvas AS Canvas,delayNumerator AS
INTEGER,delayDenominator AS INTEGER=1000) FAILS CanvasError
```

```
FUNCTION AppendChar(BYREF builder AS StringBuilder,scalar AS INTEGER) AS INTEGER FAILS
StringError
```

```
FUNCTION AppendLine(BYREF builder AS StringBuilder,text AS STRING="") AS INTEGER
```

```
FUNCTION AppendUInt64LE(BYREF bytes AS LIST OF BYTE,value AS UINT64) AS INTEGER
```

```
FUNCTION ApplyAmount(value AS LocalDateTime,amount AS DateTimeAmount,overflow AS
DateOverflow=DateOverflow.Reject) AS LocalDateTime FAILS DateTimeError
```

```
FUNCTION ApplyAmount(value AS ZonedDateTime,amount AS DateTimeAmount,options AS
CalendarAddOptions) AS ZonedDateTime FAILS DateTimeError
```

```
FUNCTION ArgumentBoolean(value AS BOOLEAN) AS ArgumentValue
```

```
FUNCTION ArgumentInteger(value AS INTEGER) AS ArgumentValue
```

```
FUNCTION ArgumentOptions[T](parsed AS ParsedArguments) AS T FAILS ArgumentError
```

```
FUNCTION ArgumentReal(value AS DOUBLE) AS ArgumentValue
```

```
FUNCTION ArgumentText(value AS STRING) AS ArgumentValue
```

```
FUNCTION ASC(text AS STRING) AS INTEGER FAILS StringError
```

```
FUNCTION ASIN(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION ASINH(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION At[T](s AS SortedSet[T],index AS INTEGER) AS T FAILS IndexError
```

```
FUNCTION ATAN(input AS DOUBLE) AS DOUBLE
```

```
FUNCTION ATAN2(y AS DOUBLE, x AS DOUBLE) AS DOUBLE
```

```
FUNCTION ATANH(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION AtDay(value AS YearMonth,day AS INTEGER) AS Date FAILS DateTimeError
```

```
FUNCTION ATN(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION AtOffset(value AS Instant,offset AS UtcOffset) AS OffsetDateTime FAILS DateTimeError
```

```
FUNCTION AtYear(value AS MonthDay,year AS INTEGER,overflow AS
DateOverflow=DateOverflow.Reject) AS Date FAILS DateTimeError
```

```
FUNCTION AtZone(value AS Instant,zone AS TimeZone) AS ZonedDateTime FAILS DateTimeError
```

```
FUNCTION AverageColor(canvas AS Canvas,alphaWeighted AS BOOLEAN=FALSE) AS Color FAILS
CanvasError,IndexError,NumericError
```

B

```
FUNCTION Back[T](q AS BoundedDeque[T]) AS BORROW T FROM q FAILS EmptyCollection WHERE T IS
RESOURCE
```

```
FUNCTION Back[T](q AS BoundedDeque[T]) AS T FAILS EmptyCollection WHERE T IS COPYABLE
```

```
FUNCTION Back[T](q AS Deque[T]) AS BORROW T FROM q FAILS EmptyCollection WHERE T IS RESOURCE
```

```
FUNCTION Back[T](q AS Deque[T]) AS T FAILS EmptyCollection WHERE T IS COPYABLE
```

```
FUNCTION BidiVisualOrder(levels AS LIST OF INTEGER) AS LIST OF INTEGER FAILS
CanvasError,IndexError
```

```
FUNCTION BitAnd(left AS UINT64, right AS UINT64) AS UINT64
```

```
FUNCTION BitCount(value AS UINT64) AS INTEGER
```

```
FUNCTION BitDifference(left AS BitSet,right AS BitSet) AS BitSet FAILS CollectionError
```

```
FUNCTION BitIntersection(left AS BitSet,right AS BitSet) AS BitSet FAILS CollectionError
```

```
FUNCTION BitNot(value AS UINT64) AS UINT64
```

```
FUNCTION BitOr(left AS UINT64, right AS UINT64) AS UINT64
```

```
FUNCTION BitSymmetricDifference(left AS BitSet,right AS BitSet) AS BitSet FAILS
CollectionError
```

```
FUNCTION BitUnion(left AS BitSet,right AS BitSet) AS BitSet FAILS CollectionError
```

```
FUNCTION BitXor(left AS UINT64, right AS UINT64) AS UINT64
```

```
FUNCTION BooleanArgument(values AS DICT OF STRING TO ArgumentValue,name AS STRING) AS BOOLEAN
FAILS ArgumentError
```

```
FUNCTION BooleanOption(name AS STRING,Default AS BOOLEAN=FALSE,Short AS STRING="",Help AS
STRING="",Global AS BOOLEAN=FALSE) AS CommandOption
```

```
FUNCTION BoxBlurCanvas(canvas AS Canvas,radius AS INTEGER=1) AS Canvas FAILS
CanvasError,IndexError,NumericError
```

```
FUNCTION BufferedBytes[T](reader AS BufferedReader[T]) AS INTEGER WHERE T IS RESOURCE
```

```
FUNCTION BufferedBytes[T](writer AS BufferedWriter[T]) AS INTEGER WHERE T IS RESOURCE
```

```
FUNCTION BufferedReader[T](TAKE source AS T,Capacity AS INTEGER=8192) AS BufferedReader[T]
FAILS IndexError WHERE T IS RESOURCE
```

```
FUNCTION BufferedWriter[T](TAKE destination AS T,Capacity AS INTEGER=8192) AS
BufferedWriter[T] FAILS IndexError WHERE T IS RESOURCE
```

```
FUNCTION BufferSize[T](writer AS BufferedWriter[T]) AS INTEGER WHERE T IS RESOURCE
```

```
FUNCTION BuildURL(scheme AS STRING,host AS STRING,Query AS DICT OF STRING TO LIST OF
STRING,path AS STRING="/",port AS INTEGER=-1) AS URL FAILS UrlError
```

```
FUNCTION BuildURL(scheme AS STRING,host AS STRING,Query AS DICT OF STRING TO STRING,path AS
STRING="/",port AS INTEGER=-1) AS URL FAILS UrlError
```

```
FUNCTION BuildURL(scheme AS STRING,host AS STRING,Query AS LIST OF URLParameter,path AS
STRING="/",port AS INTEGER=-1) AS URL FAILS UrlError
```

```
FUNCTION BuildURL(scheme AS STRING,host AS STRING,path AS STRING="/",port AS INTEGER=-1) AS
URL FAILS UrlError
```

```
FUNCTION ByteBuffer(BYVAL bytes AS LIST OF BYTE) AS ByteBuffer
```

```
FUNCTION ByteBuffer(Capacity AS INTEGER=0) AS ByteBuffer FAILS IndexError
```

```
FUNCTION ByteCount(builder AS StringBuilder) AS INTEGER
```

```
FUNCTION ByteCount(text AS STRING) AS INTEGER
```

```
FUNCTION ByteReader(BYVAL bytes AS LIST OF BYTE) AS ByteReader
```

```
FUNCTION ByteReader(buffer AS ByteBuffer) AS ByteReader
```

```
FUNCTION Bytes(buffer AS ByteBuffer) AS BORROW LIST OF BYTE FROM buffer
```

```
FUNCTION Bytes(writer AS ByteWriter) AS BORROW LIST OF BYTE FROM writer
```

```
FUNCTION BytesResponse(bytes AS LIST OF BYTE,status AS INTEGER=200,contentType AS
STRING="application/octet-stream") AS HTTPResponse
```

```
FUNCTION ByteWriter(Capacity AS INTEGER=0) AS ByteWriter FAILS IndexError
```

C

```
FUNCTION CacheCapacity[K,V](cache AS Cache[K,V]) AS INTEGER
```

```
FUNCTION CacheCount[K,V](BYREF cache AS Cache[K,V]) AS INTEGER FAILS
CollectionError,DateTimeError,IoError,LIKE ValidateKey(AS K)
```

```
FUNCTION CacheCountAt[K,V](BYREF cache AS Cache[K,V],now AS MonotonicInstant) AS INTEGER
FAILS CollectionError,DateTimeError,IoError,LIKE ValidateKey(AS K)
```

```
FUNCTION CacheExtract[K,V](BYREF cache AS Cache[K,V],key AS K) AS OPTIONAL OF V FAILS
CollectionError,DateTimeError,IoError,LIKE ValidateKey(AS K)
```

```
FUNCTION CacheExtractAt[K,V](BYREF cache AS Cache[K,V],key AS K,now AS MonotonicInstant) AS
OPTIONAL OF V FAILS CollectionError,DateTimeError,IoError,LIKE ValidateKey(AS K)
```

```
FUNCTION CacheFind[K,V](BYREF cache AS Cache[K,V],key AS K) AS BORROW OPTIONAL OF V FROM
cache FAILS CollectionError,DateTimeError,IoError,LIKE ValidateKey(AS K) WHERE V IS RESOURCE
```

```
FUNCTION CacheFind[K,V](BYREF cache AS Cache[K,V],key AS K) AS OPTIONAL OF V FAILS
CollectionError,DateTimeError,IoError,LIKE ValidateKey(AS K) WHERE V IS COPYABLE
```

```
FUNCTION CacheFindAt[K,V](BYREF cache AS Cache[K,V],key AS K,now AS MonotonicInstant) AS
BORROW OPTIONAL OF V FROM cache FAILS CollectionError,DateTimeError,IoError,LIKE
ValidateKey(AS K) WHERE V IS RESOURCE
```

```
FUNCTION CacheFindAt[K,V](BYREF cache AS Cache[K,V],key AS K,now AS MonotonicInstant) AS
OPTIONAL OF V FAILS CollectionError,DateTimeError,IoError,LIKE ValidateKey(AS K) WHERE V IS
COPYABLE
```

```
FUNCTION CachePeek[K,V](cache AS Cache[K,V],key AS K) AS BORROW OPTIONAL OF V FROM cache
FAILS CollectionError,DateTimeError,IoError,LIKE ValidateKey(AS K) WHERE V IS RESOURCE
```

```
FUNCTION CachePeek[K,V](cache AS Cache[K,V],key AS K) AS OPTIONAL OF V FAILS
CollectionError,DateTimeError,IoError,LIKE ValidateKey(AS K) WHERE V IS COPYABLE
```

```
FUNCTION CachePeekAt[K,V](cache AS Cache[K,V],key AS K,now AS MonotonicInstant) AS BORROW
OPTIONAL OF V FROM cache FAILS CollectionError,DateTimeError,IoError,LIKE ValidateKey(AS K)
WHERE V IS RESOURCE
```

```
FUNCTION CachePeekAt[K,V](cache AS Cache[K,V],key AS K,now AS MonotonicInstant) AS OPTIONAL
OF V FAILS CollectionError,DateTimeError,IoError,LIKE ValidateKey(AS K) WHERE V IS COPYABLE
```

```
FUNCTION CachePut[K,V](BYREF cache AS Cache[K,V],key AS K,BYVAL value AS V,weight AS
INTEGER=1) AS CachePutResult FAILS CollectionError,DateTimeError,IoError,LIKE ValidateKey(AS
K)
```

```
FUNCTION CachePutAt[K,V](BYREF cache AS Cache[K,V],key AS K,BYVAL value AS V,now AS
MonotonicInstant,weight AS INTEGER=1) AS CachePutResult FAILS
CollectionError,DateTimeError,IoError,LIKE ValidateKey(AS K)
```

```
FUNCTION CacheRemove[K,V](BYREF cache AS Cache[K,V],key AS K) AS BOOLEAN FAILS
CollectionError,DateTimeError,IoError,LIKE ValidateKey(AS K)
```

```
FUNCTION CacheRemoveAt[K,V](BYREF cache AS Cache[K,V],key AS K,now AS MonotonicInstant) AS
BOOLEAN FAILS CollectionError,DateTimeError,IoError,LIKE ValidateKey(AS K)
```

```
FUNCTION CacheWeightLimit[K,V](cache AS Cache[K,V]) AS INTEGER
```

```
FUNCTION CalendarBetween(start AS Date,finish AS Date,largestUnit AS
CalendarDifferenceUnit=CalendarDifferenceUnit.Day,overflow AS
DateOverflow=DateOverflow.Reject) AS Period FAILS DateTimeError
```

```
FUNCTION CanDecodeAnimation(format AS ImageFormat) AS BOOLEAN
```

```
FUNCTION CanDecodeImage(format AS ImageFormat) AS BOOLEAN
```

```
FUNCTION CanEncodeAnimation(format AS ImageFormat) AS BOOLEAN
```

```
FUNCTION CanEncodeImage(format AS ImageFormat) AS BOOLEAN
```

```
FUNCTION CanonicalPath(path AS STRING) AS STRING FAILS IoError
```

```
FUNCTION CanvasBytes(canvas AS Canvas) AS LIST OF BYTE
```

```
FUNCTION CanvasFromRGBA(width AS INTEGER,height AS INTEGER,BYVAL pixels AS LIST OF
BYTE,maxPixels AS INTEGER=16777216) AS Canvas FAILS CanvasError,NumericError,IndexError
```

```
FUNCTION CanvasHeight(canvas AS Canvas) AS INTEGER
```

```
FUNCTION CanvasWidth(canvas AS Canvas) AS INTEGER
```

```
FUNCTION Capacity(bits AS BitSet) AS INTEGER
```

```
FUNCTION Capacity(buffer AS ByteBuffer) AS INTEGER
```

```
FUNCTION Capacity(builder AS StringBuilder) AS INTEGER
```

```
FUNCTION Capacity(writer AS ByteWriter) AS INTEGER
```

```
FUNCTION Capacity[T](q AS BoundedDeque[T]) AS INTEGER
```

```
FUNCTION Capacity[T](reader AS BufferedReader[T]) AS INTEGER WHERE T IS RESOURCE
```

```
FUNCTION CASEFOLD(text AS STRING) AS STRING
```

```
FUNCTION CBRT(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION CBYTE(value AS INTEGER) AS BYTE FAILS NumericError
```

```
FUNCTION CBYTE(value AS UINT64) AS BYTE FAILS NumericError
```

```
FUNCTION CDBL(input AS STRING) AS DOUBLE FAILS NumericError
```

```
FUNCTION CDBL(value AS BYTE) AS DOUBLE
```

```
FUNCTION CDBL(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION CDBL(value AS INTEGER) AS DOUBLE
```

```
FUNCTION CDBL(value AS UINT32) AS DOUBLE
```

```
FUNCTION CDBL(value AS UINT64) AS DOUBLE
```

```
FUNCTION CEIL(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION CEILING(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION CHR(value AS INTEGER) AS STRING FAILS StringError
```

```
FUNCTION CINT(value AS BYTE) AS INTEGER
```

```
FUNCTION CINT(value AS DOUBLE) AS INTEGER FAILS NumericError
```

```
FUNCTION CINT(value AS INTEGER) AS INTEGER
```

```
FUNCTION CINT(value AS UINT32) AS INTEGER
```

```
FUNCTION CINT(value AS UINT64) AS INTEGER FAILS NumericError
```

```
FUNCTION CLAMP(value AS DOUBLE, lower AS DOUBLE, upper AS DOUBLE) AS DOUBLE
```

```
FUNCTION CLAMP(value AS INTEGER, lower AS INTEGER, upper AS INTEGER) AS INTEGER FAILS
MathError
```

```
FUNCTION Clear[K,V](BYREF items AS DICT OF K TO V) AS INTEGER
```

```
FUNCTION Clear[K](BYREF items AS SET OF K) AS INTEGER
```

```
FUNCTION Clear[T](BYREF items AS LIST OF T) AS INTEGER
```

```
SUB Clear(BYREF buffer AS ByteBuffer)
```

```
SUB Clear(BYREF builder AS StringBuilder)
```

```
SUB Clear(BYREF writer AS ByteWriter)
```

```
SUB ClearCanvas(BYREF canvas AS Canvas,color AS Color) FAILS IndexError
```

```
SUB ClearLogSinks() FAILS LoggingError
```

```
SUB ClearLogSinks(logger AS Logger) FAILS LoggingError
```

```
SUB ClearLogSinks(logger AS LoggerHandle) FAILS LoggingError
```

```
FUNCTION CloneCanvas(canvas AS Canvas) AS Canvas
```

```
FUNCTION ClonePath(path AS CanvasPath) AS CanvasPath
```

```
FUNCTION Close(db AS DATABASE) AS INTEGER FAILS SqliteError
```

```
FUNCTION Close(file AS File) AS INTEGER FAILS IoError
```

```
FUNCTION Close(logger AS Logger) AS INTEGER FAILS LoggingError
```

```
FUNCTION Close(logger AS LoggerHandle) AS INTEGER FAILS LoggingError
```

```
FUNCTION Close(socket AS TCPConnection) AS INTEGER FAILS IoError
```

```
FUNCTION Close(socket AS TCPListener) AS INTEGER FAILS IoError
```

```
FUNCTION Close(terminal AS Terminal) AS INTEGER FAILS IoError
```

```
FUNCTION Close[T](port AS SENDER OF T) AS INTEGER
```

```
SUB ClosePath(BYREF path AS CanvasPath) FAILS CanvasError
```

```
FUNCTION CloseProcessInput(process AS Process) AS INTEGER FAILS IoError
```

```
FUNCTION CloseRead(socket AS TCPConnection) AS INTEGER FAILS IoError
```

```
FUNCTION CloseWrite(socket AS TCPConnection) AS INTEGER FAILS IoError
```

```
FUNCTION CODEPOINTS(text AS STRING) AS LIST OF INTEGER
```

```
FUNCTION ColorBounds(canvas AS Canvas,minimum AS Color,maximum AS Color) AS OPTIONAL OF
PixelRect FAILS CanvasError,IndexError
```

```
FUNCTION CommandFlag(name AS STRING,short AS STRING="",description AS STRING="",global AS
BOOLEAN=FALSE) AS CommandOption
```

```
FUNCTION CommandLineArguments() AS LIST OF STRING FAILS IoError
```

```
FUNCTION CommandLineParameters() AS LIST OF STRING FAILS IoError
```

```
FUNCTION CommandOptions[T](defaults AS T) AS Command FAILS ArgumentError,StringError
```

```
FUNCTION Compare[T](left AS T, right AS T) AS CompareResult
```

```
FUNCTION CompareCaseInsensitive(left AS STRING, right AS STRING) AS CompareResult
```

```
FUNCTION CompareDate(left AS Date,right AS Date) AS INTEGER
```

```
FUNCTION CompareInstant(left AS Instant,right AS Instant) AS INTEGER
```

```
FUNCTION CompareLocalDateTime(left AS LocalDateTime,right AS LocalDateTime) AS INTEGER
```

```
FUNCTION CompareMonotonic(left AS MonotonicInstant,right AS MonotonicInstant) AS
CompareResult FAILS DateTimeError
```

```
FUNCTION CompareTime(left AS TimeOfDay,right AS TimeOfDay) AS INTEGER
```

```
FUNCTION CompileDateTimePattern(pattern AS STRING,target AS PatternTarget) AS DateTimePattern
FAILS DateTimeError
```

```
FUNCTION CompileRegex(pattern AS STRING,Flags AS STRING="",Dialect AS
RegexDialect=RegexDialect.Native,Limits AS RegexLimits=RegexLimits {}) AS Regex FAILS
RegexError
```

```
FUNCTION Complement(bits AS BitSet) AS BitSet
```

```
FUNCTION ComposeTransforms(first AS AffineTransform,after AS AffineTransform) AS
AffineTransform FAILS CanvasError
```

```
FUNCTION CompositeColor(source AS Color,destination AS Color,opacity AS DOUBLE=1.0,mode AS
BlendMode=BlendMode.SourceOver,space AS BlendSpace=BlendSpace.EncodedSRGB) AS Color FAILS
CanvasError,NumericError
```

```
FUNCTION Concat[T](left AS LIST OF T, right AS LIST OF T) AS LIST OF T
```

```
FUNCTION ConnectHTTP(url AS HTTPURL,options AS HTTPClientOptions) AS TCPConnection FAILS
IoError,TLSError,IndexError
```

```
FUNCTION CONTAINS(text AS STRING, needle AS STRING) AS BOOLEAN
```

```
FUNCTION Contains(value AS DateInterval,date AS Date) AS BOOLEAN
```

```
FUNCTION Contains(value AS InstantInterval,instant AS Instant) AS BOOLEAN
```

```
FUNCTION Contains(value AS JsonValue,key AS STRING) AS BOOLEAN FAILS JsonError
```

```
FUNCTION Contains[K,V](table AS DICT OF K TO V, key AS K) AS BOOLEAN FAILS LIKE
ValidateKey(AS K)
```

```
FUNCTION Contains[K](table AS SET OF K, key AS K) AS BOOLEAN FAILS LIKE ValidateKey(AS K)
```

```
FUNCTION Contains[T](items AS LIST OF T, value AS T) AS BOOLEAN
```

```
FUNCTION ContainsValue[K,V](table AS DICT OF K TO V, value AS V) AS BOOLEAN
```

```
FUNCTION ConvolveCanvas3x3(canvas AS Canvas,kernel AS LIST OF DOUBLE,divisor AS
DOUBLE=1.0,bias AS DOUBLE=0.0) AS Canvas FAILS CanvasError,IndexError,NumericError
```

```
FUNCTION COPYSIGN(value AS DOUBLE, sign AS DOUBLE) AS DOUBLE
```

```
FUNCTION CopyWithin(BYREF bytes AS LIST OF BYTE,source AS INTEGER,destination AS
INTEGER,count AS INTEGER) AS INTEGER FAILS IndexError
```

```
FUNCTION CopyWithin[T](BYREF items AS LIST OF T, source AS INTEGER, destination AS INTEGER,
count AS INTEGER) AS INTEGER FAILS IndexError
```

```
SUB CopyWithinCanvas(BYREF canvas AS Canvas,source AS PixelRect,x AS INTEGER,y AS
INTEGER,opacity AS DOUBLE=1.0,mode AS BlendMode=BlendMode.Copy) FAILS
CanvasError,IndexError,NumericError
```

```
FUNCTION COS(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION COSH(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION CosineSimilarity(left AS ARRAY OF BYTE,right AS ARRAY OF BYTE) AS DOUBLE FAILS
VectorDimensionError,ZeroVectorError,VectorRankError
```

```
FUNCTION CosineSimilarity(left AS ARRAY OF DOUBLE,right AS ARRAY OF DOUBLE) AS DOUBLE FAILS
VectorDimensionError,ZeroVectorError,VectorRankError
```

```
FUNCTION CosineSimilarity(left AS ARRAY OF INTEGER,right AS ARRAY OF INTEGER) AS DOUBLE FAILS
VectorDimensionError,ZeroVectorError,VectorRankError
```

```
FUNCTION CosineSimilarity(left AS ARRAY OF UINT32,right AS ARRAY OF UINT32) AS DOUBLE FAILS
VectorDimensionError,ZeroVectorError,VectorRankError
```

```
FUNCTION CosineSimilarity(left AS ARRAY OF UINT64,right AS ARRAY OF UINT64) AS DOUBLE FAILS
VectorDimensionError,ZeroVectorError,VectorRankError
```

```
FUNCTION CosineSimilarity(left AS LIST OF BYTE,right AS LIST OF BYTE) AS DOUBLE FAILS
VectorDimensionError,ZeroVectorError
```

```
FUNCTION CosineSimilarity(left AS LIST OF DOUBLE,right AS LIST OF DOUBLE) AS DOUBLE FAILS
VectorDimensionError,ZeroVectorError
```

```
FUNCTION CosineSimilarity(left AS LIST OF INTEGER,right AS LIST OF INTEGER) AS DOUBLE FAILS
VectorDimensionError,ZeroVectorError
```

```
FUNCTION CosineSimilarity(left AS LIST OF UINT32,right AS LIST OF UINT32) AS DOUBLE FAILS
VectorDimensionError,ZeroVectorError
```

```
FUNCTION CosineSimilarity(left AS LIST OF UINT64,right AS LIST OF UINT64) AS DOUBLE FAILS
VectorDimensionError,ZeroVectorError
```

```
FUNCTION Count[T](c AS Counter[T],key AS T) AS INTEGER FAILS LIKE ValidateKey(AS T)
```

```
FUNCTION CountOf[T](items AS LIST OF T, value AS T) AS INTEGER
```

```
FUNCTION CountWhere[K,V](items AS DICT OF K TO V,Predicate AS CALLABLE(K,V) AS BOOLEAN) AS
INTEGER
```

```
FUNCTION CountWhere[T](items AS LIST OF T,Predicate AS CALLABLE(T) AS BOOLEAN) AS INTEGER
```

```
FUNCTION CountWhere[T](items AS SET OF T,Predicate AS CALLABLE(T) AS BOOLEAN) AS INTEGER
```

```
FUNCTION CreateLogger() AS Logger FAILS LoggingError
```

```
FUNCTION CreateLogger(config AS LogConfig) AS Logger FAILS LoggingError
```

```
FUNCTION CreateLogger(path AS STRING) AS Logger FAILS LoggingError
```

```
FUNCTION CreateLogSink(handler AS CALLABLE(LogRecord) FAILS (Error)) AS LogSink
```

```
FUNCTION CreateLogSink[T](BYVAL context AS T,handler AS CALLABLE(BYREF T,LogRecord) FAILS
(Error)) AS LogSink
```

```
FUNCTION CreateLogSink[T](BYVAL context AS T,handler AS CALLABLE(BYREF T,LogRecord) FAILS
(Error),finish AS CALLABLE(BYREF T) FAILS (Error)) AS LogSink
```

```
FUNCTION CreateLogSink[T](BYVAL context AS T,handler AS CALLABLE(T,LogRecord) FAILS (Error))
AS LogSink
```

```
FUNCTION CreateLogSink[T](BYVAL context AS T,handler AS CALLABLE(T,LogRecord) FAILS
(Error),finish AS CALLABLE(T) FAILS (Error)) AS LogSink
```

