Good — this is exactly the level of structure we need.

You're not asking for "an intro to ZAX."
You're asking for a **Pascal Quick Guide–style Chapter 1**, with the same rhythm, density, and instructional scaffolding.

So here is **ZAX Quick Guide — Chapter 1**, written to mirror the Pascal document's tone and pattern.

ZAX Quick Guide

Chapter 1 — Overview and Toolchain

1.1 What Is ZAX?

ZAX is a **structured assembler for the Z80 family of processors**.

It combines:

- Raw Z80 instruction emission
- Structured control flow (`if`, `while`, `select`)
- Typed storage (`byte`, `word`, arrays, records, unions)
- Compile-time expressions (`const`, `sizeof`, `offsetof`)
- Hygienic inline macro expansion (`op`)
- Deterministic module layout

ZAX does **not** introduce:

- A runtime system
- Hidden prologues or epilogues
- Implicit memory allocation
- Automatic register preservation beyond declared boundaries

ZAX is designed around one principle:

Every source construct must lower predictably to bounded Z80 instructions.

1.2 Why Use ZAX?

Traditional Z80 assembly gives total control but lacks structure.

High-level languages give structure but hide the machine.

ZAX is intended for:

- Systems programming on Z80
- Game engines
- Embedded firmware
- Educational toolchains
- ROM-based systems
- Monitor-driven development
- OS-level development (via `extern` bindings)

ZAX preserves:

- Direct register usage
- Explicit memory control
- 64KB addressing model

While adding:

- Type-aware layout
- Structured syntax
- Compile-time safety checks
- Namespaced enums
- Power-of-two storage invariants

1.3 ZAX Design Principles

ZAX v0.2 is built around these invariants:

1. Predictable Lowering

Each source-level expression must have bounded code generation.

2. No Hidden Multiplication

Composite types use **power-of-two storage sizes** to ensure index scaling uses shifts only.

3. Single Runtime Atom Rule

An address expression may contain at most **one runtime-varying component**.

4. Assembler-First Semantics

Registers are not abstracted away. You write real Z80 instructions.

5. Explicit Boundaries

Function calls follow preservation-safe contracts.

1.4 ZAX Toolchain

ZAX is a command-line compiler.

A source file uses:

filename.zax

To compile:

zax filename.zax

Output Files

Extension	Description
`.hex`	Intel HEX (64KB address space)
`.bin`	Raw binary
`.d8dbg.json`	Debug metadata

ZAX targets a **single 64KB memory space**.

Extended HEX record types (02/04) are intentionally unsupported.
Bank-switching and segmented models are outside the core language.

1.5 Development Workflow

The ZAX workflow mirrors traditional assembler development:

1. Write source file (`.zax`)
2. Compile with `zax`
3. Load output into:
    - Emulator
    - Monitor
    - ROM image builder
    - Hardware loader

There is no dynamic linker.

Modules are resolved deterministically at compile time.

1.6 External Bindings

ZAX supports importing external symbols from operating systems or firmware:

extern func os_print_char(a: byte): void

External declarations:

- Declare a boundary
- Do not generate code
- Must match the external ABI

This allows integration with:

- CP/M-style BIOS routines
- Monitor ROM calls
- Custom OS kernels
- Firmware interrupt services

1.7 Minimal ZAX Program

Create `main.zax`:

```
export func main(): void
  ld a, 'A'
  ret
end
```

Explanation

Element	Meaning
`export`	Makes symbol visible to linker
`func`	Declares a function
`(): void`	No parameters, no return value
`ld a, 'A'`	Raw Z80 instruction
`ret`	Explicit return
`end`	Terminates function

There is:

- No implicit stack frame
- No automatic register save
- No hidden return

1.8 When Compiled and Executed…

If `main` is called by a monitor:

- Register A will contain ASCII `'A'`
- Control returns to caller

The emitted machine code corresponds directly to:

```
3E 41   ; LD A,'A'
C9      ; RET
```

(Exact encoding may vary depending on assembler output mode.)

1.9 Important Points to Note

- ZAX instructions are real Z80 instructions.
- Composite types are padded to power-of-two storage sizes.
- `sizeof` reflects padded storage size.
- Address expressions may contain only one runtime-varying component.
- Function overloading is not supported.
- Signed storage types are not provided.
- Enums must be namespace-qualified.
- Ops expand inline and do not create stack frames.

1.10 What Comes Next

The next chapter explains the **ZAX Storage Model**:

- Scalar widths
- Composite layout
- Power-of-two padding
- `sizeof`
- `offsetof`
- Storage-visible invariants

Understanding storage is required before arrays and indexing.

Excellent — now we enter the structural core of the language.

Following the Pascal Quick Guide rhythm:

- Concept framing
- Formal syntax
- Reference tables
- Runnable examples
- “When compiled and executed…” output blocks

ZAX Quick Guide

Chapter 2 — The Storage Model

2.1 Introduction

Before arrays, indexing, or structured control flow, you must understand **how ZAX stores data**.

ZAX is a **width-based, storage-visible language**.

There is:

- No hidden packing model
- No dual “logical vs physical size”
- No signed vs unsigned storage distinction

All layout decisions are mechanical and predictable.

2.2 Scalar Types

ZAX defines three fundamental scalar storage widths:

Type	Size (bytes)	Description
`byte`	1	8-bit value
`word`	2	16-bit value
`ptr`	