```
FUNCTION CreatePathMask(path AS CanvasPath,width AS INTEGER,height AS INTEGER,rule AS
FillRule=FillRule.NonZero,aa AS Antialiasing=Antialiasing.Standard,maxPixels AS
INTEGER=16777216) AS Canvas FAILS CanvasError,IndexError,NumericError
```

```
FUNCTION CreateTextLayout(font AS Font,text AS STRING,options AS TextLayoutOptions) AS
TextLayout FAILS CanvasError,IndexError,NumericError
```

```
FUNCTION CreateTextLayout(fonts AS FontSet,text AS STRING,options AS TextLayoutOptions) AS
TextLayout FAILS CanvasError,IndexError,NumericError
```

```
FUNCTION CropCanvas(canvas AS Canvas,region AS PixelRect) AS Canvas FAILS
CanvasError,IndexError,NumericError
```

```
SUB CubicTo(BYREF path AS CanvasPath,firstX AS DOUBLE,firstY AS DOUBLE,secondX AS
DOUBLE,secondY AS DOUBLE,x AS DOUBLE,y AS DOUBLE) FAILS CanvasError
```

```
FUNCTION CUINT32(value AS INTEGER) AS UINT32 FAILS NumericError
```

```
FUNCTION CUINT32(value AS UINT64) AS UINT32 FAILS NumericError
```

```
FUNCTION CUINT64(value AS INTEGER) AS UINT64 FAILS NumericError
```

D

```
FUNCTION DateFromEpochDays(days AS INTEGER) AS Date FAILS DateTimeError
```

```
FUNCTION DatePart(value AS LocalDateTime) AS Date
```

```
FUNCTION DayOfMonth(value AS Date) AS INTEGER
```

```
FUNCTION DayOfYear(value AS Date) AS INTEGER FAILS DateTimeError
```

```
FUNCTION DaysBetween(start AS Date,finish AS Date) AS INTEGER
```

```
FUNCTION DaysInMonth(year AS INTEGER,month AS INTEGER) AS INTEGER FAILS DateTimeError
```

```
FUNCTION DaysInMonth(year AS INTEGER,month AS Month) AS INTEGER FAILS DateTimeError
```

```
FUNCTION DaysInYear(year AS INTEGER) AS INTEGER
```

```
FUNCTION DeadlineAfter(clock AS Clock,timeout AS Duration) AS Deadline FAILS DateTimeError
```

```
FUNCTION DecodeAnimation(bytes AS LIST OF BYTE,maxPixels AS INTEGER=16777216,maxFrames AS
INTEGER=1000,maxDecodedBytes AS INTEGER=268435456,maxEncodedBytes AS INTEGER=67108864) AS
Animation FAILS CanvasError,IndexError,NumericError,StringError
```

```
FUNCTION DecodeAPNG(bytes AS LIST OF BYTE,maxPixels AS INTEGER=16777216,maxFrames AS
INTEGER=1000,maxDecodedBytes AS INTEGER=268435456,maxEncodedBytes AS INTEGER=67108864) AS
Animation FAILS CanvasError,IndexError,NumericError,StringError
```

```
FUNCTION DecodeBMP(data AS LIST OF BYTE,maxPixels AS INTEGER=16777216,maxEncodedBytes AS
INTEGER=67108864) AS Canvas FAILS CanvasError,IndexError,NumericError
```

```
FUNCTION DecodeGIF(bytes AS LIST OF BYTE,maxPixels AS INTEGER=16777216,maxFrames AS
INTEGER=1000,maxDecodedBytes AS INTEGER=268435456,maxEncodedBytes AS INTEGER=67108864) AS
Animation FAILS CanvasError,IndexError,NumericError,StringError
```

```
FUNCTION DecodeImage(bytes AS LIST OF BYTE,maxPixels AS INTEGER=16777216,maxEncodedBytes AS
INTEGER=67108864) AS Canvas FAILS CanvasError,IndexError,NumericError,StringError
```

```
FUNCTION DecodeJPEG(bytes AS LIST OF BYTE,maxPixels AS INTEGER=16777216,maxEncodedBytes AS
INTEGER=67108864,maxWorkingBytes AS INTEGER=268435456) AS Canvas FAILS
CanvasError,IndexError,NumericError
```

```
FUNCTION DecodeJSON[T](bytes AS LIST OF BYTE) AS T FAILS JsonError,LIKE __JsonDecodeErrors(AS
T)
```

```
FUNCTION DecodeJSON[T](bytes AS LIST OF BYTE,codec AS JsonCodec) AS T FAILS JsonError,LIKE
codec, LIKE __JsonDecodeErrors(AS T)
```

```
FUNCTION DecodeJSON[T](text AS STRING) AS T FAILS JsonError,LIKE __JsonDecodeErrors(AS T)
```

```
FUNCTION DecodeJSON[T](text AS STRING,codec AS JsonCodec) AS T FAILS JsonError,LIKE codec,
LIKE __JsonDecodeErrors(AS T)
```

```
FUNCTION DecodePNG(data AS LIST OF BYTE,maxPixels AS INTEGER=16777216,maxEncodedBytes AS
INTEGER=67108864) AS Canvas FAILS CanvasError,IndexError,NumericError,StringError
```

```
FUNCTION DecoderInfo(decoder AS TextDecoder) AS TextConversionInfo
```

```
FUNCTION DecodeText(bytes AS LIST OF BYTE,encoding AS TextEncoding,errors AS
TextErrorPolicy=TextErrorPolicy.Strict,bom AS
TextDecodeBom=TextDecodeBom.Preserve,maxOutputBytes AS INTEGER=67108864) AS STRING FAILS
TextEncodingError,IndexError,NumericError,StringError
```

```
FUNCTION DecodeTextChunk(BYREF decoder AS TextDecoder,bytes AS LIST OF BYTE,final AS
BOOLEAN=FALSE) AS STRING FAILS TextEncodingError,IndexError,NumericError,StringError
```

```
FUNCTION DecodeTextWithInfo(bytes AS LIST OF BYTE,encoding AS TextEncoding,errors AS
TextErrorPolicy=TextErrorPolicy.Strict,bom AS
TextDecodeBom=TextDecodeBom.Preserve,maxOutputBytes AS INTEGER=67108864) AS DecodedText FAILS
TextEncodingError,IndexError,NumericError,StringError
```

```
FUNCTION DecodeURLComponent(text AS STRING,form AS BOOLEAN=FALSE) AS STRING FAILS
HTTPError,StringError
```

```
FUNCTION DecodeUtf8(bytes AS LIST OF BYTE) AS STRING FAILS StringError
```

```
FUNCTION DecodeUtf8(bytes AS LIST OF INTEGER) AS STRING FAILS StringError
```

```
FUNCTION DecodeVP8(bytes AS LIST OF BYTE,maxPixels AS INTEGER=16777216,maxWorkingBytes AS
INTEGER=268435456) AS Canvas FAILS CanvasError,IndexError,NumericError
```

```
FUNCTION DecodeVP8L(bytes AS LIST OF BYTE,maxPixels AS INTEGER=16777216,maxWorkingBytes AS
INTEGER=268435456) AS Canvas FAILS CanvasError,IndexError,NumericError
```

```
FUNCTION DecodeWebP(bytes AS LIST OF BYTE,maxPixels AS INTEGER=16777216,maxEncodedBytes AS
INTEGER=67108864,maxWorkingBytes AS INTEGER=268435456) AS Canvas FAILS
CanvasError,IndexError,NumericError,StringError
```

```
FUNCTION DecodeWebPAnimation(bytes AS LIST OF BYTE,maxPixels AS INTEGER=16777216,maxFrames AS
INTEGER=1000,maxDecodedBytes AS INTEGER=268435456,maxEncodedBytes AS INTEGER=67108864) AS
Animation FAILS CanvasError,IndexError,NumericError,StringError
```

```
FUNCTION DefaultLogConfig() AS LogConfig
```

```
FUNCTION DefaultLogger() AS LoggerHandle
```

```
FUNCTION DEGREES(radians AS DOUBLE) AS DOUBLE
```

```
FUNCTION Dequeue[T](BYREF q AS Queue[T]) AS T FAILS EmptyCollection
```

```
FUNCTION DialTCP(address AS IPAddress,port AS INTEGER) AS TCPConnection FAILS IoError
```

```
FUNCTION DialTCP(address AS IPAddress,port AS INTEGER,timeout AS Duration) AS TCPConnection
FAILS IoError
```

```
FUNCTION DialTCP(host AS STRING,port AS INTEGER) AS TCPConnection FAILS IoError
```

```
FUNCTION DialTCP(host AS STRING,port AS INTEGER,timeout AS Duration) AS TCPConnection FAILS
IoError
```

```
FUNCTION DialTLS(host AS STRING,port AS INTEGER) AS TCPConnection FAILS IoError,TLSError
```

```
FUNCTION DialTLS(host AS STRING,port AS INTEGER,options AS TLSClientOptions) AS TCPConnection
FAILS IoError,TLSError
```

```
FUNCTION DialTLS(host AS STRING,port AS INTEGER,options AS TLSClientOptions,timeout AS
Duration) AS TCPConnection FAILS IoError,TLSError
```

```
FUNCTION Difference[K](left AS SET OF K, right AS SET OF K) AS SET OF K
```

```
FUNCTION Digest(bytes AS LIST OF BYTE) AS LIST OF BYTE
```

```
FUNCTION DiscoverFonts(directories AS LIST OF STRING,maxFiles AS INTEGER=10000,maxFontBytes
AS INTEGER=33554432) AS FontCatalog FAILS CanvasError,IndexError,NumericError,StringError
```

```
FUNCTION DiscoverSystemFonts(maxFiles AS INTEGER=10000) AS FontCatalog FAILS
CanvasError,IndexError,NumericError,StringError
```

```
FUNCTION Display(value AS BOOLEAN) AS STRING
```

```
FUNCTION Display(value AS BYTE) AS STRING
```

```
FUNCTION Display(value AS DOUBLE) AS STRING
```

```
FUNCTION Display(value AS INTEGER) AS STRING
```

```
FUNCTION Display(value AS STRING) AS STRING
```

```
FUNCTION Display(value AS UINT32) AS STRING
```

```
FUNCTION Display(value AS UINT64) AS STRING
```

```
FUNCTION Display(value AS UUID) AS STRING
```

```
FUNCTION Display[T](value AS T) AS STRING
```

```
FUNCTION Distance(left AS ARRAY OF BYTE,right AS ARRAY OF BYTE) AS DOUBLE FAILS
VectorDimensionError,VectorRankError
```

```
FUNCTION Distance(left AS ARRAY OF DOUBLE,right AS ARRAY OF DOUBLE) AS DOUBLE FAILS
VectorDimensionError,VectorRankError
```

```
FUNCTION Distance(left AS ARRAY OF INTEGER,right AS ARRAY OF INTEGER) AS DOUBLE FAILS
VectorDimensionError,VectorRankError
```

```
FUNCTION Distance(left AS ARRAY OF UINT32,right AS ARRAY OF UINT32) AS DOUBLE FAILS
VectorDimensionError,VectorRankError
```

```
FUNCTION Distance(left AS ARRAY OF UINT64,right AS ARRAY OF UINT64) AS DOUBLE FAILS
VectorDimensionError,VectorRankError
```

```
FUNCTION Distance(left AS LIST OF BYTE,right AS LIST OF BYTE) AS DOUBLE FAILS
VectorDimensionError
```

```
FUNCTION Distance(left AS LIST OF DOUBLE,right AS LIST OF DOUBLE) AS DOUBLE FAILS
VectorDimensionError
```

```
FUNCTION Distance(left AS LIST OF INTEGER,right AS LIST OF INTEGER) AS DOUBLE FAILS
VectorDimensionError
```

```
FUNCTION Distance(left AS LIST OF UINT32,right AS LIST OF UINT32) AS DOUBLE FAILS
VectorDimensionError
```

```
FUNCTION Distance(left AS LIST OF UINT64,right AS LIST OF UINT64) AS DOUBLE FAILS
VectorDimensionError
```

```
FUNCTION DivideDuration(value AS Duration,divisor AS INTEGER,rounding AS
RoundingMode=RoundingMode.RejectLoss) AS Duration FAILS DateTimeError
```

```
FUNCTION Dot(left AS ARRAY OF BYTE,right AS ARRAY OF BYTE) AS UINT64 FAILS
VectorDimensionError,VectorRankError,NumericError
```

```
FUNCTION Dot(left AS ARRAY OF DOUBLE,right AS ARRAY OF DOUBLE) AS DOUBLE FAILS
VectorDimensionError,VectorRankError
```

```
FUNCTION Dot(left AS ARRAY OF INTEGER,right AS ARRAY OF INTEGER) AS INTEGER FAILS
VectorDimensionError,VectorRankError,NumericError
```

```
FUNCTION Dot(left AS ARRAY OF UINT32,right AS ARRAY OF UINT32) AS UINT64 FAILS
VectorDimensionError,VectorRankError,NumericError
```

```
FUNCTION Dot(left AS ARRAY OF UINT64,right AS ARRAY OF UINT64) AS UINT64 FAILS
VectorDimensionError,VectorRankError,NumericError
```

```
FUNCTION Dot(left AS LIST OF BYTE,right AS LIST OF BYTE) AS UINT64 FAILS
VectorDimensionError,NumericError
```

```
FUNCTION Dot(left AS LIST OF DOUBLE,right AS LIST OF DOUBLE) AS DOUBLE FAILS
VectorDimensionError
```

```
FUNCTION Dot(left AS LIST OF INTEGER,right AS LIST OF INTEGER) AS INTEGER FAILS
VectorDimensionError,NumericError
```

```
FUNCTION Dot(left AS LIST OF UINT32,right AS LIST OF UINT32) AS UINT64 FAILS
VectorDimensionError,NumericError
```

```
FUNCTION Dot(left AS LIST OF UINT64,right AS LIST OF UINT64) AS UINT64 FAILS
VectorDimensionError,NumericError
```

```
FUNCTION DoubleBits(value AS DOUBLE) AS INTEGER
```

```
FUNCTION DoubleFromBits(bits AS INTEGER) AS DOUBLE
```

```
SUB DrawAnimationFrame(BYREF canvas AS Canvas,animation AS Animation,index AS INTEGER,x AS
INTEGER=0,y AS INTEGER=0,opacity AS DOUBLE=1.0,mode AS BlendMode=BlendMode.SourceOver,space
AS BlendSpace=BlendSpace.EncodedSRGB) FAILS CanvasError,IndexError,NumericError
```

```
SUB DrawBitmapText(BYREF canvas AS Canvas,text AS STRING,x AS INTEGER,y AS INTEGER,color AS
Color,scale AS INTEGER=1,strictGlyphs AS BOOLEAN=FALSE) FAILS
CanvasError,IndexError,NumericError
```

```
SUB DrawCircle(BYREF canvas AS Canvas,cx AS DOUBLE,cy AS DOUBLE,radius AS DOUBLE,stroke AS
Stroke) FAILS CanvasError,IndexError,NumericError
```

```
SUB DrawDashedPath(BYREF canvas AS Canvas,path AS CanvasPath,stroke AS Stroke,dashes AS LIST
OF DOUBLE,dashOffset AS DOUBLE=0.0,mode AS BlendMode=BlendMode.SourceOver,space AS
BlendSpace=BlendSpace.EncodedSRGB) FAILS CanvasError,IndexError,NumericError
```

```
SUB DrawDashedPolyline(BYREF canvas AS Canvas,points AS LIST OF Point,stroke AS Stroke,dashes
AS LIST OF DOUBLE,dashOffset AS DOUBLE=0.0,closed AS BOOLEAN=FALSE) FAILS
CanvasError,IndexError,NumericError
```

```
SUB DrawEllipse(BYREF canvas AS Canvas,cx AS DOUBLE,cy AS DOUBLE,rx AS DOUBLE,ry AS
DOUBLE,stroke AS Stroke) FAILS CanvasError,IndexError,NumericError
```

```
SUB DrawImage(BYREF destination AS Canvas,source AS Canvas,x AS INTEGER,y AS INTEGER,opacity
AS DOUBLE=1.0,mode AS BlendMode=BlendMode.SourceOver,space AS
BlendSpace=BlendSpace.EncodedSRGB) FAILS CanvasError,IndexError,NumericError
```

```
SUB DrawLine(BYREF canvas AS Canvas,x0 AS DOUBLE,y0 AS DOUBLE,x1 AS DOUBLE,y1 AS
DOUBLE,stroke AS Stroke) FAILS CanvasError,IndexError,NumericError
```

```
SUB DrawPath(BYREF canvas AS Canvas,path AS CanvasPath,stroke AS Stroke,mode AS
BlendMode=BlendMode.SourceOver,space AS BlendSpace=BlendSpace.EncodedSRGB) FAILS
CanvasError,IndexError,NumericError
```

```
SUB DrawPolyline(BYREF canvas AS Canvas,points AS LIST OF Point,stroke AS Stroke,closed AS
BOOLEAN=FALSE) FAILS CanvasError,IndexError,NumericError
```

```
SUB DrawRect(BYREF canvas AS Canvas,x AS DOUBLE,y AS DOUBLE,width AS DOUBLE,height AS
DOUBLE,stroke AS Stroke) FAILS CanvasError,IndexError,NumericError
```

```
SUB DrawRoundedRect(BYREF canvas AS Canvas,x AS DOUBLE,y AS DOUBLE,width AS DOUBLE,height AS
DOUBLE,radius AS DOUBLE,stroke AS Stroke) FAILS CanvasError,IndexError,NumericError
```

```
SUB DrawText(BYREF canvas AS Canvas,font AS Font,text AS STRING,x AS DOUBLE,y AS DOUBLE,style
AS TextStyle) FAILS CanvasError,IndexError,NumericError
```

```
SUB DrawTextLayout(BYREF canvas AS Canvas,layout AS TextLayout,x AS DOUBLE,y AS DOUBLE,color
AS Color,aa AS Antialiasing=Antialiasing.Standard,mode AS
BlendMode=BlendMode.SourceOver,space AS BlendSpace=BlendSpace.EncodedSRGB) FAILS
CanvasError,IndexError,NumericError
```

```
FUNCTION DurationMicroseconds(value AS Duration,rounding AS
RoundingMode=RoundingMode.RejectLoss) AS INTEGER FAILS DateTimeError
```

```
FUNCTION DurationMilliseconds(value AS Duration,rounding AS
RoundingMode=RoundingMode.RejectLoss) AS INTEGER FAILS DateTimeError
```

```
FUNCTION DurationNanoseconds(value AS Duration) AS INTEGER FAILS DateTimeError
```

```
FUNCTION DurationSeconds(value AS Duration) AS INTEGER
```

E

```
FUNCTION E() AS DOUBLE
```

```
FUNCTION EdgeCanvas(canvas AS Canvas) AS Canvas FAILS CanvasError,IndexError,NumericError
```

```
FUNCTION ElapsedBetween(start AS Instant,finish AS Instant) AS Duration FAILS DateTimeError
```

```
FUNCTION ElapsedBetween(start AS MonotonicInstant,finish AS MonotonicInstant) AS Duration
FAILS DateTimeError
```

```
FUNCTION EmbossCanvas(canvas AS Canvas) AS Canvas FAILS CanvasError,IndexError,NumericError
```

```
FUNCTION EncodeAnimation(animation AS Animation,format AS
ImageFormat=ImageFormat.APNG,maxEncodedBytes AS INTEGER=67108864) AS LIST OF BYTE FAILS
CanvasError,IndexError,NumericError,StringError
```

```
FUNCTION EncodeAPNG(animation AS Animation,level AS INTEGER=6,maxEncodedBytes AS
INTEGER=67108864) AS LIST OF BYTE FAILS CanvasError,IndexError,NumericError,StringError
```

```
FUNCTION EncodeBMP(canvas AS Canvas,maxEncodedBytes AS INTEGER=67108864) AS LIST OF BYTE
FAILS CanvasError,IndexError,NumericError
```

```
FUNCTION EncodeGIF(canvas AS Canvas,quantize AS BOOLEAN=TRUE,maxEncodedBytes AS
INTEGER=67108864) AS LIST OF BYTE FAILS CanvasError,IndexError,NumericError
```

```
FUNCTION EncodeGIFAnimation(animation AS Animation,quantize AS BOOLEAN=TRUE,maxEncodedBytes
AS INTEGER=67108864) AS LIST OF BYTE FAILS CanvasError,IndexError,NumericError
```

```
FUNCTION EncodeImage(canvas AS Canvas,format AS ImageFormat=ImageFormat.PNG) AS LIST OF BYTE
FAILS CanvasError,IndexError,NumericError,StringError
```

```
FUNCTION EncodeJPEG(canvas AS Canvas,quality AS INTEGER=90,grayscale AS
BOOLEAN=FALSE,maxEncodedBytes AS INTEGER=67108864) AS LIST OF BYTE FAILS
CanvasError,IndexError,NumericError
```

```
FUNCTION EncodeJPEGOnBackground(canvas AS Canvas,background AS Color,quality AS
INTEGER=90,grayscale AS BOOLEAN=FALSE,maxEncodedBytes AS INTEGER=67108864) AS LIST OF BYTE
FAILS CanvasError,IndexError,NumericError
```

```
FUNCTION EncodeJSON[T](value AS T) AS STRING FAILS JsonError,LIKE __JsonEncodeErrors(AS T)
```

```
FUNCTION EncodeJSON[T](value AS T,codec AS JsonCodec) AS STRING FAILS JsonError,LIKE codec,
LIKE __JsonEncodeErrors(AS T)
```

```
FUNCTION EncodePNG(canvas AS Canvas,level AS INTEGER=1,filter AS
PNGFilter=PNGFilter.Adaptive,maxEncodedBytes AS INTEGER=67108864) AS LIST OF BYTE FAILS
CanvasError,IndexError,NumericError,StringError
```

```
FUNCTION EncoderInfo(encoder AS TextEncoder) AS TextConversionInfo
```

```
FUNCTION EncodeText(text AS STRING,encoding AS TextEncoding,errors AS
TextErrorPolicy=TextErrorPolicy.Strict,bom AS TextEncodeBom=TextEncodeBom.Omit,maxOutputBytes
AS INTEGER=67108864) AS LIST OF BYTE FAILS
TextEncodingError,IndexError,NumericError,StringError
```

```
FUNCTION EncodeTextChunk(BYREF encoder AS TextEncoder,text AS STRING,final AS BOOLEAN=FALSE)
AS LIST OF BYTE FAILS TextEncodingError,IndexError,NumericError,StringError
```

```
FUNCTION EncodeTextWithInfo(text AS STRING,encoding AS TextEncoding,errors AS
TextErrorPolicy=TextErrorPolicy.Strict,bom AS TextEncodeBom=TextEncodeBom.Omit,maxOutputBytes
AS INTEGER=67108864) AS EncodedText FAILS
TextEncodingError,IndexError,NumericError,StringError
```

```
FUNCTION EncodeURLComponent(text AS STRING) AS STRING FAILS StringError
```

```
FUNCTION EncodeUtf8(text AS STRING) AS LIST OF BYTE
```

```
FUNCTION EncodeWebP(canvas AS Canvas,maxEncodedBytes AS INTEGER=67108864,maxWorkingBytes AS
INTEGER=268435456) AS LIST OF BYTE FAILS CanvasError,IndexError,NumericError
```

```
FUNCTION EncodeWebPAnimation(animation AS Animation,maxEncodedBytes AS
INTEGER=67108864,maxWorkingBytes AS INTEGER=268435456) AS LIST OF BYTE FAILS
CanvasError,IndexError,NumericError
```

```
FUNCTION EncodeWebPLossless(canvas AS Canvas,maxEncodedBytes AS
INTEGER=67108864,maxWorkingBytes AS INTEGER=268435456) AS LIST OF BYTE FAILS
CanvasError,IndexError,NumericError
```

```
FUNCTION EndOfMonth(value AS Date) AS Date FAILS DateTimeError
```

```
FUNCTION EndOfStream[T](reader AS BufferedReader[T]) AS BOOLEAN WHERE T IS RESOURCE
```

```
FUNCTION EndOfYear(value AS Date) AS Date FAILS DateTimeError
```

```
FUNCTION Endpoint[T](reader AS BufferedReader[T]) AS BORROW T FROM reader WHERE T IS RESOURCE
```

```
FUNCTION Endpoint[T](writer AS BufferedWriter[T]) AS BORROW T FROM writer WHERE T IS RESOURCE
```

```
FUNCTION ENDSWITH(text AS STRING, suffix AS STRING) AS BOOLEAN
```

```
SUB Enqueue[T](BYREF q AS Queue[T],BYVAL value AS T)
```

```
FUNCTION EnterTerminalScreen(terminal AS Terminal) AS INTEGER FAILS IoError
```

```
FUNCTION EpochDays(value AS Date) AS INTEGER
```

```
FUNCTION Equal[T](left AS T, right AS T) AS BOOLEAN
```

```
FUNCTION ErrorKindName(kind AS DateTimeErrorKind) AS STRING
```

```
FUNCTION EscapeURLComponent(text AS STRING,form AS BOOLEAN=FALSE) AS STRING
```

```
FUNCTION EscapeURLPath(text AS STRING) AS STRING
```

```
FUNCTION EuclideanMod(value AS INTEGER,divisor AS INTEGER) AS INTEGER FAILS NumericError
```

```
SUB Exclude(BYREF bits AS BitSet,index AS INTEGER) FAILS IndexError
```

```
FUNCTION EXP(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION EXP2(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION EXPM1(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION Extend(BYREF buffer AS ByteBuffer,bytes AS LIST OF BYTE) AS INTEGER FAILS IndexError
```

```
FUNCTION Extend(BYREF buffer AS ByteBuffer,bytes AS LIST OF BYTE,start AS INTEGER,count AS
INTEGER) AS INTEGER FAILS IndexError
```

```
FUNCTION Extend[T](BYREF items AS LIST OF T, values AS LIST OF T) AS INTEGER
```

```
FUNCTION Extract[K,V](BYREF d AS SortedDictionary[K,V],key AS K) AS OPTIONAL OF V FAILS
IndexError,LIKE __ValueOrderCheck(AS K),LIKE ValidateKey(AS K)
```

```
FUNCTION Extract[K,V](BYREF m AS MultiMap[K,V],key AS K) AS OPTIONAL OF LIST OF V FAILS LIKE
ValidateKey(AS K)
```

```
FUNCTION Extract[K,V](BYREF table AS DICT OF K TO V, key AS K) AS OPTIONAL OF V FAILS LIKE
ValidateKey(AS K)
```

```
FUNCTION ExtractAt[K,V](BYREF m AS MultiMap[K,V],key AS K,index AS INTEGER) AS V FAILS
MissingKey,IndexError,LIKE ValidateKey(AS K)
```

```
FUNCTION ExtractAt[T](BYREF items AS LIST OF T,index AS INTEGER) AS T FAILS IndexError
```

F

```
SUB FillCircle(BYREF canvas AS Canvas,cx AS DOUBLE,cy AS DOUBLE,radius AS DOUBLE,color AS
Color,aa AS Antialiasing=Antialiasing.Standard) FAILS CanvasError,IndexError,NumericError
```

```
SUB FillEllipse(BYREF canvas AS Canvas,cx AS DOUBLE,cy AS DOUBLE,rx AS DOUBLE,ry AS
DOUBLE,color AS Color,aa AS Antialiasing=Antialiasing.Standard) FAILS
CanvasError,IndexError,NumericError
```

```
SUB FillPath(BYREF canvas AS Canvas,path AS CanvasPath,color AS Color,rule AS
FillRule=FillRule.NonZero,aa AS Antialiasing=Antialiasing.Standard,mode AS
BlendMode=BlendMode.SourceOver,space AS BlendSpace=BlendSpace.EncodedSRGB) FAILS
CanvasError,IndexError,NumericError
```

```
SUB FillPolygon(BYREF canvas AS Canvas,points AS LIST OF Point,color AS Color,aa AS
Antialiasing=Antialiasing.Standard) FAILS CanvasError,IndexError,NumericError
```

```
SUB FillRect(BYREF canvas AS Canvas,x AS DOUBLE,y AS DOUBLE,width AS DOUBLE,height AS
DOUBLE,color AS Color,aa AS Antialiasing=Antialiasing.Standard,mode AS
```

```
BlendMode=BlendMode.SourceOver,space AS BlendSpace=BlendSpace.EncodedSRGB) FAILS
CanvasError,IndexError,NumericError
```

```
SUB FillRoundedRect(BYREF canvas AS Canvas,x AS DOUBLE,y AS DOUBLE,width AS DOUBLE,height AS
DOUBLE,radius AS DOUBLE,color AS Color,aa AS Antialiasing=Antialiasing.Standard) FAILS
CanvasError,IndexError,NumericError
```

```
FUNCTION FillSecureRandom(BYREF bytes AS LIST OF BYTE) AS INTEGER FAILS IoError
```

```
FUNCTION Filter[K,V](items AS DICT OF K TO V,Predicate AS CALLABLE(K,V) AS BOOLEAN) AS DICT
OF K TO V FAILS LIKE ValidateKey(AS K) WHERE V IS COPYABLE
```

```
FUNCTION Filter[T](items AS LIST OF T,Predicate AS CALLABLE(T) AS BOOLEAN) AS LIST OF T WHERE
T IS COPYABLE
```

```
FUNCTION Filter[T](items AS SET OF T,Predicate AS CALLABLE(T) AS BOOLEAN) AS SET OF T WHERE T
IS COPYABLE
```

```
FUNCTION Find(value AS JsonValue,key AS STRING) AS OPTIONAL OF JsonValue FAILS JsonError
```

```
FUNCTION Find[K,V](m AS MultiMap[K,V],key AS K) AS BORROW OPTIONAL OF LIST OF V FROM m FAILS
LIKE ValidateKey(AS K) WHERE V IS RESOURCE
```

```
FUNCTION Find[K,V](m AS MultiMap[K,V],key AS K) AS OPTIONAL OF LIST OF V FAILS LIKE
ValidateKey(AS K) WHERE V IS COPYABLE
```

```
FUNCTION Find[K,V](table AS DICT OF K TO V, key AS K) AS OPTIONAL OF V FAILS LIKE
ValidateKey(AS K) WHERE V IS COPYABLE
```

```
FUNCTION Find[K,V](table AS DICT OF K TO V,key AS K) AS BORROW OPTIONAL OF V FROM table FAILS
LIKE ValidateKey(AS K) WHERE V IS RESOURCE
```

```
FUNCTION FindFont(catalog AS FontCatalog,family AS STRING,weight AS INTEGER=400,italic AS
BOOLEAN=FALSE,allowSubstitution AS BOOLEAN=FALSE) AS FontMatch FAILS
CanvasError,IndexError,NumericError,StringError
```

```
FUNCTION FindTimeZone(value AS ZoneDatabase,id AS STRING) AS TimeZone FAILS DateTimeError
```

```
FUNCTION FindWhere[T](items AS LIST OF T,Predicate AS CALLABLE(T) AS BOOLEAN) AS OPTIONAL OF
T WHERE T IS COPYABLE
```

```
FUNCTION Finish[T](BYVAL writer AS BufferedWriter[T]) AS T FAILS IoError WHERE T IS RESOURCE
```

```
FUNCTION FinishHTTPRequest(connection AS TCPConnection,BYREF exchange AS HTTPExchange) AS
INTEGER FAILS IoError,HTTPError,StringError
```

```
FUNCTION FinishHTTPRequest(connection AS TCPConnection,BYREF exchange AS
HTTPExchange,trailers AS LIST OF HTTPHeader) AS INTEGER FAILS IoError,HTTPError,StringError
```

```
FUNCTION FinishHTTPResponse(connection AS TCPConnection,BYREF exchange AS HTTPExchange) AS
INTEGER FAILS IoError,HTTPError,StringError
```

```
FUNCTION FinishHTTPResponse(connection AS TCPConnection,BYREF exchange AS
HTTPExchange,trailers AS LIST OF HTTPHeader) AS INTEGER FAILS IoError,HTTPError,StringError
```

```
FUNCTION FinishTextDecoder(BYREF decoder AS TextDecoder) AS STRING FAILS
TextEncodingError,IndexError,NumericError,StringError
```

```
FUNCTION FinishTextEncoder(BYREF encoder AS TextEncoder) AS LIST OF BYTE FAILS
TextEncodingError,IndexError,NumericError,StringError
```

```
FUNCTION FinishTLS(connection AS TCPConnection,timeout AS Duration) AS INTEGER FAILS
IoError,TLSError
```

```
FUNCTION Finite(value AS DOUBLE) AS DOUBLE FAILS NonFiniteError
```

```
FUNCTION FitCanvas(source AS Canvas,width AS INTEGER,height AS INTEGER,fit AS
ImageFit=ImageFit.Contain,sampling AS Sampling=Sampling.Bilinear,space AS
BlendSpace=BlendSpace.EncodedSRGB,maxPixels AS INTEGER=16777216) AS Canvas FAILS
CanvasError,IndexError,NumericError
```

```
FUNCTION FIX(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION FixedOffsetZone(offset AS UtcOffset) AS TimeZone FAILS DateTimeError
```

```
FUNCTION FlattenCanvas(canvas AS Canvas,background AS Color,space AS
BlendSpace=BlendSpace.EncodedSRGB) AS Canvas FAILS CanvasError,IndexError,NumericError
```

```
SUB Flip(BYREF bits AS BitSet,index AS INTEGER) FAILS IndexError
```

```
FUNCTION FlipCanvas(canvas AS Canvas,horizontal AS BOOLEAN=TRUE,vertical AS BOOLEAN=FALSE) AS
Canvas FAILS CanvasError,IndexError,NumericError
```

```
FUNCTION FLOOR(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION FloorDiv(value AS INTEGER,divisor AS INTEGER) AS INTEGER FAILS NumericError
```

```
FUNCTION FloorToInteger(value AS DOUBLE) AS INTEGER FAILS NumericError
```

```
FUNCTION Flush(file AS File) AS INTEGER FAILS IoError
```

```
FUNCTION Flush(logger AS Logger) AS INTEGER FAILS LoggingError
```

```
FUNCTION Flush(logger AS LoggerHandle) AS INTEGER FAILS LoggingError
```

```
FUNCTION Flush[T](BYREF writer AS BufferedWriter[T]) AS INTEGER FAILS IoError WHERE T IS
RESOURCE
```

```
FUNCTION FMA(x AS DOUBLE, y AS DOUBLE, z AS DOUBLE) AS DOUBLE
```

```
FUNCTION FMOD(x AS DOUBLE, y AS DOUBLE) AS DOUBLE
```

```
FUNCTION FontCatalogCount(catalog AS FontCatalog) AS INTEGER
```

```
FUNCTION FontCatalogEntry(catalog AS FontCatalog,index AS INTEGER) AS FontInfo FAILS
CanvasError,IndexError
```

```
FUNCTION FontDiscoveryWarnings(catalog AS FontCatalog) AS LIST OF STRING
```

```
FUNCTION FontFaceCount(bytes AS LIST OF BYTE) AS INTEGER FAILS CanvasError,IndexError
```

```
FUNCTION FontHasGlyph(font AS Font,scalar AS INTEGER) AS BOOLEAN FAILS CanvasError,IndexError
```

```
FUNCTION FontSetCount(fonts AS FontSet) AS INTEGER
```

```
FUNCTION FontSetHasGlyph(fonts AS FontSet,scalar AS INTEGER) AS BOOLEAN FAILS
CanvasError,IndexError
```

```
FUNCTION FORMAT(template AS STRING, values AS LIST OF STRING) AS STRING FAILS StringError
```

```
FUNCTION FormatDate(value AS Date) AS STRING FAILS DateTimeError
```

```
FUNCTION FormatDateTimeAmount(value AS DateTimeAmount) AS STRING FAILS DateTimeError
```

```
FUNCTION FormatDuration(value AS Duration) AS STRING FAILS DateTimeError
```

```
FUNCTION FormatHttpDate(value AS Instant) AS STRING FAILS DateTimeError
```

```
FUNCTION FormatInstant(value AS Instant) AS STRING FAILS DateTimeError
```

```
FUNCTION FormatInstant(value AS Instant,options AS FormatOptions) AS STRING FAILS
DateTimeError
```

```
FUNCTION FormatIsoWeekDate(value AS IsoWeekDate) AS STRING FAILS DateTimeError
```

```
FUNCTION FormatLocalDateTime(value AS LocalDateTime) AS STRING FAILS DateTimeError
```

```
FUNCTION FormatLocalDateTime(value AS LocalDateTime,options AS FormatOptions) AS STRING FAILS
DateTimeError
```

```
FUNCTION FormatLogRecord(record AS LogRecord,pattern AS STRING) AS STRING FAILS LoggingError
```

```
FUNCTION FormatMetadata[M](graph AS TypeGraph,node AS INTEGER,field AS STRING="") AS OPTIONAL
OF M FAILS JsonError
```

```
FUNCTION FormatMetadata[T,M](witness AS OPTIONAL OF T,field AS STRING="") AS OPTIONAL OF M
```

```
FUNCTION FormatMonthDay(value AS MonthDay) AS STRING FAILS DateTimeError
```

```
FUNCTION FormatOffset(value AS UtcOffset) AS STRING FAILS DateTimeError
```

```
FUNCTION FormatOffsetDateTime(value AS OffsetDateTime) AS STRING FAILS DateTimeError
```

```
FUNCTION FormatOffsetDateTime(value AS OffsetDateTime,options AS FormatOptions) AS STRING
FAILS DateTimeError
```

```
FUNCTION FormatOffsetTime(value AS OffsetTime) AS STRING FAILS DateTimeError
```

```
FUNCTION FormatPeriod(value AS Period) AS STRING FAILS DateTimeError
```

```
FUNCTION FormatRfc3339(value AS Instant) AS STRING FAILS DateTimeError
```

```
FUNCTION FormatRfc3339(value AS Instant,options AS FormatOptions) AS STRING FAILS
DateTimeError
```

```
FUNCTION FormatRfc3339(value AS OffsetDateTime) AS STRING FAILS DateTimeError
```

```
FUNCTION FormatRfc3339(value AS OffsetDateTime,options AS FormatOptions) AS STRING FAILS
DateTimeError
```

```
FUNCTION FormatRfc9557(value AS ZonedDateTime) AS STRING FAILS DateTimeError
```

```
FUNCTION FormatRfc9557(value AS ZonedDateTime,options AS FormatOptions) AS STRING FAILS
DateTimeError
```

```
FUNCTION FormatTime(value AS TimeOfDay) AS STRING FAILS DateTimeError
```

```
FUNCTION FormatTime(value AS TimeOfDay,options AS FormatOptions) AS STRING FAILS
DateTimeError
```

```
FUNCTION FormatURL(value AS URL) AS STRING
```

```
FUNCTION FormatURLQuery(parameters AS LIST OF URLParameter,form AS BOOLEAN=TRUE) AS STRING
FAILS UrlError
```

```
FUNCTION FormatUUID(value AS UUID) AS STRING
```

```
FUNCTION FormatUUID(value AS UUID,style AS UuidFormat) AS STRING
```

```
FUNCTION FormatWithPattern(value AS Date,pattern AS DateTimePattern) AS STRING FAILS
DateTimeError
```

```
FUNCTION FormatWithPattern(value AS Instant,pattern AS DateTimePattern) AS STRING FAILS
DateTimeError
```

```
FUNCTION FormatWithPattern(value AS LocalDateTime,pattern AS DateTimePattern) AS STRING FAILS
DateTimeError
```

```
FUNCTION FormatWithPattern(value AS OffsetDateTime,pattern AS DateTimePattern) AS STRING
FAILS DateTimeError
```

```
FUNCTION FormatWithPattern(value AS TimeOfDay,pattern AS DateTimePattern) AS STRING FAILS
DateTimeError
```

```
FUNCTION FormatYearMonth(value AS YearMonth) AS STRING FAILS DateTimeError
```

```
FUNCTION FormatZonedDateTime(value AS ZonedDateTime) AS STRING FAILS DateTimeError
```

```
FUNCTION FormatZonedDateTime(value AS ZonedDateTime,options AS FormatOptions) AS STRING FAILS
DateTimeError
```

```
FUNCTION FREXP(value AS DOUBLE) AS FloatParts
```

```
FUNCTION FromIsoWeekDate(value AS IsoWeekDate) AS Date FAILS DateTimeError
```

```
FUNCTION FromJSON[T](value AS JsonValue) AS T FAILS JsonError,LIKE __JsonDecodeErrors(AS T)
```

```
FUNCTION FromJSON[T](value AS JsonValue,codec AS JsonCodec) AS T FAILS JsonError,LIKE codec,
LIKE __JsonDecodeErrors(AS T)
```

```
FUNCTION FromValue[T](value AS JsonValue) AS T FAILS JsonError,LIKE __JsonDecodeErrors(AS T)
```

```
FUNCTION Front[T](q AS BoundedDeque[T]) AS BORROW T FROM q FAILS EmptyCollection WHERE T IS
RESOURCE
```

```
FUNCTION Front[T](q AS BoundedDeque[T]) AS T FAILS EmptyCollection WHERE T IS COPYABLE
```

```
FUNCTION Front[T](q AS Deque[T]) AS BORROW T FROM q FAILS EmptyCollection WHERE T IS RESOURCE
```

```
FUNCTION Front[T](q AS Deque[T]) AS T FAILS EmptyCollection WHERE T IS COPYABLE
```

```
FUNCTION Front[T](q AS Queue[T]) AS BORROW T FROM q FAILS EmptyCollection WHERE T IS RESOURCE
```

```
FUNCTION Front[T](q AS Queue[T]) AS T FAILS EmptyCollection WHERE T IS COPYABLE
```

G

```
FUNCTION GaussianBlurCanvas(canvas AS Canvas,sigma AS DOUBLE) AS Canvas FAILS
CanvasError,IndexError,NumericError
```

```
FUNCTION Get(value AS JsonValue,key AS STRING) AS JsonValue FAILS JsonError,MissingKey
```

```
FUNCTION Get(value AS JsonValue,key AS STRING,fallback AS JsonValue) AS JsonValue FAILS
JsonError
```

```
FUNCTION Get[K,V](m AS MultiMap[K,V],key AS K) AS BORROW LIST OF V FROM m FAILS
MissingKey,LIKE ValidateKey(AS K) WHERE V IS RESOURCE
```

```
FUNCTION Get[K,V](m AS MultiMap[K,V],key AS K) AS LIST OF V FAILS MissingKey,LIKE
ValidateKey(AS K) WHERE V IS COPYABLE
```

```
FUNCTION Get[K,V](table AS DICT OF K TO V, key AS K) AS BORROW V FROM table FAILS MissingKey,
LIKE ValidateKey(AS K) WHERE V IS RESOURCE
```

```
FUNCTION Get[K,V](table AS DICT OF K TO V, key AS K) AS V FAILS MissingKey, LIKE
ValidateKey(AS K) WHERE V IS COPYABLE
```

```
FUNCTION Get[K,V](table AS DICT OF K TO V, key AS K, fallback AS V) AS BORROW V FROM table OR
fallback FAILS LIKE ValidateKey(AS K) WHERE V IS RESOURCE
```

```
FUNCTION Get[K,V](table AS DICT OF K TO V, key AS K, fallback AS V) AS V FAILS LIKE
ValidateKey(AS K) WHERE V IS COPYABLE
```

```
FUNCTION GetAll[K,V](m AS MultiMap[K,V],key AS K) AS LIST OF V FAILS LIKE ValidateKey(AS K)
WHERE V IS COPYABLE
```

```
FUNCTION GetFileInfo(file AS File) AS FileInfo FAILS IoError
```

```
FUNCTION GetFileInfo(path AS STRING,followSymbolicLinks AS BOOLEAN=TRUE) AS FileInfo FAILS
IoError
```

```
FUNCTION GetFontMetrics(font AS Font,size AS DOUBLE) AS FontMetrics FAILS CanvasError
```

```
FUNCTION GetKerning(font AS Font,leftScalar AS INTEGER,rightScalar AS INTEGER,size AS
DOUBLE=24.0) AS DOUBLE FAILS CanvasError,IndexError
```

```
FUNCTION GetLogConfig() AS LogConfig FAILS LoggingError
```

```
FUNCTION GetLogConfig(logger AS Logger) AS LogConfig FAILS LoggingError
```

```
FUNCTION GetLogConfig(logger AS LoggerHandle) AS LogConfig FAILS LoggingError
```

```
FUNCTION GetPixel(canvas AS Canvas,x AS INTEGER,y AS INTEGER) AS Color FAILS
CanvasError,IndexError,NumericError
```

```
FUNCTION GetTerminalSize(terminal AS Terminal) AS TerminalSize FAILS IoError
```

```
FUNCTION GraphemeBoundaries(text AS STRING) AS LIST OF INTEGER FAILS CanvasError,IndexError
```

```
FUNCTION GrayscaleCanvas(canvas AS Canvas) AS Canvas FAILS
CanvasError,IndexError,NumericError
```

H

```
FUNCTION Has(bits AS BitSet,index AS INTEGER) AS BOOLEAN FAILS IndexError
```

```
FUNCTION Has[K,V](d AS SortedDictionary[K,V],key AS K) AS BOOLEAN FAILS LIKE
__ValueOrderCheck(AS K),LIKE ValidateKey(AS K)
```

```
FUNCTION Has[K,V](m AS MultiMap[K,V],key AS K) AS BOOLEAN FAILS LIKE ValidateKey(AS K)
```

```
FUNCTION Has[T](s AS SortedSet[T],value AS T) AS BOOLEAN FAILS LIKE __ValueOrderCheck(AS
T),LIKE ValidateKey(AS T)
```

```
FUNCTION Hash[T](value AS T) AS INTEGER
```

```
FUNCTION HasURLQueryParameter(parameters AS LIST OF URLParameter,name AS STRING) AS BOOLEAN
```

```
FUNCTION HasURLQueryParameter(value AS URL,name AS STRING) AS BOOLEAN FAILS UrlError
```

```
FUNCTION Header(name AS STRING,value AS STRING) AS HTTPHeader
```

```
FUNCTION HexDigest(bytes AS LIST OF BYTE) AS STRING FAILS StringError
```

```
FUNCTION Histogram(canvas AS Canvas) AS ColorHistogram FAILS IndexError
```

```
FUNCTION HistogramRegion(canvas AS Canvas,region AS PixelRect) AS ColorHistogram FAILS
CanvasError,IndexError
```

```
FUNCTION HitTestTextLayout(layout AS TextLayout,x AS DOUBLE,y AS DOUBLE) AS TextHit FAILS
CanvasError,IndexError
```

```
FUNCTION Hour(value AS TimeOfDay) AS INTEGER
```

```
FUNCTION Hours(value AS INTEGER) AS Duration FAILS DateTimeError
```

```
FUNCTION HTTPClientOptions() AS HTTPClientOptions
```

```
FUNCTION HTTPContentType(path AS STRING) AS STRING FAILS StringError
```

```
FUNCTION HTTPDate(seconds AS INTEGER) AS STRING FAILS StringError
```

```
FUNCTION HTTPExchange() AS HTTPExchange
```

```
FUNCTION HTTPExchange(options AS HTTPOptions) AS HTTPExchange
```

```
FUNCTION HTTPFallback(BYREF router AS HTTPRouter,handler AS
CALLABLE(HTTPRequest,TCPConnection,BYREF HTTPExchange) FAILS (Error)) AS INTEGER
```

```
FUNCTION HTTPGet(url AS STRING) AS HTTPResponse FAILS
IoError,HTTPError,TLSError,IndexError,StringError
```

```
FUNCTION HTTPGet(url AS STRING,options AS HTTPClientOptions) AS HTTPResponse FAILS
IoError,HTTPError,TLSError,IndexError,StringError
```

```
FUNCTION HTTPGet(url AS URL) AS HTTPResponse FAILS
IoError,HTTPError,TLSError,IndexError,StringError
```

```
FUNCTION HTTPGet(url AS URL,options AS HTTPClientOptions) AS HTTPResponse FAILS
IoError,HTTPError,TLSError,IndexError,StringError
```

```
FUNCTION HTTPGetJSON[T](url AS STRING) AS OPTIONAL OF T FAILS
IoError,HTTPError,TLSError,IndexError,StringError,HTTPStatusError,JsonError,LIKE
__JsonDecodeErrors(AS T)
```

```
FUNCTION HTTPGetJSON[T](url AS STRING,BYREF response AS HTTPResponse,codec AS
JsonCodec,options AS HTTPClientOptions) AS OPTIONAL OF T FAILS
IoError,HTTPError,TLSError,IndexError,StringError,HTTPStatusError,JsonError,LIKE codec, LIKE
__JsonDecodeErrors(AS T)
```

```
FUNCTION HTTPGetJSON[T](url AS STRING,codec AS JsonCodec,options AS HTTPClientOptions) AS
OPTIONAL OF T FAILS
IoError,HTTPError,TLSError,IndexError,StringError,HTTPStatusError,JsonError,LIKE codec, LIKE
__JsonDecodeErrors(AS T)
```

```
FUNCTION HTTPGetPlain(url AS STRING) AS HTTPResponse FAILS
IoError,HTTPError,IndexError,StringError
```

```
FUNCTION HTTPGetPlain(url AS STRING,options AS HTTPClientOptions) AS HTTPResponse FAILS
IoError,HTTPError,IndexError,StringError
```

```
FUNCTION HTTPGetPlain(url AS URL) AS HTTPResponse FAILS
IoError,HTTPError,IndexError,StringError
```

```
FUNCTION HTTPGetPlain(url AS URL,options AS HTTPClientOptions) AS HTTPResponse FAILS
IoError,HTTPError,IndexError,StringError
```

```
FUNCTION HTTPHeaderValue(headers AS LIST OF HTTPHeader,name AS STRING) AS OPTIONAL OF STRING
```

```
FUNCTION HTTPOptions() AS HTTPOptions
```

```
FUNCTION HTTPPost(url AS STRING,body AS LIST OF BYTE,contentType AS STRING) AS HTTPResponse
FAILS IoError,HTTPError,TLSError,IndexError,StringError
```

```
FUNCTION HTTPPost(url AS STRING,body AS LIST OF BYTE,contentType AS STRING,options AS
HTTPClientOptions) AS HTTPResponse FAILS IoError,HTTPError,TLSError,IndexError,StringError
```

```
FUNCTION HTTPPost(url AS STRING,text AS STRING,contentType AS STRING) AS HTTPResponse FAILS
IoError,HTTPError,TLSError,IndexError,StringError
```

```
FUNCTION HTTPPost(url AS STRING,text AS STRING,contentType AS STRING,options AS
HTTPClientOptions) AS HTTPResponse FAILS IoError,HTTPError,TLSError,IndexError,StringError
```

```
FUNCTION HTTPPost(url AS URL,body AS LIST OF BYTE,contentType AS STRING) AS HTTPResponse
FAILS IoError,HTTPError,TLSError,IndexError,StringError
```

```
FUNCTION HTTPPost(url AS URL,body AS LIST OF BYTE,contentType AS STRING,options AS
HTTPClientOptions) AS HTTPResponse FAILS IoError,HTTPError,TLSError,IndexError,StringError
```

```
FUNCTION HTTPPost(url AS URL,text AS STRING,contentType AS STRING) AS HTTPResponse FAILS
IoError,HTTPError,TLSError,IndexError,StringError
```

```
FUNCTION HTTPPost(url AS URL,text AS STRING,contentType AS STRING,options AS
HTTPClientOptions) AS HTTPResponse FAILS IoError,HTTPError,TLSError,IndexError,StringError
```

```
FUNCTION HTTPPostJSON[B,T](url AS STRING,body AS B) AS OPTIONAL OF T FAILS
IoError,HTTPError,TLSError,IndexError,StringError,HTTPStatusError,JsonError,LIKE
__JsonEncodeErrors(AS B), LIKE __JsonDecodeErrors(AS T)
```

```
FUNCTION HTTPPostJSON[B,T](url AS STRING,body AS B,BYREF response AS HTTPResponse,codec AS
JsonCodec,options AS HTTPClientOptions) AS OPTIONAL OF T FAILS
IoError,HTTPError,TLSError,IndexError,StringError,HTTPStatusError,JsonError,LIKE codec, LIKE
__JsonEncodeErrors(AS B), LIKE __JsonDecodeErrors(AS T)
```

```
FUNCTION HTTPQueryValues(request AS HTTPRequest,name AS STRING) AS LIST OF STRING FAILS
HTTPError,StringError
```

```
FUNCTION HTTPResponse() AS HTTPResponse
```

```
FUNCTION HTTPRouter() AS HTTPRouter
```

```
FUNCTION HYPOT(x AS DOUBLE, y AS DOUBLE) AS DOUBLE
```

I

```
FUNCTION IdentityTransform() AS AffineTransform
```

```
SUB Include(BYREF bits AS BitSet,index AS INTEGER) FAILS IndexError
```

```
FUNCTION IndexOf[T](items AS LIST OF T, value AS T) AS INTEGER
```

```
FUNCTION IndexWhere[T](items AS LIST OF T,Predicate AS CALLABLE(T) AS BOOLEAN) AS INTEGER
```

```
FUNCTION Infinity() AS DOUBLE
```

```
FUNCTION InputLine() AS STRING FAILS IoError,StringError
```

```
FUNCTION Insert[K](BYREF table AS SET OF K, key AS K) AS BOOLEAN FAILS LIKE ValidateKey(AS K)
```

```
FUNCTION Insert[T](BYREF items AS LIST OF T, index AS INTEGER, BYVAL value AS T) AS INTEGER
FAILS IndexError
```

```
FUNCTION InsertValue[T](BYREF s AS SortedSet[T],value AS T) AS BOOLEAN FAILS IndexError,LIKE
__ValueOrderCheck(AS T),LIKE ValidateKey(AS T)
```

```
FUNCTION InstantFromStorageJSON(value AS JsonValue) AS Instant FAILS DateTimeError
```

```
FUNCTION InstantFromUnixMicroseconds(value AS INTEGER) AS Instant FAILS DateTimeError
```

```
FUNCTION InstantFromUnixMilliseconds(value AS INTEGER) AS Instant FAILS DateTimeError
```

```
FUNCTION InstantFromUnixNanoseconds(value AS INTEGER) AS Instant FAILS DateTimeError
```

```
FUNCTION InstantFromUnixSeconds(seconds AS INTEGER,nanosecond AS INTEGER=0) AS Instant FAILS
DateTimeError
```

```
FUNCTION InstantToStorageJSON(value AS Instant) AS JsonValue FAILS DateTimeError
```

```
FUNCTION INSTR(text AS STRING, needle AS STRING) AS INTEGER
```

```
FUNCTION INSTRFROM(text AS STRING, needle AS STRING, start AS INTEGER) AS INTEGER FAILS
StringError
```

```
FUNCTION INT(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION IntegerArgument(values AS DICT OF STRING TO ArgumentValue,name AS STRING) AS INTEGER
FAILS ArgumentError
```

```
FUNCTION IntegerOption(name AS STRING,Default AS INTEGER=0,Min AS
INTEGER=-9223372036854775807-1,Max AS INTEGER=9223372036854775807,Short AS STRING="",Help AS
STRING="",Required AS BOOLEAN=FALSE,Global AS BOOLEAN=FALSE) AS CommandOption
```

```
FUNCTION Intersection(left AS DateInterval,right AS DateInterval) AS OPTIONAL OF DateInterval
```

```
FUNCTION Intersection(left AS InstantInterval,right AS InstantInterval) AS OPTIONAL OF
InstantInterval
```

```
FUNCTION Intersection[K](left AS SET OF K, right AS SET OF K) AS SET OF K
```

```
FUNCTION IntervalEnd(value AS DateInterval) AS Date
```

```
FUNCTION IntervalEnd(value AS InstantInterval) AS Instant
```

```
FUNCTION IntervalStart(value AS DateInterval) AS Date
```

```
FUNCTION IntervalStart(value AS InstantInterval) AS Instant
```

```
FUNCTION InvertedCanvas(canvas AS Canvas) AS Canvas FAILS IndexError,NumericError
```

```
FUNCTION InvertTransform(transform AS AffineTransform) AS AffineTransform FAILS CanvasError
```

```
FUNCTION IsAbsoluteURL(value AS URL) AS BOOLEAN
```

```
FUNCTION IsEmpty(value AS DateInterval) AS BOOLEAN
```

```
FUNCTION IsEmpty(value AS InstantInterval) AS BOOLEAN
```

```
FUNCTION IsExpired(clock AS Clock,deadline AS Deadline) AS BOOLEAN FAILS DateTimeError
```

```
FUNCTION IsFinite(value AS DOUBLE) AS BOOLEAN
```

```
FUNCTION IsInfinite(value AS DOUBLE) AS BOOLEAN
```

```
FUNCTION IsJsonNull(value AS JsonValue) AS BOOLEAN
```

```
FUNCTION IsJsonObject(value AS JsonValue) AS BOOLEAN
```

```
FUNCTION IsLeapYear(year AS INTEGER) AS BOOLEAN
```

```
FUNCTION IsMaxUUID(value AS UUID) AS BOOLEAN
```

```
FUNCTION IsNaN(value AS DOUBLE) AS BOOLEAN
```

```
FUNCTION IsNilUUID(value AS UUID) AS BOOLEAN
```

```
FUNCTION IsSqlNull(value AS SqlValue) AS BOOLEAN
```

```
FUNCTION IsTLS(connection AS TCPConnection) AS BOOLEAN
```

```
FUNCTION IsValidURL(text AS STRING) AS BOOLEAN
```

J

```
FUNCTION JOIN(parts AS LIST OF STRING, separator AS STRING) AS STRING
```

```
FUNCTION JoinURLPath(BYVAL base AS URL,path AS STRING) AS URL FAILS UrlError
```

```
FUNCTION JoinURLPath(BYVAL base AS URL,path AS STRING,Query AS DICT OF STRING TO LIST OF
STRING) AS URL FAILS UrlError
```

```
FUNCTION JoinURLPath(BYVAL base AS URL,path AS STRING,Query AS DICT OF STRING TO STRING) AS
URL FAILS UrlError
```

```
FUNCTION JoinURLPath(BYVAL base AS URL,path AS STRING,Query AS LIST OF URLParameter) AS URL
FAILS UrlError
```

```
FUNCTION JoinURLPath(base AS STRING,path AS STRING) AS STRING FAILS UrlError
```

```
FUNCTION JoinURLPath(base AS STRING,path AS STRING,Query AS DICT OF STRING TO LIST OF STRING)
AS STRING FAILS UrlError
```

```
FUNCTION JoinURLPath(base AS STRING,path AS STRING,Query AS DICT OF STRING TO STRING) AS
STRING FAILS UrlError
```

```
FUNCTION JoinURLPath(base AS STRING,path AS STRING,Query AS LIST OF URLParameter) AS STRING
FAILS UrlError
```

```
FUNCTION JsonArray() AS JsonValue
```

```
FUNCTION JsonArray(values AS LIST OF JsonValue) AS JsonValue
```

```
FUNCTION JsonAt(value AS JsonValue,index AS INTEGER) AS JsonValue FAILS JsonError
```

```
FUNCTION JsonBoolean(value AS BOOLEAN) AS JsonValue
```

```
FUNCTION JsonCodec(Pretty AS BOOLEAN=FALSE,Indent AS INTEGER=2,MaxDepth AS
INTEGER=32,MaxInputBytes AS INTEGER=8388608,MaxOutputBytes AS INTEGER=8388608,IgnoreUnknown
AS BOOLEAN=FALSE) AS JsonCodec
```

```
FUNCTION JsonField(name AS STRING,BYVAL value AS JsonValue) AS JsonMember
```

```
FUNCTION JsonField[T](name AS STRING,value AS T) AS JsonMember FAILS LIKE ToJSON(AS T)
```

```
FUNCTION JsonGetArray(value AS JsonValue) AS LIST OF JsonValue FAILS JsonError
```

```
FUNCTION JsonGetBoolean(value AS JsonValue) AS BOOLEAN FAILS JsonError
```

```
FUNCTION JsonGetDouble(value AS JsonValue) AS DOUBLE FAILS JsonError
```

```
FUNCTION JsonGetInteger(value AS JsonValue) AS INTEGER FAILS JsonError
```

```
FUNCTION JsonGetObject(value AS JsonValue) AS DICT OF STRING TO JsonValue FAILS JsonError
```

```
FUNCTION JsonGetString(value AS JsonValue) AS STRING FAILS JsonError
```

```
FUNCTION JsonGetUInt64(value AS JsonValue) AS UINT64 FAILS JsonError
```

```
FUNCTION JsonLinesReader() AS JsonLinesReader
```

```
FUNCTION JsonNull() AS JsonValue
```

```
FUNCTION JsonNumber(text AS STRING) AS JsonValue FAILS JsonError
```

```
FUNCTION JsonNumber(value AS DOUBLE) AS JsonValue FAILS JsonError
```

```
FUNCTION JsonNumber(value AS INTEGER) AS JsonValue
```

```
FUNCTION JsonNumber(value AS UINT64) AS JsonValue
```

```
FUNCTION JsonNumberText(value AS JsonValue) AS STRING FAILS JsonError
```

```
FUNCTION JsonObject() AS JsonValue
```

```
FUNCTION JsonObject(members AS LIST OF JsonMember) AS JsonValue
```

```
FUNCTION JsonObject(values AS DICT OF STRING TO JsonValue) AS JsonValue
```

```
FUNCTION JsonPointer(path AS STRING,key AS STRING) AS STRING
```

```
FUNCTION JsonPut(BYREF object AS JsonValue,key AS STRING,BYVAL value AS JsonValue) AS BOOLEAN
FAILS JsonError
```

```
FUNCTION JsonPut[T](BYREF object AS JsonValue,key AS STRING,value AS T) AS BOOLEAN FAILS
JsonError,LIKE ToJSON(AS T)
```

```
FUNCTION JSONResponse[T](value AS T,status AS INTEGER,codec AS JsonCodec) AS HTTPResponse
FAILS HTTPError,JsonError,LIKE codec, LIKE __JsonEncodeErrors(AS T)
```

```
FUNCTION JSONResponse[T](value AS T,status AS INTEGER,headers AS LIST OF HTTPHeader,codec AS
JsonCodec) AS HTTPResponse FAILS HTTPError,JsonError,LIKE codec, LIKE __JsonEncodeErrors(AS
T)
```

```
FUNCTION JSONResponse[T](value AS T,status AS INTEGER=200) AS HTTPResponse FAILS
HTTPError,JsonError,LIKE __JsonEncodeErrors(AS T)
```

```
FUNCTION JsonString(value AS STRING) AS JsonValue
```

```
FUNCTION JsonTypeSchema[T](witness AS OPTIONAL OF T) AS TypeGraph
```

K

```
FUNCTION Keys(value AS JsonValue) AS LIST OF STRING FAILS JsonError
```

```
FUNCTION Keys[K,V](table AS DICT OF K TO V) AS LIST OF K
```

```
FUNCTION Keys[K](table AS SET OF K) AS LIST OF K
```

```
FUNCTION KeySnapshot[K,V](d AS SortedDictionary[K,V]) AS LIST OF K
```

```
FUNCTION KeySnapshot[K,V](m AS MultiMap[K,V]) AS LIST OF K
```

```
FUNCTION KeySnapshot[T](c AS Counter[T]) AS LIST OF T
```

L

```
FUNCTION LastIndexOf[T](items AS LIST OF T, value AS T) AS INTEGER
```

```
FUNCTION LastIndexWhere[T](items AS LIST OF T,Predicate AS CALLABLE(T) AS BOOLEAN) AS INTEGER
```

```
FUNCTION LBOUND[T](items AS ARRAY OF T,dimension AS INTEGER=1) AS INTEGER
```

```
FUNCTION LCASE(text AS STRING) AS STRING
```

```
FUNCTION LDEXP(value AS DOUBLE, exponent AS INTEGER) AS DOUBLE
```

```
FUNCTION LeadingZeros(value AS UINT64) AS INTEGER
```

```
FUNCTION LEFT(text AS STRING, count AS INTEGER) AS STRING FAILS StringError
```

```
FUNCTION Length(bits AS BitSet) AS INTEGER
```

```
FUNCTION Length(buffer AS ByteBuffer) AS INTEGER
```

```
FUNCTION Length(builder AS StringBuilder) AS INTEGER
```

```
FUNCTION Length(reader AS ByteReader) AS INTEGER
```

```
FUNCTION Length(writer AS ByteWriter) AS INTEGER
```

```
FUNCTION Length[K,V](d AS SortedDictionary[K,V]) AS INTEGER
```

```
FUNCTION Length[K,V](m AS MultiMap[K,V]) AS INTEGER
```

```
FUNCTION Length[T](c AS Counter[T]) AS INTEGER
```

```
FUNCTION Length[T](q AS BoundedDeque[T]) AS INTEGER
```

```
FUNCTION Length[T](q AS Deque[T]) AS INTEGER
```

```
FUNCTION Length[T](q AS PriorityQueue[T]) AS INTEGER
```

```
FUNCTION Length[T](q AS Queue[T]) AS INTEGER
```

```
FUNCTION Length[T](s AS SortedSet[T]) AS INTEGER
```

```
FUNCTION LINES(text AS STRING) AS LIST OF STRING
```

```
SUB LineTo(BYREF path AS CanvasPath,x AS DOUBLE,y AS DOUBLE) FAILS CanvasError
```

```
FUNCTION ListenTCP(address AS IPAddress,port AS INTEGER,backlog AS INTEGER=128) AS
TCPListener FAILS IoError
```

```
FUNCTION ListenTCP(host AS STRING,port AS INTEGER,backlog AS INTEGER=128) AS TCPListener
FAILS IoError
```

```
FUNCTION LN(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION LoadAnimation(path AS STRING,maxPixels AS INTEGER=16777216,maxFrames AS
INTEGER=1000,maxDecodedBytes AS INTEGER=268435456,maxEncodedBytes AS INTEGER=67108864) AS
Animation FAILS CanvasError,IndexError,NumericError,StringError,IoError
```

```
FUNCTION LoadFont(path AS STRING,maxBytes AS INTEGER=33554432,faceIndex AS INTEGER=0) AS Font
FAILS CanvasError,IndexError,NumericError,IoError
```

```
FUNCTION LoadFontBytes(BYVAL bytes AS LIST OF BYTE,maxBytes AS INTEGER=33554432,faceIndex AS
INTEGER=0) AS Font FAILS CanvasError,IndexError,NumericError
```

```
FUNCTION LoadImage(path AS STRING,maxPixels AS INTEGER=16777216,maxEncodedBytes AS
INTEGER=67108864) AS Canvas FAILS CanvasError,IndexError,NumericError,StringError,IoError
```

```
FUNCTION LoadTimeZoneBytes(id AS STRING,data AS LIST OF BYTE) AS TimeZone FAILS DateTimeError
```

```
FUNCTION LoadTimeZoneFile(id AS STRING,path AS STRING) AS TimeZone FAILS DateTimeError
```

```
FUNCTION LocalPart(value AS OffsetDateTime) AS LocalDateTime
```

```
FUNCTION LocalPart(value AS ZonedDateTime) AS LocalDateTime FAILS DateTimeError
```

```
FUNCTION LocalPort(socket AS TCPConnection) AS INTEGER FAILS IoError
```

```
FUNCTION LocalPort(socket AS TCPListener) AS INTEGER FAILS IoError
```

```
FUNCTION LOG(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION LOG(value AS DOUBLE, base AS DOUBLE) AS DOUBLE
```

```
FUNCTION LOG10(input AS DOUBLE) AS DOUBLE
```

```
FUNCTION LOG1P(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION LOG2(input AS DOUBLE) AS DOUBLE
```

```
FUNCTION LogArgument(value AS BOOLEAN) AS LogFieldValue
```

```
FUNCTION LogArgument(value AS BYTE) AS LogFieldValue
FUNCTION LogArgument(value AS DOUBLE) AS LogFieldValue
FUNCTION LogArgument(value AS Error) AS LogFieldValue
FUNCTION LogArgument(value AS INTEGER) AS LogFieldValue
FUNCTION LogArgument(value AS LogFieldValue) AS LogFieldValue
FUNCTION LogArgument(value AS STRING) AS LogFieldValue
FUNCTION LogArgument(value AS UINT32) AS LogFieldValue
FUNCTION LogArgument(value AS UINT64) AS LogFieldValue
FUNCTION LogCallable[T](handler AS T,label AS STRING="<function>") AS STRING
SUB LogDebug(logger AS Logger,message AS STRING)
SUB LogDebug(logger AS LoggerHandle,message AS STRING)
SUB LogDebug(message AS STRING)
SUB LogDebug[A0,A1](message AS STRING,argument0 AS A0,argument1 AS A1)
SUB LogDebug[A0](logger AS Logger,message AS STRING,argument0 AS A0)
SUB LogDebug[A0](logger AS LoggerHandle,message AS STRING,argument0 AS A0)
SUB LogDebug[A0](message AS STRING,argument0 AS A0)
FUNCTION LogEnabled(level AS LogLevel) AS BOOLEAN
FUNCTION LogEnabled(logger AS Logger,level AS LogLevel) AS BOOLEAN
FUNCTION LogEnabled(logger AS LoggerHandle,level AS LogLevel) AS BOOLEAN
SUB LogError(logger AS Logger,message AS STRING)
SUB LogError(logger AS LoggerHandle,message AS STRING)
SUB LogError(message AS STRING)
SUB LogError[A0,A1](message AS STRING,argument0 AS A0,argument1 AS A1)
```

```
SUB LogError[A0](logger AS Logger,message AS STRING,argument0 AS A0)
SUB LogError[A0](logger AS LoggerHandle,message AS STRING,argument0 AS A0)
SUB LogError[A0](message AS STRING,argument0 AS A0)
SUB LogFatal(logger AS Logger,message AS STRING)
SUB LogFatal(logger AS LoggerHandle,message AS STRING)
SUB LogFatal(message AS STRING)
SUB LogFatal[A0,A1](message AS STRING,argument0 AS A0,argument1 AS A1)
SUB LogFatal[A0](logger AS Logger,message AS STRING,argument0 AS A0)
SUB LogFatal[A0](logger AS LoggerHandle,message AS STRING,argument0 AS A0)
SUB LogFatal[A0](message AS STRING,argument0 AS A0)
FUNCTION LogFieldIndex(name AS STRING,count AS INTEGER) AS INTEGER FAILS LogFormatError
FUNCTION LogHandle(logger AS Logger) AS LoggerHandle
FUNCTION LogHandle(logger AS LoggerHandle) AS LoggerHandle
SUB LogInfo(logger AS Logger,message AS STRING)
SUB LogInfo(logger AS LoggerHandle,message AS STRING)
SUB LogInfo(message AS STRING)
SUB LogInfo[A0,A1](message AS STRING,argument0 AS A0,argument1 AS A1)
SUB LogInfo[A0](logger AS Logger,message AS STRING,argument0 AS A0)
SUB LogInfo[A0](logger AS LoggerHandle,message AS STRING,argument0 AS A0)
SUB LogInfo[A0](message AS STRING,argument0 AS A0)
FUNCTION LogLevelName(level AS LogLevel) AS STRING
SUB LogTrace(logger AS Logger,message AS STRING)
SUB LogTrace(logger AS LoggerHandle,message AS STRING)
```

```
SUB LogTrace(message AS STRING)
```

```
SUB LogTrace[A0,A1](message AS STRING,argument0 AS A0,argument1 AS A1)
```

```
SUB LogTrace[A0](logger AS Logger,message AS STRING,argument0 AS A0)
```

```
SUB LogTrace[A0](logger AS LoggerHandle,message AS STRING,argument0 AS A0)
```

```
SUB LogTrace[A0](message AS STRING,argument0 AS A0)
```

```
SUB LogWarn(logger AS Logger,message AS STRING)
```

```
SUB LogWarn(logger AS LoggerHandle,message AS STRING)
```

```
SUB LogWarn(message AS STRING)
```

```
SUB LogWarn[A0,A1](message AS STRING,argument0 AS A0,argument1 AS A1)
```

```
SUB LogWarn[A0](logger AS Logger,message AS STRING,argument0 AS A0)
```

```
SUB LogWarn[A0](logger AS LoggerHandle,message AS STRING,argument0 AS A0)
```

```
SUB LogWarn[A0](message AS STRING,argument0 AS A0)
```

```
FUNCTION Lookup[K,V](d AS SortedDictionary[K,V],key AS K) AS BORROW OPTIONAL OF V FROM d
FAILS LIKE __ValueOrderCheck(AS K),LIKE ValidateKey(AS K) WHERE V IS RESOURCE
```

```
FUNCTION Lookup[K,V](d AS SortedDictionary[K,V],key AS K) AS OPTIONAL OF V FAILS LIKE
__ValueOrderCheck(AS K),LIKE ValidateKey(AS K) WHERE V IS COPYABLE
```

```
FUNCTION LookupRequired[K,V](d AS SortedDictionary[K,V],key AS K) AS BORROW V FROM d FAILS
MissingKey,LIKE __ValueOrderCheck(AS K),LIKE ValidateKey(AS K) WHERE V IS RESOURCE
```

```
FUNCTION LookupRequired[K,V](d AS SortedDictionary[K,V],key AS K) AS V FAILS MissingKey,LIKE
__ValueOrderCheck(AS K),LIKE ValidateKey(AS K) WHERE V IS COPYABLE
```

```
FUNCTION LOWER(text AS STRING) AS STRING
```

```
FUNCTION LowerBoundIndex[T](s AS SortedSet[T],value AS T) AS INTEGER FAILS LIKE
__ValueOrderCheck(AS T),LIKE ValidateKey(AS T)
```

```
FUNCTION LTRIM(text AS STRING) AS STRING
```

M

```
FUNCTION MakeDirectory(path AS STRING) AS INTEGER FAILS IoError
```

```
FUNCTION MapLocal(zone AS TimeZone,local AS LocalDateTime) AS LocalMapping FAILS
DateTimeError
```

```
FUNCTION MaskedCanvas(canvas AS Canvas,mask AS Canvas,selectedChannel AS
MaskChannel=MaskChannel.Alpha,inverted AS BOOLEAN=FALSE,space AS
BlendSpace=BlendSpace.EncodedSRGB) AS Canvas FAILS CanvasError,IndexError,NumericError
```

```
FUNCTION MAX(left AS DOUBLE, right AS DOUBLE) AS DOUBLE
```

```
FUNCTION MAX(left AS INTEGER, right AS INTEGER) AS INTEGER
```

```
FUNCTION MaxUUID() AS UUID
```

```
FUNCTION MeasureBitmapText(text AS STRING,scale AS INTEGER=1) AS PixelRect FAILS CanvasError
```

```
FUNCTION MeasureText(font AS Font,text AS STRING,size AS DOUBLE=24.0,strictGlyphs AS
BOOLEAN=FALSE,kerning AS BOOLEAN=TRUE) AS TextMetrics FAILS CanvasError,IndexError
```

```
FUNCTION Members(bits AS BitSet) AS LIST OF INTEGER
```

```
FUNCTION MergeURLQuery(BYVAL value AS URL,parameters AS DICT OF STRING TO LIST OF STRING) AS
URL FAILS UrlError
```

```
FUNCTION MergeURLQuery(BYVAL value AS URL,parameters AS DICT OF STRING TO STRING) AS URL
FAILS UrlError
```

```
FUNCTION MergeURLQuery(BYVAL value AS URL,parameters AS LIST OF URLParameter) AS URL FAILS
UrlError
```

```
FUNCTION Microseconds(value AS INTEGER) AS Duration
```

```
FUNCTION MID(text AS STRING, start AS INTEGER, count AS INTEGER) AS STRING FAILS StringError
```

```
FUNCTION Milliseconds(value AS INTEGER) AS Duration
```

```
FUNCTION MIN(left AS DOUBLE, right AS DOUBLE) AS DOUBLE
```

```
FUNCTION MIN(left AS INTEGER, right AS INTEGER) AS INTEGER
```

```
FUNCTION Minute(value AS TimeOfDay) AS INTEGER
```

```
FUNCTION Minutes(value AS INTEGER) AS Duration FAILS DateTimeError
```

```
FUNCTION MonotonicNanoseconds() AS INTEGER FAILS IoError
```

```
FUNCTION MonthNumber(value AS Month) AS INTEGER
```

```
FUNCTION MonthOf(value AS Date) AS Month
```

```
FUNCTION MountDirectory(BYREF router AS HTTPRouter,prefix AS STRING,root AS STRING) AS
INTEGER FAILS HTTPError,StringError
```

```
SUB MoveTo(BYREF path AS CanvasPath,x AS DOUBLE,y AS DOUBLE) FAILS CanvasError
```

```
FUNCTION MultiplyDuration(value AS Duration,factor AS INTEGER) AS Duration FAILS
DateTimeError
```

```
FUNCTION MultiplyHigh(left AS UINT64, right AS UINT64) AS UINT64
```

```
FUNCTION MultiplyWide(left AS UINT64, right AS UINT64) AS WideProduct
```

N

```
FUNCTION NaN() AS DOUBLE
```

```
FUNCTION NanosecondFraction(value AS Duration) AS INTEGER
```

```
FUNCTION NanosecondFraction(value AS Instant) AS INTEGER
```

```
FUNCTION NanosecondFraction(value AS TimeOfDay) AS INTEGER
```

```
FUNCTION Nanoseconds(value AS INTEGER) AS Duration
```

```
FUNCTION NegateDuration(value AS Duration) AS Duration FAILS DateTimeError
```

```
FUNCTION NegatePeriod(value AS Period) AS Period FAILS DateTimeError
```

```
FUNCTION NewAnimation(width AS INTEGER,height AS INTEGER,plays AS INTEGER=1,maxDecodedBytes
AS INTEGER=268435456) AS Animation FAILS CanvasError
```

```
FUNCTION NewBitSet(size AS INTEGER) AS BitSet FAILS CollectionError,IndexError
```

```
FUNCTION NewBoundedDeque[T](capacity AS INTEGER) AS BoundedDeque[T] FAILS CollectionError
```

```
FUNCTION NewCache[K,V](maxEntries AS INTEGER,policy AS
CachePolicy=CachePolicy.LRU,expireAfter AS OPTIONAL OF Duration=NONE,maxWeight AS INTEGER=0)
AS Cache[K,V] FAILS CollectionError
```

```
FUNCTION NewCanvas(width AS INTEGER,height AS INTEGER,maxPixels AS INTEGER=16777216) AS
Canvas FAILS CanvasError,NumericError,IndexError
```

```
FUNCTION NewCounter[T]() AS Counter[T]
```

```
FUNCTION NewDate(year AS INTEGER,month AS INTEGER,day AS INTEGER) AS Date FAILS DateTimeError
```

```
FUNCTION NewDate(year AS INTEGER,month AS Month,day AS INTEGER) AS Date FAILS DateTimeError
```

```
FUNCTION NewDateInterval(start AS Date,finish AS Date) AS DateInterval FAILS DateTimeError
```

```
FUNCTION NewDateTimeAmount(calendar AS Period,exact AS Duration) AS DateTimeAmount
```

```
FUNCTION NewDeque[T]() AS Deque[T]
```

```
FUNCTION NewDuration(seconds AS INTEGER,nanoseconds AS INTEGER=0) AS Duration FAILS
DateTimeError
```

```
FUNCTION NewFontSet() AS FontSet
```

```
FUNCTION NewInstantInterval(start AS Instant,finish AS Instant) AS InstantInterval FAILS
DateTimeError
```

```
FUNCTION NewIsoWeekDate(year AS INTEGER,week AS INTEGER,weekday AS INTEGER) AS IsoWeekDate
FAILS DateTimeError
```

```
FUNCTION NewIsoWeekDate(year AS INTEGER,week AS INTEGER,weekday AS Weekday) AS IsoWeekDate
FAILS DateTimeError
```

```
FUNCTION NewLocalDateTime(value AS Date,time AS TimeOfDay) AS LocalDateTime
```

```
FUNCTION NewLocalDateTime(year AS INTEGER,month AS INTEGER,day AS INTEGER,hour AS
INTEGER=0,minute AS INTEGER=0,second AS INTEGER=0,nanosecond AS INTEGER=0) AS LocalDateTime
FAILS DateTimeError
```

```
FUNCTION NewLocalDateTime(year AS INTEGER,month AS Month,day AS INTEGER,hour AS
INTEGER=0,minute AS INTEGER=0,second AS INTEGER=0,nanosecond AS INTEGER=0) AS LocalDateTime
FAILS DateTimeError
```

```
FUNCTION NewMonthDay(month AS INTEGER,day AS INTEGER) AS MonthDay FAILS DateTimeError
```

```
FUNCTION NewMonthDay(month AS Month,day AS INTEGER) AS MonthDay FAILS DateTimeError
```

```
FUNCTION NewMultiMap[K,V]() AS MultiMap[K,V]
```

```
FUNCTION NewOffsetDateTime(value AS LocalDateTime,offset AS UtcOffset) AS OffsetDateTime
FAILS DateTimeError
```

```
FUNCTION NewOffsetTime(value AS TimeOfDay,offset AS UtcOffset) AS OffsetTime
```

```
FUNCTION NewOrderedDictionary[K,V]() AS DICT OF K TO V
```

```
FUNCTION NewOrderedSet[T]() AS SET OF T
```

```
FUNCTION NewPath() AS CanvasPath
```

```
FUNCTION NewPeriod(years AS INTEGER=0,months AS INTEGER=0,weeks AS INTEGER=0,days AS
INTEGER=0) AS Period FAILS DateTimeError
```

```
FUNCTION NewPriorityQueue[T]() AS PriorityQueue[T]
```

```
FUNCTION NewQueue[T]() AS Queue[T]
```

```
FUNCTION NewRandom(seed AS INTEGER,stream AS INTEGER=54) AS RandomGenerator
```

```
FUNCTION NewRandom(seed AS UINT64,stream AS UINT64=&H36) AS RandomGenerator
```

```
FUNCTION NewSortedDictionary[K,V]() AS SortedDictionary[K,V]
```

```
FUNCTION NewSortedSet[T]() AS SortedSet[T]
```

```
FUNCTION NewSystemClock(basis AS MonotonicBasis=MonotonicBasis.ActiveTime) AS Clock FAILS
DateTimeError
```

```
FUNCTION NewTestClock(wall AS Instant) AS Clock FAILS DateTimeError
```

```
FUNCTION NewTextDecoder(encoding AS TextEncoding,errors AS
TextErrorPolicy=TextErrorPolicy.Strict,bom AS
TextDecodeBom=TextDecodeBom.Preserve,maxOutputBytes AS INTEGER=67108864) AS TextDecoder FAILS
TextEncodingError
```

```
FUNCTION NewTextEncoder(encoding AS TextEncoding,errors AS
TextErrorPolicy=TextErrorPolicy.Strict,bom AS TextEncodeBom=TextEncodeBom.Omit,maxOutputBytes
AS INTEGER=67108864) AS TextEncoder FAILS TextEncodingError
```

```
FUNCTION NewTime(hour AS INTEGER,minute AS INTEGER=0,second AS INTEGER=0,nanosecond AS
INTEGER=0) AS TimeOfDay FAILS DateTimeError
```

```
FUNCTION NewTimedCache[K,V](maxEntries AS INTEGER,policy AS CachePolicy,expireAfter AS
Duration,now AS MonotonicInstant,maxWeight AS INTEGER=0) AS Cache[K,V] FAILS CollectionError
```

```
FUNCTION NewTransitionZone(id AS STRING,initial AS UtcOffset,changes AS LIST OF
ZoneTransition,futureKnown AS BOOLEAN=TRUE) AS TimeZone FAILS DateTimeError
```

```
FUNCTION NewUtcOffset(seconds AS INTEGER) AS UtcOffset FAILS DateTimeError
```

```
FUNCTION NewUUIDClockGenerator() AS UUIDClockGenerator FAILS IoError
```

```
FUNCTION NewYearMonth(year AS INTEGER,month AS INTEGER) AS YearMonth FAILS DateTimeError
```

```
FUNCTION NewYearMonth(year AS INTEGER,month AS Month) AS YearMonth FAILS DateTimeError
```

```
FUNCTION NEXTAFTER(value AS DOUBLE, toward AS DOUBLE) AS DOUBLE
```

```
FUNCTION NextDouble(BYREF generator AS RandomGenerator) AS DOUBLE
```

```
FUNCTION NextInteger(BYREF generator AS RandomGenerator,upper AS INTEGER) AS INTEGER FAILS
NumericError
```

```
FUNCTION NextTransition(value AS TimeZone,instant AS Instant) AS OPTIONAL OF ZoneTransition
FAILS DateTimeError
```

```
FUNCTION NextUInt32(BYREF generator AS RandomGenerator) AS UINT32
```

```
FUNCTION NextUInt64(BYREF generator AS RandomGenerator) AS UINT64
```

```
FUNCTION NextUUIDv1(BYREF generator AS UUIDClockGenerator) AS UUID FAILS UuidError,IoError
```

```
FUNCTION NextUUIDv1AtTicks(BYREF generator AS UUIDClockGenerator,ticks AS UINT64) AS UUID
FAILS UuidError
```

```
FUNCTION NextUUIDv6(BYREF generator AS UUIDClockGenerator) AS UUID FAILS UuidError,IoError
```

```
FUNCTION NextUUIDv6AtTicks(BYREF generator AS UUIDClockGenerator,ticks AS UINT64) AS UUID
FAILS UuidError
```

```
FUNCTION NilUUID() AS UUID
```

```
FUNCTION NominalBetween(start AS LocalDateTime,finish AS LocalDateTime) AS Duration FAILS
DateTimeError
```

```
FUNCTION Norm(values AS ARRAY OF BYTE) AS DOUBLE FAILS VectorRankError
```

```
FUNCTION Norm(values AS ARRAY OF DOUBLE) AS DOUBLE FAILS VectorRankError
```

```
FUNCTION Norm(values AS ARRAY OF INTEGER) AS DOUBLE FAILS VectorRankError
```

```
FUNCTION Norm(values AS ARRAY OF UINT32) AS DOUBLE FAILS VectorRankError
```

```
FUNCTION Norm(values AS ARRAY OF UINT64) AS DOUBLE FAILS VectorRankError
```

```
FUNCTION Norm(values AS LIST OF BYTE) AS DOUBLE
```

```
FUNCTION Norm(values AS LIST OF DOUBLE) AS DOUBLE
```

```
FUNCTION Norm(values AS LIST OF INTEGER) AS DOUBLE
```

```
FUNCTION Norm(values AS LIST OF UINT32) AS DOUBLE
```

```
FUNCTION Norm(values AS LIST OF UINT64) AS DOUBLE
```

```
FUNCTION Normalized(values AS ARRAY OF BYTE) AS LIST OF DOUBLE FAILS
ZeroVectorError,VectorRankError
```

```
FUNCTION Normalized(values AS ARRAY OF DOUBLE) AS LIST OF DOUBLE FAILS
ZeroVectorError,VectorRankError
```

```
FUNCTION Normalized(values AS ARRAY OF INTEGER) AS LIST OF DOUBLE FAILS
ZeroVectorError,VectorRankError
```

```
FUNCTION Normalized(values AS ARRAY OF UINT32) AS LIST OF DOUBLE FAILS
ZeroVectorError,VectorRankError
```

```
FUNCTION Normalized(values AS ARRAY OF UINT64) AS LIST OF DOUBLE FAILS
ZeroVectorError,VectorRankError
```

```
FUNCTION Normalized(values AS LIST OF BYTE) AS LIST OF DOUBLE FAILS ZeroVectorError
```

```
FUNCTION Normalized(values AS LIST OF DOUBLE) AS LIST OF DOUBLE FAILS ZeroVectorError
```

```
FUNCTION Normalized(values AS LIST OF INTEGER) AS LIST OF DOUBLE FAILS ZeroVectorError
```

```
FUNCTION Normalized(values AS LIST OF UINT32) AS LIST OF DOUBLE FAILS ZeroVectorError
```

```
FUNCTION Normalized(values AS LIST OF UINT64) AS LIST OF DOUBLE FAILS ZeroVectorError
```

```
FUNCTION NormalizeURL(BYVAL value AS URL,removeDefaultPort AS BOOLEAN=FALSE) AS URL FAILS
UrlError
```

```
FUNCTION NormalizeURL(text AS STRING,removeDefaultPort AS BOOLEAN=FALSE) AS STRING FAILS
UrlError
```

```
FUNCTION NowInstant() AS Instant FAILS DateTimeError
```

```
FUNCTION NowInstant(clock AS Clock) AS Instant FAILS DateTimeError
```

```
FUNCTION NowInZone(clock AS Clock,zone AS TimeZone) AS ZonedDateTime FAILS DateTimeError
```

```
FUNCTION NowMonotonic(clock AS Clock) AS MonotonicInstant FAILS DateTimeError
```

O

```
FUNCTION OffsetAt(value AS TimeZone,instant AS Instant) AS UtcOffset FAILS DateTimeError
```

```
FUNCTION OffsetOf(value AS OffsetDateTime) AS UtcOffset
```

```
FUNCTION OffsetOf(value AS OffsetTime) AS UtcOffset
```

```
FUNCTION OffsetOf(value AS ZonedDateTime) AS UtcOffset FAILS DateTimeError
```

```
FUNCTION OffsetSeconds(value AS UtcOffset) AS INTEGER
```

```
FUNCTION OpenDatabase(path AS STRING) AS DATABASE FAILS SqliteError
```

```
FUNCTION OpenDirectory(path AS STRING) AS File FAILS IoError
```

```
FUNCTION OpenFile(path AS STRING,mode AS STRING="r") AS File FAILS IoError
```

```
FUNCTION OpenFileBelow(root AS File,path AS STRING) AS File FAILS IoError,StringError
```

```
FUNCTION OpenPseudoTerminal() AS PseudoTerminal FAILS IoError
```

```
FUNCTION OpenTerminal() AS Terminal FAILS IoError
```

P

```
FUNCTION PADLEFT(text AS STRING, width AS INTEGER) AS STRING FAILS StringError
```

```
FUNCTION PADLEFTWITH(text AS STRING, width AS INTEGER, fill AS STRING) AS STRING FAILS
StringError
```

```
FUNCTION PADRIGHT(text AS STRING, width AS INTEGER) AS STRING FAILS StringError
```

```
FUNCTION PADRIGHTWITH(text AS STRING, width AS INTEGER, fill AS STRING) AS STRING FAILS
StringError
```

```
FUNCTION ParseArguments(root AS Command) AS ArgumentResult FAILS
ArgumentError,StringError,IoError
```

```
FUNCTION ParseArguments(root AS Command,arguments AS LIST OF STRING) AS ArgumentResult FAILS
ArgumentError,StringError
```

```
FUNCTION ParseArguments[T](defaults AS T) AS ArgumentResult FAILS
ArgumentError,StringError,IoError
```

```
FUNCTION ParseArguments[T](defaults AS T,arguments AS LIST OF STRING) AS ArgumentResult FAILS
ArgumentError,StringError
```

```
FUNCTION ParseDate(text AS STRING) AS Date FAILS DateTimeError
```

```
FUNCTION ParseDate(text AS STRING,options AS ParseOptions) AS Date FAILS DateTimeError
```

```
FUNCTION ParseDateTimeAmount(text AS STRING) AS DateTimeAmount FAILS DateTimeError
```

```
FUNCTION ParseDateWithPattern(text AS STRING,pattern AS DateTimePattern) AS Date FAILS
DateTimeError
```

```
FUNCTION ParseDuration(text AS STRING) AS Duration FAILS DateTimeError
```

```
FUNCTION ParseFormatSpec(text AS STRING) AS FormatSpec
```

```
FUNCTION ParseHttpDate(text AS STRING) AS Instant FAILS DateTimeError
```

```
FUNCTION ParseHTTPURL(text AS STRING) AS HTTPURL FAILS HTTPError,StringError
```

```
FUNCTION ParseHTTPURL(value AS URL) AS HTTPURL FAILS HTTPError,StringError
```

```
FUNCTION ParseInstant(text AS STRING) AS Instant FAILS DateTimeError
```

```
FUNCTION ParseInstant(text AS STRING,options AS ParseOptions) AS Instant FAILS DateTimeError
```

```
FUNCTION ParseInstantWithPattern(text AS STRING,pattern AS DateTimePattern) AS Instant FAILS
DateTimeError
```

```
FUNCTION ParseIPAddress(host AS STRING) AS IPAddress FAILS IoError
```

```
FUNCTION ParseIsoWeekDate(text AS STRING) AS IsoWeekDate FAILS DateTimeError
```

```
FUNCTION ParseJSON(bytes AS LIST OF BYTE) AS JsonValue FAILS JsonError
```

```
FUNCTION ParseJSON(bytes AS LIST OF BYTE,codec AS JsonCodec) AS JsonValue FAILS JsonError
```

```
FUNCTION ParseJSON(text AS STRING) AS JsonValue FAILS JsonError
```

```
FUNCTION ParseJSON(text AS STRING,codec AS JsonCodec) AS JsonValue FAILS JsonError
```

```
FUNCTION ParseJsonDocument(text AS STRING) AS JsonDocument FAILS JsonError
```

```
FUNCTION ParseJsonDocument(text AS STRING,codec AS JsonCodec) AS JsonDocument FAILS JsonError
```

```
FUNCTION ParseLocalDateTime(text AS STRING) AS LocalDateTime FAILS DateTimeError
```

```
FUNCTION ParseLocalDateTime(text AS STRING,options AS ParseOptions) AS LocalDateTime FAILS
DateTimeError
```

```
FUNCTION ParseLocalDateTimeWithPattern(text AS STRING,pattern AS DateTimePattern) AS
LocalDateTime FAILS DateTimeError
```

```
FUNCTION ParseLogPattern(pattern AS STRING) AS LIST OF LogPatternPart FAILS LogFormatError
```

```
FUNCTION ParseMonthDay(text AS STRING) AS MonthDay FAILS DateTimeError
```

```
FUNCTION ParseOffset(text AS STRING) AS UtcOffset FAILS DateTimeError
```

```
FUNCTION ParseOffsetDateTime(text AS STRING) AS OffsetDateTime FAILS DateTimeError
```

```
FUNCTION ParseOffsetDateTime(text AS STRING,options AS ParseOptions) AS OffsetDateTime FAILS
DateTimeError
```

```
FUNCTION ParseOffsetDateTimeWithPattern(text AS STRING,pattern AS DateTimePattern) AS
OffsetDateTime FAILS DateTimeError
```

```
FUNCTION ParseOffsetTime(text AS STRING) AS OffsetTime FAILS DateTimeError
```

```
FUNCTION ParsePeriod(text AS STRING) AS Period FAILS DateTimeError
```

```
FUNCTION ParseRfc3339(text AS STRING) AS ParsedTimestamp FAILS DateTimeError
```

```
FUNCTION ParseRfc9557(text AS STRING,database AS ZoneDatabase) AS ParsedZonedTimestamp FAILS
DateTimeError
```

```
FUNCTION ParseRfc9557(text AS STRING,database AS ZoneDatabase,conflict AS
OffsetConflictPolicy,options AS LocalResolveOptions) AS ParsedZonedTimestamp FAILS
DateTimeError
```

```
FUNCTION ParseTime(text AS STRING) AS TimeOfDay FAILS DateTimeError
```

```
FUNCTION ParseTime(text AS STRING,options AS ParseOptions) AS TimeOfDay FAILS DateTimeError
```

```
FUNCTION ParseTimeWithPattern(text AS STRING,pattern AS DateTimePattern) AS TimeOfDay FAILS
DateTimeError
```

```
FUNCTION ParseURL(text AS STRING) AS URL FAILS UrlError
```

```
FUNCTION ParseURLQuery(text AS STRING,form AS BOOLEAN=TRUE) AS LIST OF URLParameter FAILS
UrlError
```

```
FUNCTION ParseUUID(text AS STRING) AS UUID FAILS UuidError
```

```
FUNCTION ParseYearMonth(text AS STRING) AS YearMonth FAILS DateTimeError
```

```
FUNCTION ParseZonedDateTime(text AS STRING,database AS ZoneDatabase) AS ZonedDateTime FAILS
DateTimeError
```

```
FUNCTION ParseZonedDateTime(text AS STRING,database AS ZoneDatabase,conflict AS
OffsetConflictPolicy,options AS LocalResolveOptions) AS ZonedDateTime FAILS DateTimeError
```

```
FUNCTION Parts(value AS Date) AS DateParts
```

```
FUNCTION Peek[T](q AS PriorityQueue[T]) AS BORROW T FROM q FAILS EmptyCollection WHERE T IS
RESOURCE
```

```
FUNCTION Peek[T](q AS PriorityQueue[T]) AS T FAILS EmptyCollection WHERE T IS COPYABLE
```

```
FUNCTION PeekPriority[T](q AS PriorityQueue[T]) AS INTEGER FAILS EmptyCollection
```

```
FUNCTION PeerPort(socket AS TCPConnection) AS INTEGER FAILS IoError
```

```
FUNCTION PendingBytes[T](writer AS BufferedWriter[T]) AS LIST OF BYTE FAILS IndexError WHERE
T IS RESOURCE
```

```
FUNCTION PeriodDays(value AS INTEGER) AS Period
```

```
FUNCTION PeriodDaysPart(value AS Period) AS INTEGER
```

```
FUNCTION PeriodFromStorageJSON(value AS JsonValue) AS Period FAILS DateTimeError
```

```
FUNCTION PeriodMonths(value AS INTEGER) AS Period
```

```
FUNCTION PeriodMonthsPart(value AS Period) AS INTEGER
```

```
FUNCTION PeriodToStorageJSON(value AS Period) AS JsonValue FAILS DateTimeError
```

```
FUNCTION PeriodWeeks(value AS INTEGER) AS Period FAILS DateTimeError
```

```
FUNCTION PeriodYears(value AS INTEGER) AS Period FAILS DateTimeError
```

```
FUNCTION PI() AS DOUBLE
```

```
FUNCTION Pop[T](BYREF items AS LIST OF T) AS T FAILS EmptyCollection
```

```
FUNCTION PopBack[T](BYREF q AS BoundedDeque[T]) AS T FAILS EmptyCollection
```

```
FUNCTION PopBack[T](BYREF q AS Deque[T]) AS T FAILS EmptyCollection
```

```
FUNCTION PopFront[T](BYREF q AS BoundedDeque[T]) AS T FAILS EmptyCollection
```

```
FUNCTION PopFront[T](BYREF q AS Deque[T]) AS T FAILS EmptyCollection
```

```
FUNCTION PopPriority[T](BYREF q AS PriorityQueue[T]) AS T FAILS EmptyCollection
```

```
FUNCTION Position(reader AS ByteReader) AS INTEGER
```

```
FUNCTION POW(base AS DOUBLE, exponent AS DOUBLE) AS DOUBLE
```

```
FUNCTION PreviousTransition(value AS TimeZone,instant AS Instant) AS OPTIONAL OF
ZoneTransition FAILS DateTimeError
```

```
FUNCTION PrintArgumentHelp(help AS ArgumentHelp) AS INTEGER FAILS IoError
```

```
FUNCTION ProcessExited(process AS Process) AS BOOLEAN FAILS IoError
```

```
FUNCTION PruneExpired[K,V](BYREF cache AS Cache[K,V]) AS INTEGER FAILS
CollectionError,DateTimeError,IoError,LIKE ValidateKey(AS K)
```

```
FUNCTION PruneExpiredAt[K,V](BYREF cache AS Cache[K,V],now AS MonotonicInstant) AS INTEGER
FAILS CollectionError,DateTimeError,IoError,LIKE ValidateKey(AS K)
```

```
FUNCTION PseudoTerminalName(terminal AS PseudoTerminal) AS STRING FAILS IoError
```

```
SUB Push[T](BYREF q AS PriorityQueue[T],priority AS INTEGER,BYVAL value AS T)
```

```
FUNCTION PushBack[T](BYREF q AS BoundedDeque[T],BYVAL value AS T) AS OPTIONAL OF T FAILS
EmptyCollection,CollectionError
```

```
SUB PushBack[T](BYREF q AS Deque[T],BYVAL value AS T)
```

```
FUNCTION PushFront[T](BYREF q AS BoundedDeque[T],BYVAL value AS T) AS OPTIONAL OF T FAILS
EmptyCollection,CollectionError
```

```
SUB PushFront[T](BYREF q AS Deque[T],BYVAL value AS T)
```

```
FUNCTION Put[K,V](BYREF table AS DICT OF K TO V, key AS K, BYVAL value AS V) AS BOOLEAN FAILS
LIKE ValidateKey(AS K)
```

Q

```
SUB QuadraticTo(BYREF path AS CanvasPath,controlX AS DOUBLE,controlY AS DOUBLE,x AS DOUBLE,y
AS DOUBLE) FAILS CanvasError
```

```
FUNCTION Query(db AS DATABASE,sql AS STRING) AS SqlResult FAILS SqliteError
```

```
FUNCTION Query(db AS DATABASE,sql AS STRING,parameters AS LIST OF OPTIONAL OF STRING) AS
SqlResult FAILS SqliteError
```

```
FUNCTION Query(db AS DATABASE,sql AS STRING,parameters AS LIST OF SqlValue) AS SqlTypedResult
FAILS SqliteError
```

```
FUNCTION QueryValues(db AS DATABASE,sql AS STRING) AS SqlTypedResult FAILS SqliteError
```

R

```
FUNCTION RADIANS(degrees AS DOUBLE) AS DOUBLE
```

```
SUB RANDOMIZE() FAILS IoError
```

```
SUB RANDOMIZE(seed AS INTEGER)
```

```
SUB RANDOMIZE(seed AS UINT64)
```

```
FUNCTION Range[T](s AS SortedSet[T],first AS T,finish AS T) AS LIST OF T FAILS LIKE
__ValueOrderCheck(AS T),LIKE ValidateKey(AS T)
```

```
FUNCTION RangeKeys[K,V](d AS SortedDictionary[K,V],first AS K,finish AS K) AS LIST OF K FAILS
LIKE __ValueOrderCheck(AS K),LIKE ValidateKey(AS K)
```

```
FUNCTION Rank[T](items AS ARRAY OF T) AS INTEGER
```

```
FUNCTION ReadBinaryFile(path AS STRING) AS LIST OF BYTE FAILS IoError
```

```
FUNCTION ReadByte() AS INTEGER FAILS IoError
```

```
FUNCTION ReadByte(BYREF reader AS ByteReader) AS BYTE FAILS IndexError
```

```
FUNCTION ReadByte[T](BYREF reader AS BufferedReader[T]) AS OPTIONAL OF BYTE FAILS
IoError,IndexError WHERE T IS RESOURCE
```

```
FUNCTION ReadBytes(BYREF reader AS ByteReader,count AS INTEGER) AS LIST OF BYTE FAILS
IndexError
```

```
FUNCTION ReadBytes[T](BYREF reader AS BufferedReader[T],maximum AS INTEGER) AS LIST OF BYTE
FAILS IoError,IndexError WHERE T IS RESOURCE
```

```
FUNCTION ReadDirectory(file AS File) AS LIST OF DirectoryEntry FAILS
IoError,IndexError,StringError
```

```
FUNCTION ReadDirectory(path AS STRING) AS LIST OF DirectoryEntry FAILS
IoError,IndexError,StringError
```

```
FUNCTION ReadDouble(BYREF reader AS ByteReader,Order AS ByteOrder=ByteOrder.LittleEndian) AS
DOUBLE FAILS IndexError
```

```
FUNCTION ReadFileBytes(path AS STRING) AS LIST OF INTEGER FAILS IoError
```

```
FUNCTION ReadFileInto(file AS File,BYREF bytes AS LIST OF BYTE,start AS INTEGER,count AS
INTEGER) AS INTEGER FAILS IoError,IndexError
```

```
FUNCTION ReadFileSnapshot(path AS STRING) AS FileSnapshot FAILS
IoError,IndexError,FileChanged
```

```
FUNCTION ReadFileText(path AS STRING) AS STRING FAILS IoError,StringError
```

```
FUNCTION ReadFontFaceCount(path AS STRING,maxBytes AS INTEGER=33554432) AS INTEGER FAILS
CanvasError,IndexError,NumericError,IoError
```

```
FUNCTION ReadFontInfo(path AS STRING,maxBytes AS INTEGER=33554432,faceIndex AS INTEGER=0) AS
FontInfo FAILS CanvasError,IndexError,NumericError,StringError,IoError
```

```
FUNCTION ReadHTTPBody(connection AS TCPConnection,BYREF exchange AS HTTPExchange,maximum AS
INTEGER=8192) AS LIST OF BYTE FAILS IoError,HTTPError,IndexError,StringError
```

```
FUNCTION ReadHTTPBodyAll(connection AS TCPConnection,BYREF exchange AS HTTPExchange) AS LIST
OF BYTE FAILS IoError,HTTPError,IndexError,StringError
```

```
FUNCTION ReadHTTPRequest(connection AS TCPConnection,BYREF exchange AS HTTPExchange) AS
OPTIONAL OF HTTPRequest FAILS IoError,HTTPError,IndexError,StringError
```

```
FUNCTION ReadHTTPResponse(connection AS TCPConnection,BYREF exchange AS HTTPExchange) AS
HTTPResponse FAILS IoError,HTTPError,IndexError,StringError
```

```
FUNCTION ReadInt16(BYREF reader AS ByteReader,Order AS ByteOrder=ByteOrder.LittleEndian) AS
INTEGER FAILS IndexError
```

```
FUNCTION ReadInt32(BYREF reader AS ByteReader,Order AS ByteOrder=ByteOrder.LittleEndian) AS
INTEGER FAILS IndexError
```

```
FUNCTION ReadInt64(BYREF reader AS ByteReader,Order AS ByteOrder=ByteOrder.LittleEndian) AS
INTEGER FAILS IndexError
```

```
FUNCTION ReadInt8(BYREF reader AS ByteReader) AS INTEGER FAILS IndexError
```

```
FUNCTION ReadInto(BYREF bytes AS LIST OF BYTE, start AS INTEGER, count AS INTEGER) AS INTEGER
FAILS IoError,IndexError
```

```
FUNCTION ReadInto(BYREF bytes AS LIST OF INTEGER, start AS INTEGER, count AS INTEGER) AS
INTEGER FAILS IoError,IndexError
```

```
FUNCTION ReadInto(BYREF reader AS ByteReader,BYREF bytes AS LIST OF BYTE,start AS
INTEGER,count AS INTEGER) AS INTEGER FAILS IndexError
```

```
FUNCTION ReadInto[T](BYREF reader AS BufferedReader[T],BYREF bytes AS LIST OF BYTE,start AS
INTEGER,count AS INTEGER) AS INTEGER FAILS IoError,IndexError WHERE T IS RESOURCE
```

```
FUNCTION ReadJSON[T](file AS File) AS T FAILS IoError,IndexError,JsonError,LIKE
__JsonDecodeErrors(AS T)
```

```
FUNCTION ReadJSON[T](file AS File,codec AS JsonCodec) AS T FAILS
IoError,IndexError,JsonError,LIKE codec, LIKE __JsonDecodeErrors(AS T)
```

```
FUNCTION ReadJSONBody[T](request AS HTTPRequest) AS T FAILS HTTPError,JsonError,LIKE
request,LIKE __JsonDecodeErrors(AS T)
```

```
FUNCTION ReadJSONBody[T](request AS HTTPRequest,codec AS JsonCodec) AS T FAILS
HTTPError,JsonError,LIKE codec, LIKE __JsonDecodeErrors(AS T)
```

```
FUNCTION ReadJSONBody[T](request AS HTTPRequest,connection AS TCPConnection,BYREF exchange AS
HTTPExchange) AS T FAILS IoError,HTTPError,IndexError,StringError,JsonError,LIKE
__JsonDecodeErrors(AS T)
```

```
FUNCTION ReadJSONBody[T](request AS HTTPRequest,connection AS TCPConnection,BYREF exchange AS
HTTPExchange,codec AS JsonCodec) AS T FAILS
IoError,HTTPError,IndexError,StringError,JsonError,LIKE codec, LIKE __JsonDecodeErrors(AS T)
```

```
FUNCTION ReadJSONFile[T](path AS STRING) AS T FAILS IoError,IndexError,JsonError,LIKE
__JsonDecodeErrors(AS T)
```

```
FUNCTION ReadJSONFile[T](path AS STRING,codec AS JsonCodec) AS T FAILS
IoError,IndexError,JsonError,LIKE codec, LIKE __JsonDecodeErrors(AS T)
```

```
FUNCTION ReadJSONLine[T](file AS File,BYREF reader AS JsonLinesReader) AS OPTIONAL OF T FAILS
IoError,IndexError,JsonError,LIKE __JsonDecodeErrors(AS T)
```

```
FUNCTION ReadJSONLine[T](file AS File,BYREF reader AS JsonLinesReader,codec AS JsonCodec) AS
OPTIONAL OF T FAILS IoError,IndexError,JsonError,LIKE codec, LIKE __JsonDecodeErrors(AS T)
```

```
FUNCTION ReadJSONResponse[T](response AS HTTPResponse) AS OPTIONAL OF T FAILS
HTTPStatusError,HTTPError,JsonError,LIKE __JsonDecodeErrors(AS T)
```

```
FUNCTION ReadJSONResponse[T](response AS HTTPResponse,codec AS JsonCodec) AS OPTIONAL OF T
FAILS HTTPStatusError,HTTPError,JsonError,LIKE codec, LIKE __JsonDecodeErrors(AS T)
```

```
FUNCTION ReadLine() AS OPTIONAL OF STRING FAILS IoError,StringError
```

```
FUNCTION ReadLine[T](BYREF reader AS BufferedReader[T],MaxBytes AS INTEGER=1048576) AS
OPTIONAL OF STRING FAILS IoError,IndexError,StringError WHERE T IS RESOURCE
```

```
FUNCTION ReadProcessInto(process AS Process,stream AS INTEGER,BYREF bytes AS LIST OF
BYTE,start AS INTEGER,count AS INTEGER) AS INTEGER FAILS IoError,IndexError
```

```
FUNCTION ReadTCP(connection AS TCPConnection,maximum AS INTEGER=8192) AS LIST OF BYTE FAILS
IoError,IndexError
```

```
FUNCTION ReadTCPInto(connection AS TCPConnection,BYREF bytes AS LIST OF BYTE,start AS
INTEGER,count AS INTEGER) AS INTEGER FAILS IoError,IndexError
```

```
FUNCTION ReadTCPInto(connection AS TCPConnection,BYREF bytes AS LIST OF BYTE,start AS
INTEGER,count AS INTEGER,timeout AS Duration) AS INTEGER FAILS IoError,IndexError
```

```
FUNCTION ReadTerminalEvent(terminal AS Terminal) AS TerminalEvent FAILS IoError,StringError
```

```
FUNCTION ReadTerminalEvent(terminal AS Terminal,timeout AS Duration) AS OPTIONAL OF
TerminalEvent FAILS IoError,StringError
```

```
FUNCTION ReadUInt16(BYREF reader AS ByteReader,Order AS ByteOrder=ByteOrder.LittleEndian) AS
UINT32 FAILS IndexError
```

```
FUNCTION ReadUInt16LE(bytes AS LIST OF BYTE,offset AS INTEGER) AS UINT32 FAILS IndexError
```

```
FUNCTION ReadUInt32(BYREF reader AS ByteReader,Order AS ByteOrder=ByteOrder.LittleEndian) AS
UINT32 FAILS IndexError
```

```
FUNCTION ReadUInt32LE(bytes AS LIST OF BYTE,offset AS INTEGER) AS UINT32 FAILS IndexError
```

```
FUNCTION ReadUInt64(BYREF reader AS ByteReader,Order AS ByteOrder=ByteOrder.LittleEndian) AS
UINT64 FAILS IndexError
```

```
FUNCTION ReadUInt64LE(bytes AS LIST OF BYTE,offset AS INTEGER) AS UINT64 FAILS IndexError
```

```
FUNCTION RealArgument(values AS DICT OF STRING TO ArgumentValue,name AS STRING) AS DOUBLE
FAILS ArgumentError
```

```
FUNCTION RealOption(name AS STRING,Default AS DOUBLE=0.0,Short AS STRING="",Help AS
STRING="",Required AS BOOLEAN=FALSE,Global AS BOOLEAN=FALSE) AS CommandOption
```

```
FUNCTION Receive[T](port AS RECEIVER OF T) AS OPTIONAL OF T
```

```
FUNCTION RedactedURL(value AS URL) AS STRING
```

```
FUNCTION RedirectResponse(location AS STRING,status AS INTEGER=302) AS HTTPResponse
```

```
FUNCTION RegexFind(pattern AS Regex,text AS STRING) AS OPTIONAL OF RegexMatch FAILS
RegexError
```

```
FUNCTION RegexFind(pattern AS STRING,text AS STRING,Flags AS STRING="",Dialect AS
RegexDialect=RegexDialect.Native,Limits AS RegexLimits=RegexLimits {}) AS OPTIONAL OF
RegexMatch FAILS RegexError
```

```
FUNCTION RegexFindAll(pattern AS Regex,text AS STRING) AS LIST OF RegexMatch FAILS RegexError
```

```
FUNCTION RegexFindAll(pattern AS STRING,text AS STRING,Flags AS STRING="",Dialect AS
RegexDialect=RegexDialect.Native,Limits AS RegexLimits=RegexLimits {}) AS LIST OF RegexMatch
FAILS RegexError
```

```
FUNCTION RegexIsMatch(pattern AS Regex,text AS STRING) AS BOOLEAN FAILS RegexError
```

```
FUNCTION RegexIsMatch(pattern AS STRING,text AS STRING,Flags AS STRING="",Dialect AS
RegexDialect=RegexDialect.Native,Limits AS RegexLimits=RegexLimits {}) AS BOOLEAN FAILS
RegexError
```

```
FUNCTION RegexReplace(pattern AS Regex,text AS STRING,replacement AS STRING,all AS
BOOLEAN=TRUE) AS STRING FAILS RegexError
```

```
FUNCTION RegexReplace(pattern AS STRING,text AS STRING,replacement AS STRING,all AS
BOOLEAN=TRUE,Flags AS STRING="",Dialect AS RegexDialect=RegexDialect.Native,Limits AS
RegexLimits=RegexLimits {}) AS STRING FAILS RegexError
```

```
FUNCTION RegexSplit(pattern AS Regex,text AS STRING) AS LIST OF STRING FAILS RegexError
```

```
FUNCTION RegexSplit(pattern AS STRING,text AS STRING,Flags AS STRING="",Dialect AS
RegexDialect=RegexDialect.Native,Limits AS RegexLimits=RegexLimits {}) AS LIST OF STRING
FAILS RegexError
```

```
FUNCTION REMAINDER(x AS DOUBLE, y AS DOUBLE) AS DOUBLE
```

```
FUNCTION Remaining(clock AS Clock,deadline AS Deadline) AS Duration FAILS DateTimeError
```

```
FUNCTION Remaining(reader AS ByteReader) AS INTEGER
```

```
FUNCTION Remove[K,V](BYREF table AS DICT OF K TO V, key AS K) AS BOOLEAN FAILS LIKE
ValidateKey(AS K)
```

```
FUNCTION Remove[K](BYREF table AS SET OF K, key AS K) AS BOOLEAN FAILS LIKE ValidateKey(AS K)
```

```
FUNCTION Remove[T](BYREF items AS LIST OF T, value AS T) AS BOOLEAN
```

```
FUNCTION RemoveAt[T](BYREF items AS LIST OF T, index AS INTEGER) AS INTEGER FAILS IndexError
```

```
FUNCTION RemoveKey[K,V](BYREF m AS MultiMap[K,V],key AS K) AS INTEGER FAILS MissingKey,LIKE
ValidateKey(AS K)
```

```
FUNCTION RemoveOne[K,V](BYREF m AS MultiMap[K,V],key AS K,value AS V) AS BOOLEAN FAILS
MissingKey,LIKE ValidateKey(AS K) WHERE V IS COPYABLE
```

```
FUNCTION REMOVEPREFIX(text AS STRING, prefix AS STRING) AS STRING
```

```
FUNCTION REMOVESUFFIX(text AS STRING, suffix AS STRING) AS STRING
```

```
FUNCTION RemoveURLQueryParameter(BYREF parameters AS LIST OF URLParameter,name AS STRING) AS
INTEGER
```

```
FUNCTION RemoveURLQueryParameter(BYREF value AS URL,name AS STRING) AS INTEGER FAILS UrlError
```

```
FUNCTION RemoveValue[K,V](BYREF d AS SortedDictionary[K,V],key AS K) AS BOOLEAN FAILS
IndexError,LIKE __ValueOrderCheck(AS K),LIKE ValidateKey(AS K)
```

```
FUNCTION RemoveValue[T](BYREF c AS Counter[T],key AS T,amount AS INTEGER=1) AS INTEGER FAILS
CollectionError,LIKE ValidateKey(AS T)
```

```
FUNCTION RemoveValue[T](BYREF s AS SortedSet[T],value AS T) AS BOOLEAN FAILS IndexError,LIKE
__ValueOrderCheck(AS T),LIKE ValidateKey(AS T)
```

```
FUNCTION RenderHelp(root AS Command) AS STRING FAILS StringError
```

```
FUNCTION RenderHelp(root AS Command,path AS LIST OF STRING) AS STRING FAILS StringError
```

```
FUNCTION REPEAT(text AS STRING, count AS INTEGER) AS STRING FAILS StringError
```

```
FUNCTION REPLACE(text AS STRING, old AS STRING, replacement AS STRING) AS STRING
```

```
FUNCTION ReplaceOffsetKeepingLocal(value AS OffsetDateTime,offset AS UtcOffset) AS
OffsetDateTime FAILS DateTimeError
```

```
FUNCTION ReplaceZoneKeepingLocal(value AS ZonedDateTime,zone AS TimeZone) AS ZonedDateTime
FAILS DateTimeError
```

```
FUNCTION ReplaceZoneKeepingLocal(value AS ZonedDateTime,zone AS TimeZone,options AS
LocalResolveOptions) AS ZonedDateTime FAILS DateTimeError
```

```
FUNCTION ReportArgumentError(failure AS ArgumentError,prefix AS STRING="") AS INTEGER FAILS
IoError,StringError
```

```
FUNCTION RequestHTTP(method AS STRING,url AS STRING,headers AS LIST OF HTTPHeader,body AS
LIST OF BYTE,options AS HTTPClientOptions) AS HTTPResponse FAILS
IoError,HTTPError,TLSError,IndexError,StringError
```

```
FUNCTION RequestHTTP(method AS STRING,url AS URL,headers AS LIST OF HTTPHeader,body AS LIST
OF BYTE,options AS HTTPClientOptions) AS HTTPResponse FAILS
IoError,HTTPError,TLSError,IndexError,StringError
```

```
FUNCTION RequestPlainHTTP(method AS STRING,url AS STRING,headers AS LIST OF HTTPHeader,body
AS LIST OF BYTE,options AS HTTPClientOptions) AS HTTPResponse FAILS
IoError,HTTPError,IndexError,StringError
```

```
FUNCTION RequestPlainHTTP(method AS STRING,url AS URL,headers AS LIST OF HTTPHeader,body AS
LIST OF BYTE,options AS HTTPClientOptions) AS HTTPResponse FAILS
IoError,HTTPError,IndexError,StringError
```

```
FUNCTION Reserve(BYREF buffer AS ByteBuffer,capacity AS INTEGER) AS INTEGER FAILS IndexError
```

```
FUNCTION Reserve(BYREF builder AS StringBuilder,capacity AS INTEGER) AS INTEGER FAILS
IndexError
```

```
FUNCTION Reserve(BYREF writer AS ByteWriter,capacity AS INTEGER) AS INTEGER FAILS IndexError
```

```
FUNCTION Reserve[T](BYREF items AS LIST OF T, capacity AS INTEGER) AS INTEGER FAILS
IndexError
```

```
SUB Reset(BYREF bits AS BitSet)
```

```
SUB Reset[K,V](BYREF cache AS Cache[K,V])
```

```
SUB Reset[K,V](BYREF d AS SortedDictionary[K,V])
```

```
SUB Reset[K,V](BYREF m AS MultiMap[K,V])
```

```
SUB Reset[T](BYREF c AS Counter[T])
```

```
SUB Reset[T](BYREF q AS BoundedDeque[T])
```

```
SUB Reset[T](BYREF q AS Deque[T])
```

```
SUB Reset[T](BYREF q AS PriorityQueue[T])
```

```
SUB Reset[T](BYREF q AS Queue[T])
```

```
SUB Reset[T](BYREF s AS SortedSet[T])
```

```
FUNCTION ResidentCount[K,V](cache AS Cache[K,V]) AS INTEGER
```

```
FUNCTION ResidentWeight[K,V](cache AS Cache[K,V]) AS INTEGER
```

```
FUNCTION Resize[T](BYREF items AS LIST OF T, length AS INTEGER, fill AS T) AS INTEGER FAILS
IndexError
```

```
FUNCTION ResizeCanvas(canvas AS Canvas,width AS INTEGER,height AS INTEGER,sampling AS
Sampling=Sampling.Bilinear,space AS BlendSpace=BlendSpace.EncodedSRGB,maxPixels AS
INTEGER=16777216,maxWorkingBytes AS INTEGER=268435456) AS Canvas FAILS
CanvasError,IndexError,NumericError
```

```
FUNCTION ResolveBidiLevels(text AS STRING,direction AS TextDirection=TextDirection.Auto) AS
LIST OF INTEGER FAILS CanvasError,IndexError,NumericError
```

```
FUNCTION ResolveHost(host AS STRING) AS LIST OF IPAddress FAILS IoError,ResolveError
```

```
FUNCTION ResolveHost(host AS STRING,timeout AS Duration) AS LIST OF IPAddress FAILS
IoError,ResolveError
```

```
FUNCTION ResolveLocal(zone AS TimeZone,local AS LocalDateTime) AS ZonedDateTime FAILS
DateTimeError
```

```
FUNCTION ResolveLocal(zone AS TimeZone,local AS LocalDateTime,options AS LocalResolveOptions)
AS ZonedDateTime FAILS DateTimeError
```

```
FUNCTION ResolveURL(base AS STRING,reference AS STRING) AS STRING FAILS UrlError
```

```
FUNCTION ResolveURL(base AS URL,BYVAL reference AS URL) AS URL FAILS UrlError
```

```
FUNCTION ResolveURL(base AS URL,reference AS STRING) AS URL FAILS UrlError
```

```
FUNCTION ResumeTerminal(terminal AS Terminal,screen AS BOOLEAN=TRUE) AS INTEGER FAILS IoError
```

```
FUNCTION Reverse[T](BYREF items AS LIST OF T) AS INTEGER
```

```
FUNCTION Reversed(text AS STRING) AS STRING
```

```
FUNCTION Reversed[T](items AS LIST OF T) AS LIST OF T
```

```
FUNCTION RGB(red AS INTEGER,green AS INTEGER,blue AS INTEGER) AS Color FAILS
CanvasError,NumericError
```

```
FUNCTION RGBA(red AS INTEGER,green AS INTEGER,blue AS INTEGER,alpha AS INTEGER=255) AS Color
FAILS CanvasError,NumericError
```

```
FUNCTION RIGHT(text AS STRING, count AS INTEGER) AS STRING FAILS StringError
```

```
FUNCTION RINSTR(text AS STRING, needle AS STRING) AS INTEGER
```

```
FUNCTION RND(mode AS INTEGER=1) AS DOUBLE
```

```
FUNCTION RotateCanvas(source AS Canvas,degrees AS DOUBLE,expand AS BOOLEAN=TRUE,sampling AS
Sampling=Sampling.Bilinear,space AS BlendSpace=BlendSpace.EncodedSRGB,maxPixels AS
INTEGER=16777216) AS Canvas FAILS CanvasError,IndexError,NumericError
```

```
FUNCTION RotateCanvas90(canvas AS Canvas,clockwiseTurns AS INTEGER=1) AS Canvas FAILS
CanvasError,IndexError,NumericError
```

```
FUNCTION RotateLeft(value AS UINT64, count AS INTEGER) AS UINT64
```

```
FUNCTION RotateRight(value AS UINT64, count AS INTEGER) AS UINT64
```

```
FUNCTION RotationTransform(degrees AS DOUBLE) AS AffineTransform FAILS CanvasError
```

```
FUNCTION ROUND(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION ROUND(value AS DOUBLE, places AS INTEGER) AS DOUBLE
```

```
FUNCTION RoundDuration(value AS Duration,unit AS FixedUnit,increment AS INTEGER=1,rounding AS
RoundingMode=RoundingMode.RejectLoss) AS Duration FAILS DateTimeError
```

```
FUNCTION RoundTime(value AS TimeOfDay,unit AS FixedUnit,increment AS INTEGER=1,rounding AS
RoundingMode=RoundingMode.RejectLoss) AS TimeShift FAILS DateTimeError
```

```
FUNCTION Route(BYREF router AS HTTPRouter,method AS STRING,pattern AS STRING,handler AS
CALLABLE(HTTPRequest) AS HTTPResponse FAILS (Error)) AS INTEGER FAILS HTTPError,StringError
```

```
FUNCTION Route(BYREF router AS HTTPRouter,method AS STRING,pattern AS STRING,handler AS
CALLABLE(HTTPRequest) AS HTTPResponse FAILS (Error),values AS DICT OF STRING TO STRING) AS
INTEGER FAILS HTTPError,StringError
```

```
FUNCTION Route(BYREF router AS HTTPRouter,method AS STRING,pattern AS STRING,handler AS
CALLABLE(HTTPRequest,TCPConnection,BYREF HTTPExchange) FAILS (Error)) AS INTEGER FAILS
HTTPError,StringError
```

```
FUNCTION Route(BYREF router AS HTTPRouter,method AS STRING,pattern AS STRING,handler AS
CALLABLE(HTTPRequest,TCPConnection,BYREF HTTPExchange) FAILS (Error),values AS DICT OF STRING
TO STRING) AS INTEGER FAILS HTTPError,StringError
```

```
FUNCTION RouteGet(BYREF router AS HTTPRouter,pattern AS STRING,handler AS
CALLABLE(HTTPRequest) AS HTTPResponse FAILS (Error)) AS INTEGER FAILS HTTPError,StringError
```

```
FUNCTION RouteGet(BYREF router AS HTTPRouter,pattern AS STRING,handler AS
CALLABLE(HTTPRequest,TCPConnection,BYREF HTTPExchange) FAILS (Error)) AS INTEGER FAILS
HTTPError,StringError
```

```
FUNCTION RoutePost(BYREF router AS HTTPRouter,pattern AS STRING,handler AS
CALLABLE(HTTPRequest) AS HTTPResponse FAILS (Error)) AS INTEGER FAILS HTTPError,StringError
```

```
FUNCTION RoutePost(BYREF router AS HTTPRouter,pattern AS STRING,handler AS
CALLABLE(HTTPRequest,TCPConnection,BYREF HTTPExchange) FAILS (Error)) AS INTEGER FAILS
HTTPError,StringError
```

```
FUNCTION RTRIM(text AS STRING) AS STRING
```

```
FUNCTION RunInteractive(executable AS STRING,arguments AS LIST OF STRING,directory AS
STRING=".") AS INTEGER FAILS IoError
```

```
FUNCTION RunProcess(executable AS STRING,arguments AS LIST OF STRING,input AS
STRING="",directory AS STRING=".") AS ProcessResult FAILS IoError,IndexError
```

S

```
FUNCTION SameFileVersion(left AS FileInfo,right AS FileInfo) AS BOOLEAN
```

```
FUNCTION SameInstant(left AS OffsetDateTime,right AS OffsetDateTime) AS BOOLEAN FAILS
DateTimeError
```

```
FUNCTION SameInstant(left AS ZonedDateTime,right AS ZonedDateTime) AS BOOLEAN
```

```
FUNCTION SameLocalDateTime(left AS LocalDateTime,right AS LocalDateTime) AS BOOLEAN
```

```
FUNCTION SameZoneRules(left AS TimeZone,right AS TimeZone) AS BOOLEAN
```

```
FUNCTION SamplePixel(canvas AS Canvas,x AS DOUBLE,y AS DOUBLE,sampling AS
Sampling=Sampling.Bilinear,space AS BlendSpace=BlendSpace.EncodedSRGB) AS Color FAILS
CanvasError,IndexError,NumericError
```

```
SUB SaveAnimation(animation AS Animation,path AS STRING,format AS
ImageFormat=ImageFormat.Auto,maxEncodedBytes AS INTEGER=67108864) FAILS
CanvasError,IndexError,NumericError,StringError,IoError
```

```
SUB SaveImage(canvas AS Canvas,path AS STRING,format AS ImageFormat=ImageFormat.Auto) FAILS
CanvasError,IndexError,NumericError,StringError,IoError
```

```
FUNCTION Scaled(values AS ARRAY OF BYTE,factor AS BYTE) AS LIST OF BYTE FAILS
VectorRankError,NumericError
```

```
FUNCTION Scaled(values AS ARRAY OF DOUBLE,factor AS DOUBLE) AS LIST OF DOUBLE FAILS
VectorRankError
```

```
FUNCTION Scaled(values AS ARRAY OF INTEGER,factor AS INTEGER) AS LIST OF INTEGER FAILS
VectorRankError,NumericError
```

```
FUNCTION Scaled(values AS ARRAY OF UINT32,factor AS UINT32) AS LIST OF UINT32 FAILS
VectorRankError,NumericError
```

```
FUNCTION Scaled(values AS ARRAY OF UINT64,factor AS UINT64) AS LIST OF UINT64 FAILS
VectorRankError,NumericError
```

```
FUNCTION Scaled(values AS LIST OF BYTE,factor AS BYTE) AS LIST OF BYTE FAILS NumericError
```

```
FUNCTION Scaled(values AS LIST OF DOUBLE,factor AS DOUBLE) AS LIST OF DOUBLE
```

```
FUNCTION Scaled(values AS LIST OF INTEGER,factor AS INTEGER) AS LIST OF INTEGER FAILS
NumericError
```

```
FUNCTION Scaled(values AS LIST OF UINT32,factor AS UINT32) AS LIST OF UINT32 FAILS
NumericError
```

```
FUNCTION Scaled(values AS LIST OF UINT64,factor AS UINT64) AS LIST OF UINT64 FAILS
NumericError
```

```
FUNCTION ScalingTransform(x AS DOUBLE,y AS DOUBLE) AS AffineTransform FAILS CanvasError
```

```
FUNCTION Second(value AS TimeOfDay) AS INTEGER
```

```
FUNCTION Seconds(value AS INTEGER) AS Duration
```

```
FUNCTION SecureRandomBytes(count AS INTEGER) AS LIST OF BYTE FAILS IoError,IndexError
```

```
FUNCTION Seek(BYREF reader AS ByteReader,position AS INTEGER) AS INTEGER FAILS IndexError
```

```
FUNCTION SeekFile(file AS File,position AS INTEGER) AS INTEGER FAILS IoError
```

```
FUNCTION Send[T](port AS SENDER OF T, message AS T) AS INTEGER FAILS ChannelClosed
```

```
FUNCTION SendHTTPRequest(connection AS TCPConnection,BYREF exchange AS HTTPExchange,method AS
STRING,url AS HTTPURL,headers AS LIST OF HTTPHeader,body AS LIST OF BYTE) AS INTEGER FAILS
IoError,HTTPError,StringError
```

```
FUNCTION SerializationSchema[T](witness AS OPTIONAL OF T) AS SerializationSchema
```

```
FUNCTION SerializationTypeSchema[T](witness AS OPTIONAL OF T) AS TypeGraph
```

```
SUB ServeDirectory(connection AS TCPConnection,root AS STRING)
```

```
SUB ServeDirectory(connection AS TCPConnection,root AS STRING,options AS HTTPOptions)
```

```
SUB ServeDirectory(connection AS TCPConnection,root AS STRING,tls AS TLSServerOptions)
```

```
SUB ServeDirectory(connection AS TCPConnection,root AS STRING,tls AS TLSServerOptions,options
AS HTTPOptions)
```

```
SUB ServeHTTP(connection AS TCPConnection,router AS HTTPRouter)
```

```
SUB ServeHTTP(connection AS TCPConnection,router AS HTTPRouter,options AS HTTPOptions)
```

```
SUB ServeHTTP(listener AS TCPListener,router AS HTTPRouter) FAILS IoError
```

```
SUB ServeHTTP(listener AS TCPListener,router AS HTTPRouter,options AS HTTPOptions) FAILS
IoError
```

```
FUNCTION ServeHTTPChecked(connection AS TCPConnection,router AS HTTPRouter,options AS
HTTPOptions) AS INTEGER FAILS IoError,HTTPError,IndexError,StringError,JsonError,LIKE router
```

```
SUB ServeHTTPS(connection AS TCPConnection,router AS HTTPRouter,tls AS TLSServerOptions)
```

```
SUB ServeHTTPS(connection AS TCPConnection,router AS HTTPRouter,tls AS
TLSServerOptions,options AS HTTPOptions)
```

```
SUB ServeHTTPS(listener AS TCPListener,router AS HTTPRouter,tls AS TLSServerOptions) FAILS
IoError
```

```
SUB ServeHTTPS(listener AS TCPListener,router AS HTTPRouter,tls AS TLSServerOptions,options
AS HTTPOptions) FAILS IoError
```

```
FUNCTION SetDefault[K,V](BYREF table AS DICT OF K TO OPTIONAL OF V, key AS K) AS OPTIONAL OF
V FAILS LIKE ValidateKey(AS K) WHERE V IS COPYABLE
```

```
FUNCTION SetDefault[K,V](BYREF table AS DICT OF K TO OPTIONAL OF V,key AS K) AS BORROW
OPTIONAL OF V FROM table FAILS LIKE ValidateKey(AS K) WHERE V IS RESOURCE
```

```
FUNCTION SetDefault[K,V](BYREF table AS DICT OF K TO V, key AS K, fallback AS V) AS V FAILS
LIKE ValidateKey(AS K) WHERE V IS COPYABLE
```

```
FUNCTION SetDefault[K,V](BYREF table AS DICT OF K TO V,key AS K,BYVAL fallback AS V) AS
BORROW V FROM table FAILS LIKE ValidateKey(AS K) WHERE V IS RESOURCE
```

```
SUB SetDefaultLogger(BYVAL logger AS Logger) FAILS LoggingError
```

```
SUB SetDefaultLogger(logger AS LoggerHandle) FAILS LoggingError
```

```
FUNCTION SetDifference[T](left AS SortedSet[T],right AS SortedSet[T]) AS SortedSet[T] FAILS
LIKE __ValueOrderCheck(AS T),LIKE ValidateKey(AS T)
```

```
FUNCTION SetFilePermissions(path AS STRING,mode AS INTEGER) AS INTEGER FAILS IoError
```

```
FUNCTION SetIntersection[T](left AS SortedSet[T],right AS SortedSet[T]) AS SortedSet[T] FAILS
LIKE __ValueOrderCheck(AS T),LIKE ValidateKey(AS T)
```

```
SUB SetLogConfig(config AS LogConfig) FAILS LoggingError
```

```
SUB SetLogConfig(logger AS Logger,config AS LogConfig) FAILS LoggingError
```

```
SUB SetLogConfig(logger AS LoggerHandle,config AS LogConfig) FAILS LoggingError
```

```
SUB SetLogFormat(logger AS Logger,pattern AS STRING) FAILS LoggingError
```

```
SUB SetLogFormat(logger AS LoggerHandle,pattern AS STRING) FAILS LoggingError
```

```
SUB SetLogFormat(pattern AS STRING) FAILS LoggingError
```

```
SUB SetLogLevel(level AS LogLevel) FAILS LoggingError
```

```
SUB SetLogLevel(logger AS Logger,level AS LogLevel) FAILS LoggingError
```

```
SUB SetLogLevel(logger AS LoggerHandle,level AS LogLevel) FAILS LoggingError
```

```
SUB SetPixel(BYREF canvas AS Canvas,x AS INTEGER,y AS INTEGER,color AS Color) FAILS
CanvasError,IndexError,NumericError
```

```
FUNCTION SetSymmetricDifference[T](left AS SortedSet[T],right AS SortedSet[T]) AS
SortedSet[T] FAILS LIKE __ValueOrderCheck(AS T),LIKE ValidateKey(AS T)
```

```
FUNCTION SetUnion[T](left AS SortedSet[T],right AS SortedSet[T]) AS SortedSet[T] FAILS LIKE
__ValueOrderCheck(AS T),LIKE ValidateKey(AS T)
```

```
SUB SetURLQueryParameter(BYREF parameters AS LIST OF URLParameter,name AS STRING,value AS
STRING)
```

```
SUB SetURLQueryParameter(BYREF value AS URL,name AS STRING,text AS STRING) FAILS UrlError
```

```
FUNCTION SetValue[K,V](BYREF d AS SortedDictionary[K,V],key AS K,BYVAL value AS V) AS BOOLEAN
FAILS IndexError,LIKE __ValueOrderCheck(AS K),LIKE ValidateKey(AS K)
```

```
FUNCTION SGN(value AS DOUBLE) AS INTEGER FAILS MathError
```

```
FUNCTION SGN(value AS INTEGER) AS INTEGER
```

```
FUNCTION SharpenCanvas(canvas AS Canvas) AS Canvas FAILS CanvasError,IndexError,NumericError
```

```
FUNCTION ShiftLeft(value AS UINT64, count AS INTEGER) AS UINT64
```

```
FUNCTION ShiftRight(value AS UINT64, count AS INTEGER) AS UINT64
```

```
FUNCTION ShiftTime(value AS TimeOfDay,amount AS Duration) AS TimeShift FAILS DateTimeError
```

```
FUNCTION ShiftTime(value AS TimeOfDay,nanoseconds AS INTEGER) AS TimeShift
```

```
FUNCTION ShowTerminalCursor(terminal AS Terminal,visible AS BOOLEAN) AS INTEGER FAILS IoError
```

```
FUNCTION SIGN(value AS DOUBLE) AS INTEGER FAILS MathError
```

```
FUNCTION SIGN(value AS INTEGER) AS INTEGER
```

```
FUNCTION SIN(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION SINH(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION Skip(BYREF reader AS ByteReader,count AS INTEGER) AS INTEGER FAILS IndexError
```

```
FUNCTION Slice[T](items AS LIST OF T, start AS INTEGER, finish AS INTEGER) AS LIST OF T FAILS
IndexError
```

```
FUNCTION Snapshot[T](q AS BoundedDeque[T]) AS LIST OF T WHERE T IS COPYABLE
```

```
FUNCTION Snapshot[T](q AS Deque[T]) AS LIST OF T WHERE T IS COPYABLE
```

```
FUNCTION Snapshot[T](q AS Queue[T]) AS LIST OF T WHERE T IS COPYABLE
```

```
FUNCTION Snapshot[T](s AS SortedSet[T]) AS LIST OF T
```

```
FUNCTION Sort[T,R](BYREF items AS LIST OF T,Compare AS CALLABLE(T,T) AS R) AS INTEGER FAILS
LIKE __SortResult(AS R)
```

```
FUNCTION Sort[T](BYREF items AS LIST OF T) AS INTEGER FAILS LIKE __ValueOrderCheck(AS T)
```

```
FUNCTION Sorted(text AS STRING) AS STRING
```

```
FUNCTION Sorted[R](text AS STRING, Compare AS CALLABLE(STRING, STRING) AS R) AS STRING FAILS
LIKE __SortResult(AS R)
```

```
FUNCTION Sorted[T,R](items AS LIST OF T, Compare AS CALLABLE(T,T) AS R) AS LIST OF T FAILS
LIKE __SortResult(AS R)
```

```
FUNCTION Sorted[T](items AS LIST OF T) AS LIST OF T FAILS LIKE __ValueOrderCheck(AS T)
```

```
FUNCTION SPACE(count AS INTEGER) AS STRING FAILS StringError
```

```
FUNCTION SPLIT(text AS STRING, separator AS STRING="") AS LIST OF STRING FAILS StringError
```

```
FUNCTION SqlBlob(value AS LIST OF BYTE) AS SqlValue
```

```
FUNCTION SqlBoolean(value AS BOOLEAN) AS SqlValue
```

```
FUNCTION SqlGetBlob(value AS SqlValue) AS LIST OF BYTE FAILS SqliteError
```

```
FUNCTION SqlGetBoolean(value AS SqlValue) AS BOOLEAN FAILS SqliteError
```

```
FUNCTION SqlGetInteger(value AS SqlValue) AS INTEGER FAILS SqliteError
```

```
FUNCTION SqlGetReal(value AS SqlValue) AS DOUBLE FAILS SqliteError
```

```
FUNCTION SqlGetText(value AS SqlValue) AS STRING FAILS SqliteError
```

```
FUNCTION SqlInteger(value AS INTEGER) AS SqlValue
```

```
FUNCTION SqlInteger(value AS UINT64) AS SqlValue FAILS SqliteError
```

```
FUNCTION SqlNull() AS SqlValue
```

```
FUNCTION SqlReal(value AS DOUBLE) AS SqlValue
```

```
FUNCTION SqlText(value AS STRING) AS SqlValue
```

```
FUNCTION SQR(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION SQRT(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION StartHTTPRequest(connection AS TCPConnection,BYREF exchange AS HTTPExchange,method
AS STRING,url AS HTTPURL,headers AS LIST OF HTTPHeader,length AS INTEGER=-1) AS INTEGER FAILS
IoError,HTTPError,StringError
```

```
FUNCTION StartHTTPResponse(connection AS TCPConnection,BYREF exchange AS HTTPExchange,status
AS INTEGER,headers AS LIST OF HTTPHeader) AS INTEGER FAILS IoError,HTTPError,StringError
```

```
FUNCTION StartOfDay(value AS Date,zone AS TimeZone) AS ZonedDateTime FAILS DateTimeError
```

```
FUNCTION StartOfDay(value AS Date,zone AS TimeZone,options AS LocalResolveOptions) AS
ZonedDateTime FAILS DateTimeError
```

```
FUNCTION StartOfMonth(value AS Date) AS Date FAILS DateTimeError
```

```
FUNCTION StartOfWeek(value AS Date,firstDay AS INTEGER=1) AS Date FAILS DateTimeError
```

```
FUNCTION StartOfWeek(value AS Date,firstDay AS Weekday) AS Date FAILS DateTimeError
```

```
FUNCTION StartOfYear(value AS Date) AS Date FAILS DateTimeError
```

```
FUNCTION StartProcess(executable AS STRING,arguments AS LIST OF STRING,directory AS
STRING=".",interactive AS BOOLEAN=FALSE) AS Process FAILS IoError
```

```
FUNCTION STARTSWITH(text AS STRING, prefix AS STRING) AS BOOLEAN
```

```
FUNCTION StartTLS(connection AS TCPConnection,options AS TLSClientOptions) AS INTEGER FAILS
IoError,TLSError
```

```
FUNCTION StartTLS(connection AS TCPConnection,options AS TLSClientOptions,timeout AS
Duration) AS INTEGER FAILS IoError,TLSError
```

```
FUNCTION StartTLS(connection AS TCPConnection,options AS TLSServerOptions) AS INTEGER FAILS
IoError,TLSError
```

```
FUNCTION StartTLS(connection AS TCPConnection,options AS TLSServerOptions,timeout AS
Duration) AS INTEGER FAILS IoError,TLSError
```

```
FUNCTION Stderr() AS File FAILS IoError
```

```
FUNCTION Stdin() AS File FAILS IoError
```

```
FUNCTION Stdout() AS File FAILS IoError
```

```
FUNCTION STR(value AS BYTE) AS STRING
```

```
FUNCTION STR(value AS DOUBLE) AS STRING
```

```
FUNCTION STR(value AS INTEGER) AS STRING
```

```
FUNCTION STR(value AS UINT32) AS STRING
```

```
FUNCTION STR(value AS UINT64) AS STRING
```

```
FUNCTION StringBuilder(Capacity AS INTEGER=0) AS StringBuilder FAILS IndexError
```

```
FUNCTION StringBuilder(text AS STRING) AS StringBuilder
```

```
FUNCTION StringifyJSON(value AS JsonValue) AS STRING FAILS JsonError
```

```
FUNCTION StringifyJSON(value AS JsonValue,codec AS JsonCodec) AS STRING FAILS JsonError
```

```
FUNCTION SubtractDuration(left AS Duration,right AS Duration) AS Duration FAILS DateTimeError
```

```
FUNCTION SuspendTerminal(terminal AS Terminal) AS INTEGER FAILS IoError
```

```
FUNCTION SwapAt[T](BYREF items AS LIST OF T,first AS INTEGER,second AS INTEGER) AS INTEGER
FAILS IndexError
```

```
FUNCTION SymmetricDifference[K](left AS SET OF K, right AS SET OF K) AS SET OF K
```

T

```
FUNCTION TakeBytes(BYREF buffer AS ByteBuffer) AS LIST OF BYTE
```

```
FUNCTION TakeBytes(BYREF writer AS ByteWriter) AS LIST OF BYTE
```

```
FUNCTION TakeString(BYREF builder AS StringBuilder) AS STRING
```

```
FUNCTION TAN(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION TANH(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION TAU() AS DOUBLE
```

```
FUNCTION TerminalClipText(text AS STRING,start AS INTEGER,count AS INTEGER,tabSize AS
INTEGER=4) AS STRING FAILS IndexError
```

```
FUNCTION TerminalColor(foreground AS INTEGER,background AS INTEGER) AS STRING FAILS
IndexError
```

```
FUNCTION TerminalCursor(row AS INTEGER,column AS INTEGER) AS STRING FAILS IndexError
```

```
FUNCTION TerminalTextWidth(text AS STRING,tabSize AS INTEGER=4) AS INTEGER FAILS IndexError
```

```
FUNCTION TerminateProcess(process AS Process) AS INTEGER FAILS IoError
```

```
FUNCTION TextArgument(values AS DICT OF STRING TO ArgumentValue,name AS STRING) AS STRING
FAILS ArgumentError
```

```
FUNCTION TextLayoutGlyph(layout AS TextLayout,index AS INTEGER) AS TextGlyphInfo FAILS
CanvasError,IndexError
```

```
FUNCTION TextLayoutGlyphCount(layout AS TextLayout) AS INTEGER
```

```
FUNCTION TextLayoutHeight(layout AS TextLayout) AS DOUBLE
```

```
FUNCTION TextLayoutInkBounds(layout AS TextLayout) AS OPTIONAL OF TextInkBounds FAILS
IndexError
```

```
FUNCTION TextLayoutLine(layout AS TextLayout,index AS INTEGER) AS TextLine FAILS
CanvasError,IndexError
```

```
FUNCTION TextLayoutLineCount(layout AS TextLayout) AS INTEGER
```

```
FUNCTION TextLayoutWidth(layout AS TextLayout) AS DOUBLE
```

```
FUNCTION TextOption(name AS STRING,Default AS STRING="",Short AS STRING="",Help AS
STRING="",Required AS BOOLEAN=FALSE,Global AS BOOLEAN=FALSE) AS CommandOption
```

```
FUNCTION TextResponse(text AS STRING,status AS INTEGER=200,contentType AS STRING="text/plain;
charset=utf-8") AS HTTPResponse
```

```
FUNCTION Thumbnail(canvas AS Canvas,maxWidth AS INTEGER,maxHeight AS INTEGER,allowUpscale AS
BOOLEAN=FALSE,sampling AS Sampling=Sampling.Bilinear) AS Canvas FAILS
CanvasError,IndexError,NumericError
```

```
FUNCTION TimePart(value AS LocalDateTime) AS TimeOfDay
```

```
FUNCTION TimePart(value AS OffsetTime) AS TimeOfDay
```

```
FUNCTION TLSCipherSuite(connection AS TCPConnection) AS STRING FAILS
TLSError,IndexError,StringError
```

```
FUNCTION TLSClientOptions() AS TLSClientOptions
```

```
FUNCTION TLSLastError(connection AS TCPConnection) AS STRING FAILS
TLSError,IndexError,StringError
```

```
FUNCTION TLSNegotiatedProtocol(connection AS TCPConnection) AS STRING FAILS
TLSError,IndexError,StringError
```

```
FUNCTION TLSPeerCertificate(connection AS TCPConnection) AS LIST OF BYTE FAILS
TLSError,IndexError
```

```
FUNCTION TLSProtocolVersion(connection AS TCPConnection) AS STRING FAILS
TLSError,IndexError,StringError
```

```
FUNCTION TLSServerOptions(certificate AS STRING,privateKey AS STRING) AS TLSServerOptions
```

```
FUNCTION TLSVerifyCode(connection AS TCPConnection) AS INTEGER
```

```
FUNCTION ToArray[T](BYVAL values AS LIST OF T) AS ARRAY OF T
```

```
FUNCTION ToBytes(buffer AS ByteBuffer) AS LIST OF BYTE
```

```
FUNCTION ToBytes(writer AS ByteWriter) AS LIST OF BYTE
```

```
FUNCTION ToInstant(value AS OffsetDateTime) AS Instant FAILS DateTimeError
```

```
FUNCTION ToInstant(value AS ZonedDateTime) AS Instant
```

```
FUNCTION ToIsoWeekDate(value AS Date) AS IsoWeekDate
```

```
FUNCTION ToJSON[T](value AS T) AS JsonValue FAILS JsonError,LIKE __JsonEncodeErrors(AS T)
```

```
FUNCTION ToJSON[T](value AS T,codec AS JsonCodec) AS JsonValue FAILS JsonError,LIKE codec,
LIKE __JsonEncodeErrors(AS T)
```

```
FUNCTION ToList(text AS STRING) AS LIST OF STRING
```

```
FUNCTION ToList[T](items AS LIST OF T) AS LIST OF T
```

```
FUNCTION ToList[T](items AS SET OF T) AS LIST OF T
```

```
FUNCTION ToMonthDay(value AS Date) AS MonthDay
```

```
FUNCTION ToSet(text AS STRING) AS SET OF STRING
```

```
FUNCTION ToSet[T](items AS LIST OF T) AS SET OF T FAILS LIKE ValidateKey(AS T)
```

```
FUNCTION ToSet[T](items AS SET OF T) AS SET OF T
```

```
FUNCTION ToString(builder AS StringBuilder) AS STRING
```

```
FUNCTION TotalCount[K,V](m AS MultiMap[K,V]) AS INTEGER
```

```
FUNCTION TotalCount[T](c AS Counter[T]) AS INTEGER
```

```
FUNCTION ToValue[T](value AS T) AS JsonValue FAILS JsonError,LIKE __JsonEncodeErrors(AS T)
```

```
FUNCTION ToYearMonth(value AS Date) AS YearMonth
```

```
FUNCTION TrailingZeros(value AS UINT64) AS INTEGER
```

```
FUNCTION TransformPath(path AS CanvasPath,transform AS AffineTransform) AS CanvasPath FAILS
CanvasError,IndexError
```

```
FUNCTION TransformPoint(transform AS AffineTransform,point AS Point) AS Point FAILS
CanvasError
```

```
FUNCTION TranslationTransform(x AS DOUBLE,y AS DOUBLE) AS AffineTransform FAILS CanvasError
```

```
FUNCTION Transparent() AS Color
```

```
FUNCTION TransposeCanvas(canvas AS Canvas) AS Canvas FAILS
CanvasError,IndexError,NumericError
```

```
FUNCTION TRIM(text AS STRING) AS STRING
```

```
FUNCTION TRUNC(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION Truncate(BYREF buffer AS ByteBuffer,length AS INTEGER) AS INTEGER FAILS IndexError
```

```
FUNCTION Truncate[T](BYREF items AS LIST OF T, length AS INTEGER) AS INTEGER FAILS IndexError
```

```
FUNCTION TryReadProcessInto(process AS Process,stream AS INTEGER,BYREF bytes AS LIST OF
BYTE,start AS INTEGER,count AS INTEGER) AS OPTIONAL OF INTEGER FAILS IoError,IndexError
```

```
FUNCTION TryReadPseudoTerminalInto(terminal AS PseudoTerminal,BYREF bytes AS LIST OF
BYTE,start AS INTEGER,count AS INTEGER) AS OPTIONAL OF INTEGER FAILS IoError,IndexError
```

```
FUNCTION TypeSchema[T](witness AS OPTIONAL OF T) AS TypeGraph
```

U

```
FUNCTION UBOUND[T](items AS ARRAY OF T,dimension AS INTEGER=1) AS INTEGER
```

```
FUNCTION UCASE(text AS STRING) AS STRING
```

```
FUNCTION UInt64Bits(value AS UINT64) AS INTEGER
```

```
FUNCTION UInt64FromBits(value AS INTEGER) AS UINT64
```

```
FUNCTION UInt64Max() AS UINT64
```

```
FUNCTION ULP(value AS DOUBLE) AS DOUBLE
```

```
FUNCTION Unbuffer[T](BYVAL reader AS BufferedReader[T]) AS T WHERE T IS RESOURCE
```

```
FUNCTION UnescapeURLComponent(text AS STRING,form AS BOOLEAN=FALSE) AS STRING FAILS UrlError
```

```
FUNCTION UNICODEVERSION() AS STRING
```

```
FUNCTION Union[K](left AS SET OF K, right AS SET OF K) AS SET OF K
```

```
FUNCTION UnixMicroseconds(value AS Instant,rounding AS RoundingMode=RoundingMode.RejectLoss)
AS INTEGER FAILS DateTimeError
```

```
FUNCTION UnixMilliseconds(value AS Instant,rounding AS RoundingMode=RoundingMode.RejectLoss)
AS INTEGER FAILS DateTimeError
```

```
FUNCTION UnixNanoseconds(value AS Instant) AS INTEGER FAILS DateTimeError
```

```
FUNCTION UnixSeconds(value AS Instant) AS INTEGER
```

```
FUNCTION UnixTimeNanoseconds() AS INTEGER FAILS IoError
```

```
FUNCTION UPPER(text AS STRING) AS STRING
```

```
FUNCTION URL(text AS STRING) AS URL FAILS UrlError
```

```
FUNCTION URLAuthority(value AS URL) AS STRING
```

```
FUNCTION URLFragment(value AS URL) AS STRING FAILS UrlError
```

```
FUNCTION URLHostname(value AS URL) AS STRING FAILS UrlError
```

```
FUNCTION URLParameters(values AS DICT OF STRING TO LIST OF STRING) AS LIST OF URLParameter
```

```
FUNCTION URLParameters(values AS DICT OF STRING TO STRING) AS LIST OF URLParameter
```

```
FUNCTION URLPassword(value AS URL) AS OPTIONAL OF STRING FAILS UrlError
```

```
FUNCTION URLPath(value AS URL) AS STRING FAILS UrlError
```

```
FUNCTION URLPathSegments(value AS URL) AS LIST OF STRING FAILS UrlError
```

```
FUNCTION URLPort(value AS URL) AS OPTIONAL OF INTEGER FAILS UrlError
```

```
FUNCTION URLQuery(value AS URL,form AS BOOLEAN=TRUE) AS LIST OF URLParameter FAILS UrlError
```

```
FUNCTION URLQueryValue(parameters AS LIST OF URLParameter,name AS STRING) AS OPTIONAL OF
STRING
```

```
FUNCTION URLQueryValue(value AS URL,name AS STRING) AS OPTIONAL OF STRING FAILS UrlError
```

```
FUNCTION URLQueryValueOr(parameters AS LIST OF URLParameter,name AS STRING,fallback AS
STRING) AS STRING
```

```
FUNCTION URLQueryValueOr(value AS URL,name AS STRING,fallback AS STRING) AS STRING FAILS
UrlError
```

```
FUNCTION URLQueryValues(parameters AS LIST OF URLParameter,name AS STRING) AS LIST OF STRING
```

```
FUNCTION URLQueryValues(value AS URL,name AS STRING) AS LIST OF STRING FAILS UrlError
```

```
FUNCTION URLUsername(value AS URL) AS OPTIONAL OF STRING FAILS UrlError
```

```
FUNCTION UseHTTP(BYREF router AS HTTPRouter,handler AS
CALLABLE(HTTPRequest,TCPConnection,BYREF HTTPExchange) FAILS (Error)) AS INTEGER
```

```
FUNCTION UtcZone() AS TimeZone
```

```
FUNCTION UUID(bytes AS LIST OF BYTE) AS UUID FAILS UuidError
```

```
FUNCTION UUID(high AS INTEGER,low AS INTEGER) AS UUID
```

```
FUNCTION UUID(high AS INTEGER,low AS UINT64) AS UUID
```

```
FUNCTION UUID(high AS UINT64,low AS INTEGER) AS UUID
```

```
FUNCTION UUID(high AS UINT64,low AS UINT64) AS UUID
```

```
FUNCTION UUID(text AS STRING) AS UUID FAILS UuidError
```

```
FUNCTION UUIDBytes(value AS UUID) AS LIST OF BYTE
```

```
FUNCTION UUIDClockSequence(value AS UUID) AS INTEGER FAILS UuidError
```

```
FUNCTION UUIDFromBytes(bytes AS LIST OF BYTE) AS UUID FAILS UuidError
```

```
FUNCTION UUIDFromGuidBytes(bytes AS LIST OF BYTE) AS UUID FAILS UuidError
```

```
FUNCTION UUIDGregorianTicks(value AS UUID) AS UINT64 FAILS UuidError
```

```
FUNCTION UUIDGuidBytes(value AS UUID) AS LIST OF BYTE
```

```
FUNCTION UUIDNamespaceDNS() AS UUID
```

```
FUNCTION UUIDNamespaceOID() AS UUID
```

```
FUNCTION UUIDNamespaceURL() AS UUID
```

```
FUNCTION UUIDNamespaceX500() AS UUID
```

```
FUNCTION UUIDNode(value AS UUID) AS UINT64 FAILS UuidError
```

```
FUNCTION UUIDUnixMilliseconds(value AS UUID) AS INTEGER FAILS UuidError
```

```
FUNCTION UUIDv1() AS UUID FAILS UuidError,IoError
```

```
FUNCTION UUIDv1(ticks AS UINT64,sequence AS INTEGER,node AS UINT64) AS UUID FAILS UuidError
```

```
FUNCTION UUIDv1ToV6(value AS UUID) AS UUID FAILS UuidError
```

```
FUNCTION UUIDv3(namespace AS UUID,name AS STRING) AS UUID
```

```
FUNCTION UUIDv4() AS UUID FAILS IoError
```

```
FUNCTION UUIDv5(namespace AS UUID,name AS STRING) AS UUID
```

```
FUNCTION UUIDv6() AS UUID FAILS UuidError,IoError
```

```
FUNCTION UUIDv6(ticks AS UINT64,sequence AS INTEGER,node AS UINT64) AS UUID FAILS UuidError
```

```
FUNCTION UUIDv6ToV1(value AS UUID) AS UUID FAILS UuidError
```

```
FUNCTION UUIDv7() AS UUID FAILS UuidError,IoError
```

```
FUNCTION UUIDv7(milliseconds AS INTEGER) AS UUID FAILS UuidError,IoError
```

```
FUNCTION UUIDv7(milliseconds AS INTEGER,entropy AS LIST OF BYTE) AS UUID FAILS UuidError
```

```
FUNCTION UUIDv8(high AS UINT64,low AS UINT64) AS UUID
```

```
FUNCTION UUIDVariant(value AS UUID) AS UUIDVariantKind
```

```
FUNCTION UUIDVersion(value AS UUID) AS INTEGER
```

V

```
FUNCTION VAL(text AS STRING) AS INTEGER FAILS NumericError
```

```
FUNCTION VALDOUBLE(text AS STRING) AS DOUBLE FAILS NumericError
```

```
FUNCTION ValidateKey[T](value AS T) AS INTEGER FAILS LIKE __ValueKeyCheck(AS T)
```

```
FUNCTION ValidateLogLayout(pattern AS STRING) AS INTEGER FAILS LoggingError
```

```
FUNCTION ValidateLogMessage(pattern AS STRING,count AS INTEGER) AS INTEGER FAILS
LogFormatError
```

```
FUNCTION ValidFormatSpec(text AS STRING, kind AS INTEGER) AS BOOLEAN
```

```
FUNCTION Values(value AS JsonValue) AS LIST OF JsonValue FAILS JsonError
```

```
FUNCTION Values[K,V](BYREF table AS DICT OF K TO V) AS BORROW LIST OF V FROM table
```

```
FUNCTION Values[K,V](d AS SortedDictionary[K,V]) AS BORROW LIST OF V FROM d WHERE V IS
RESOURCE
```

```
FUNCTION Values[K,V](table AS DICT OF K TO V) AS LIST OF V WHERE V IS COPYABLE
```

```
FUNCTION ValueSchema[T](witness AS OPTIONAL OF T) AS ValueSchema
```

```
FUNCTION ValueSnapshot[K,V](d AS SortedDictionary[K,V]) AS LIST OF V WHERE V IS COPYABLE
```

```
FUNCTION VectorAdd(left AS ARRAY OF BYTE,right AS ARRAY OF BYTE) AS LIST OF BYTE FAILS
VectorDimensionError,VectorRankError,NumericError
```

```
FUNCTION VectorAdd(left AS ARRAY OF DOUBLE,right AS ARRAY OF DOUBLE) AS LIST OF DOUBLE FAILS
VectorDimensionError,VectorRankError
```

```
FUNCTION VectorAdd(left AS ARRAY OF INTEGER,right AS ARRAY OF INTEGER) AS LIST OF INTEGER
FAILS VectorDimensionError,VectorRankError,NumericError
```

```
FUNCTION VectorAdd(left AS ARRAY OF UINT32,right AS ARRAY OF UINT32) AS LIST OF UINT32 FAILS
VectorDimensionError,VectorRankError,NumericError
```

```
FUNCTION VectorAdd(left AS ARRAY OF UINT64,right AS ARRAY OF UINT64) AS LIST OF UINT64 FAILS
VectorDimensionError,VectorRankError,NumericError
```

```
FUNCTION VectorAdd(left AS LIST OF BYTE,right AS LIST OF BYTE) AS LIST OF BYTE FAILS
VectorDimensionError,NumericError
```

```
FUNCTION VectorAdd(left AS LIST OF DOUBLE,right AS LIST OF DOUBLE) AS LIST OF DOUBLE FAILS
VectorDimensionError
```

```
FUNCTION VectorAdd(left AS LIST OF INTEGER,right AS LIST OF INTEGER) AS LIST OF INTEGER FAILS
VectorDimensionError,NumericError
```

```
FUNCTION VectorAdd(left AS LIST OF UINT32,right AS LIST OF UINT32) AS LIST OF UINT32 FAILS
VectorDimensionError,NumericError
```

```
FUNCTION VectorAdd(left AS LIST OF UINT64,right AS LIST OF UINT64) AS LIST OF UINT64 FAILS
VectorDimensionError,NumericError
```

```
FUNCTION VectorDivide(left AS ARRAY OF BYTE,right AS ARRAY OF BYTE) AS LIST OF BYTE FAILS
VectorDimensionError,VectorRankError,NumericError
```

```
FUNCTION VectorDivide(left AS ARRAY OF DOUBLE,right AS ARRAY OF DOUBLE) AS LIST OF DOUBLE
FAILS VectorDimensionError,VectorRankError
```

```
FUNCTION VectorDivide(left AS ARRAY OF INTEGER,right AS ARRAY OF INTEGER) AS LIST OF INTEGER
FAILS VectorDimensionError,VectorRankError,NumericError
```

```
FUNCTION VectorDivide(left AS ARRAY OF UINT32,right AS ARRAY OF UINT32) AS LIST OF UINT32
FAILS VectorDimensionError,VectorRankError,NumericError
```

```
FUNCTION VectorDivide(left AS ARRAY OF UINT64,right AS ARRAY OF UINT64) AS LIST OF UINT64
FAILS VectorDimensionError,VectorRankError,NumericError
```

```
FUNCTION VectorDivide(left AS LIST OF BYTE,right AS LIST OF BYTE) AS LIST OF BYTE FAILS
VectorDimensionError,NumericError
```

```
FUNCTION VectorDivide(left AS LIST OF DOUBLE,right AS LIST OF DOUBLE) AS LIST OF DOUBLE FAILS
VectorDimensionError
```

```
FUNCTION VectorDivide(left AS LIST OF INTEGER,right AS LIST OF INTEGER) AS LIST OF INTEGER
FAILS VectorDimensionError,NumericError
```

```
FUNCTION VectorDivide(left AS LIST OF UINT32,right AS LIST OF UINT32) AS LIST OF UINT32 FAILS
VectorDimensionError,NumericError
```

```
FUNCTION VectorDivide(left AS LIST OF UINT64,right AS LIST OF UINT64) AS LIST OF UINT64 FAILS
VectorDimensionError,NumericError
```

```
FUNCTION VectorMultiply(left AS ARRAY OF BYTE,right AS ARRAY OF BYTE) AS LIST OF BYTE FAILS
VectorDimensionError,VectorRankError,NumericError
```

```
FUNCTION VectorMultiply(left AS ARRAY OF DOUBLE,right AS ARRAY OF DOUBLE) AS LIST OF DOUBLE
FAILS VectorDimensionError,VectorRankError
```

```
FUNCTION VectorMultiply(left AS ARRAY OF INTEGER,right AS ARRAY OF INTEGER) AS LIST OF
INTEGER FAILS VectorDimensionError,VectorRankError,NumericError
```

```
FUNCTION VectorMultiply(left AS ARRAY OF UINT32,right AS ARRAY OF UINT32) AS LIST OF UINT32
FAILS VectorDimensionError,VectorRankError,NumericError
```

```
FUNCTION VectorMultiply(left AS ARRAY OF UINT64,right AS ARRAY OF UINT64) AS LIST OF UINT64
FAILS VectorDimensionError,VectorRankError,NumericError
```

```
FUNCTION VectorMultiply(left AS LIST OF BYTE,right AS LIST OF BYTE) AS LIST OF BYTE FAILS
VectorDimensionError,NumericError
```

```
FUNCTION VectorMultiply(left AS LIST OF DOUBLE,right AS LIST OF DOUBLE) AS LIST OF DOUBLE
FAILS VectorDimensionError
```

```
FUNCTION VectorMultiply(left AS LIST OF INTEGER,right AS LIST OF INTEGER) AS LIST OF INTEGER
FAILS VectorDimensionError,NumericError
```

```
FUNCTION VectorMultiply(left AS LIST OF UINT32,right AS LIST OF UINT32) AS LIST OF UINT32
FAILS VectorDimensionError,NumericError
```

```
FUNCTION VectorMultiply(left AS LIST OF UINT64,right AS LIST OF UINT64) AS LIST OF UINT64
FAILS VectorDimensionError,NumericError
```

```
FUNCTION VectorSubtract(left AS ARRAY OF BYTE,right AS ARRAY OF BYTE) AS LIST OF BYTE FAILS
VectorDimensionError,VectorRankError,NumericError
```

```
FUNCTION VectorSubtract(left AS ARRAY OF DOUBLE,right AS ARRAY OF DOUBLE) AS LIST OF DOUBLE
FAILS VectorDimensionError,VectorRankError
```

```
FUNCTION VectorSubtract(left AS ARRAY OF INTEGER,right AS ARRAY OF INTEGER) AS LIST OF
INTEGER FAILS VectorDimensionError,VectorRankError,NumericError
```

```
FUNCTION VectorSubtract(left AS ARRAY OF UINT32,right AS ARRAY OF UINT32) AS LIST OF UINT32
FAILS VectorDimensionError,VectorRankError,NumericError
```

```
FUNCTION VectorSubtract(left AS ARRAY OF UINT64,right AS ARRAY OF UINT64) AS LIST OF UINT64
FAILS VectorDimensionError,VectorRankError,NumericError
```

```
FUNCTION VectorSubtract(left AS LIST OF BYTE,right AS LIST OF BYTE) AS LIST OF BYTE FAILS
VectorDimensionError,NumericError
```

```
FUNCTION VectorSubtract(left AS LIST OF DOUBLE,right AS LIST OF DOUBLE) AS LIST OF DOUBLE
FAILS VectorDimensionError
```

```
FUNCTION VectorSubtract(left AS LIST OF INTEGER,right AS LIST OF INTEGER) AS LIST OF INTEGER
FAILS VectorDimensionError,NumericError
```

```
FUNCTION VectorSubtract(left AS LIST OF UINT32,right AS LIST OF UINT32) AS LIST OF UINT32
FAILS VectorDimensionError,NumericError
```

```
FUNCTION VectorSubtract(left AS LIST OF UINT64,right AS LIST OF UINT64) AS LIST OF UINT64
FAILS VectorDimensionError,NumericError
```

W

```
FUNCTION WaitForIO(fds AS LIST OF INTEGER,events AS LIST OF INTEGER,deadline AS INTEGER=0) AS
INTEGER FAILS IoError
```

```
FUNCTION WaitProcess(process AS Process) AS INTEGER FAILS IoError
```

```
FUNCTION WarpCanvas(source AS Canvas,transform AS AffineTransform,width AS INTEGER,height AS
INTEGER,sampling AS Sampling=Sampling.Bilinear,space AS
BlendSpace=BlendSpace.EncodedSRGB,maxPixels AS INTEGER=16777216) AS Canvas FAILS
CanvasError,IndexError,NumericError
```

```
FUNCTION WeekdayNumber(value AS Weekday) AS INTEGER
```

```
FUNCTION WeekdayOf(value AS Date) AS Weekday
```

```
FUNCTION WeekOfYear(value AS Date,rules AS WeekRules) AS WeekDateParts FAILS DateTimeError
```

```
FUNCTION WithOffsetSameInstant(value AS OffsetDateTime,offset AS UtcOffset) AS OffsetDateTime
FAILS DateTimeError
```

```
FUNCTION WithoutURLCredentials(BYVAL value AS URL) AS URL
```

```
FUNCTION WithoutURLFragment(BYVAL value AS URL) AS URL
```

```
FUNCTION WithoutURLPort(BYVAL value AS URL) AS URL
```

```
FUNCTION WithoutURLQuery(BYVAL value AS URL) AS URL
```

```
FUNCTION WithURLCredentials(BYVAL value AS URL,username AS STRING) AS URL FAILS UrlError
```

```
FUNCTION WithURLCredentials(BYVAL value AS URL,username AS STRING,password AS STRING) AS URL
FAILS UrlError
```

```
FUNCTION WithURLFragment(BYVAL value AS URL,fragment AS STRING) AS URL
```

```
FUNCTION WithURLHost(BYVAL value AS URL,host AS STRING,port AS INTEGER=-1) AS URL FAILS
UrlError
```

```
FUNCTION WithURLPath(BYVAL value AS URL,path AS STRING) AS URL FAILS UrlError
```

```
FUNCTION WithURLPort(BYVAL value AS URL,port AS INTEGER) AS URL FAILS UrlError
```

```
FUNCTION WithURLQuery(BYVAL value AS URL,parameters AS DICT OF STRING TO LIST OF STRING) AS
URL FAILS UrlError
```

```
FUNCTION WithURLQuery(BYVAL value AS URL,parameters AS DICT OF STRING TO STRING) AS URL FAILS
UrlError
```

```
FUNCTION WithURLQuery(BYVAL value AS URL,parameters AS LIST OF URLParameter,form AS
BOOLEAN=TRUE) AS URL FAILS UrlError
```

```
FUNCTION WithURLScheme(BYVAL value AS URL,scheme AS STRING) AS URL FAILS UrlError
```

```
FUNCTION WithZoneSameInstant(value AS ZonedDateTime,zone AS TimeZone) AS ZonedDateTime FAILS
DateTimeError
```

```
FUNCTION WrappingAdd(left AS UINT64, right AS UINT64) AS UINT64
```

```
FUNCTION WrappingMultiply(left AS UINT64, right AS UINT64) AS UINT64
```

```
FUNCTION WrappingSubtract(left AS UINT64, right AS UINT64) AS UINT64
```

```
FUNCTION Write[T](BYREF writer AS BufferedWriter[T],text AS STRING) AS INTEGER FAILS IoError
WHERE T IS RESOURCE
```

```
FUNCTION WriteAll(bytes AS LIST OF BYTE) AS INTEGER FAILS IoError
```

```
FUNCTION WriteAll(bytes AS LIST OF INTEGER) AS INTEGER FAILS IoError
```

```
FUNCTION WriteAll(text AS STRING) AS INTEGER FAILS IoError
```

```
FUNCTION WriteAt(BYREF buffer AS ByteBuffer,offset AS INTEGER,bytes AS LIST OF BYTE) AS
INTEGER FAILS IndexError
```

```
FUNCTION WriteByte(BYREF writer AS ByteWriter,value AS BYTE) AS INTEGER
```

```
FUNCTION WriteByte(value AS INTEGER) AS INTEGER FAILS IoError
```

```
FUNCTION WriteBytes(BYREF writer AS ByteWriter,bytes AS LIST OF BYTE) AS INTEGER FAILS
IndexError
```

```
FUNCTION WriteBytes[T](BYREF writer AS BufferedWriter[T],bytes AS LIST OF BYTE) AS INTEGER
FAILS IoError,IndexError WHERE T IS RESOURCE
```

```
FUNCTION WriteDouble(BYREF writer AS ByteWriter,value AS DOUBLE,Order AS
ByteOrder=ByteOrder.LittleEndian) AS INTEGER
```

```
FUNCTION WriteFile(file AS File,bytes AS LIST OF BYTE) AS INTEGER FAILS IoError
```

```
FUNCTION WriteFile(file AS File,text AS STRING) AS INTEGER FAILS IoError
```

```
FUNCTION WriteFileAtomic(path AS STRING,bytes AS LIST OF BYTE) AS INTEGER FAILS IoError
```

```
FUNCTION WriteFileAtomic(path AS STRING,text AS STRING,expected AS OPTIONAL OF
FileInfo=NONE,expectAbsent AS BOOLEAN=FALSE) AS INTEGER FAILS IoError,FileChanged
```

```
FUNCTION WriteHTTPBody(connection AS TCPConnection,BYREF exchange AS HTTPExchange,bytes AS
LIST OF BYTE) AS INTEGER FAILS IoError,HTTPError,StringError
```

```
FUNCTION WriteHTTPBody(connection AS TCPConnection,BYREF exchange AS HTTPExchange,text AS
STRING) AS INTEGER FAILS IoError,HTTPError,StringError
```

```
FUNCTION WriteHTTPResponse(connection AS TCPConnection,BYREF exchange AS
HTTPExchange,response AS HTTPResponse) AS INTEGER FAILS IoError,HTTPError,StringError
```

```
FUNCTION WriteHTTPResponse(connection AS TCPConnection,BYREF exchange AS HTTPExchange,status
AS INTEGER,headers AS LIST OF HTTPHeader,body AS LIST OF BYTE) AS INTEGER FAILS
IoError,HTTPError,StringError
```

```
FUNCTION WriteHTTPResponse(connection AS TCPConnection,BYREF exchange AS HTTPExchange,status
AS INTEGER,text AS STRING,contentType AS STRING="text/plain; charset=utf-8") AS INTEGER FAILS
IoError,HTTPError,StringError
```

```
FUNCTION WriteInt16(BYREF writer AS ByteWriter,value AS INTEGER,Order AS
ByteOrder=ByteOrder.LittleEndian) AS INTEGER FAILS NumericError
```

```
FUNCTION WriteInt32(BYREF writer AS ByteWriter,value AS INTEGER,Order AS
ByteOrder=ByteOrder.LittleEndian) AS INTEGER FAILS NumericError
```

```
FUNCTION WriteInt64(BYREF writer AS ByteWriter,value AS INTEGER,Order AS
ByteOrder=ByteOrder.LittleEndian) AS INTEGER
```

```
FUNCTION WriteInt8(BYREF writer AS ByteWriter,value AS INTEGER) AS INTEGER FAILS NumericError
```

```
FUNCTION WriteJSON[T](file AS File,value AS T) AS INTEGER FAILS IoError,JsonError,LIKE
__JsonEncodeErrors(AS T)
```

```
FUNCTION WriteJSON[T](file AS File,value AS T,codec AS JsonCodec) AS INTEGER FAILS
IoError,JsonError,LIKE codec, LIKE __JsonEncodeErrors(AS T)
```

```
FUNCTION WriteJSONFile[T](path AS STRING,value AS T) AS INTEGER FAILS IoError,JsonError,LIKE
__JsonEncodeErrors(AS T)
```

```
FUNCTION WriteJSONFile[T](path AS STRING,value AS T,codec AS JsonCodec) AS INTEGER FAILS
IoError,JsonError,LIKE codec, LIKE __JsonEncodeErrors(AS T)
```

```
FUNCTION WriteJSONLine[T](file AS File,value AS T) AS INTEGER FAILS IoError,JsonError,LIKE
__JsonEncodeErrors(AS T)
```

```
FUNCTION WriteJSONLine[T](file AS File,value AS T,codec AS JsonCodec) AS INTEGER FAILS
IoError,JsonError,LIKE codec, LIKE __JsonEncodeErrors(AS T)
```

```
FUNCTION WriteLine[T](BYREF writer AS BufferedWriter[T],text AS STRING="") AS INTEGER FAILS
IoError WHERE T IS RESOURCE
```

```
SUB WriteLog(logger AS Logger,level AS LogLevel,message AS STRING,values AS LIST OF
LogFieldValue) FAILS LoggingError
```

```
SUB WriteLog(logger AS LoggerHandle,level AS LogLevel,message AS STRING,values AS LIST OF
LogFieldValue) FAILS LoggingError
```

```
FUNCTION WriteProcess(process AS Process,text AS STRING) AS INTEGER FAILS IoError
```

```
FUNCTION WritePseudoTerminal(terminal AS PseudoTerminal,text AS STRING) AS INTEGER FAILS
IoError
```

```
FUNCTION WriteTCP(connection AS TCPConnection,bytes AS LIST OF BYTE) AS INTEGER FAILS IoError
```

```
FUNCTION WriteTCP(connection AS TCPConnection,bytes AS LIST OF BYTE,timeout AS Duration) AS
INTEGER FAILS IoError
```

```
FUNCTION WriteTCP(connection AS TCPConnection,text AS STRING) AS INTEGER FAILS IoError
```

```
FUNCTION WriteTCP(connection AS TCPConnection,text AS STRING,timeout AS Duration) AS INTEGER
FAILS IoError
```

```
FUNCTION WriteTerminal(terminal AS Terminal,text AS STRING) AS INTEGER FAILS IoError
```

```
FUNCTION WriteUInt16(BYREF writer AS ByteWriter,value AS UINT32,Order AS
ByteOrder=ByteOrder.LittleEndian) AS INTEGER FAILS NumericError
```

```
FUNCTION WriteUInt32(BYREF writer AS ByteWriter,value AS UINT32,Order AS
ByteOrder=ByteOrder.LittleEndian) AS INTEGER
```

```
FUNCTION WriteUInt64(BYREF writer AS ByteWriter,value AS UINT64,Order AS
ByteOrder=ByteOrder.LittleEndian) AS INTEGER
```

Y

```
FUNCTION Year(value AS Date) AS INTEGER
```

Z

```
FUNCTION ZoneDatabaseDigest(value AS ZoneDatabase) AS STRING
```

```
FUNCTION ZoneDatabaseFromEntries(entries AS LIST OF ZoneDataEntry,version AS OPTIONAL OF
STRING) AS ZoneDatabase FAILS DateTimeError
```

```
FUNCTION ZoneDatabaseVersion(value AS ZoneDatabase) AS OPTIONAL OF STRING
```

```
FUNCTION ZonedFromStorageJSON(value AS JsonValue,database AS ZoneDatabase) AS ZonedDateTime
FAILS DateTimeError
```

```
FUNCTION ZonedToStorageJSON(value AS ZonedDateTime) AS JsonValue FAILS DateTimeError
```

```
FUNCTION ZoneId(value AS TimeZone) AS STRING
```

```
FUNCTION ZoneIds(value AS ZoneDatabase) AS LIST OF STRING
```

```
FUNCTION ZoneOf(value AS ZonedDateTime) AS TimeZone
```

```
FUNCTION ZoneRulesIdentity(value AS TimeZone) AS STRING
```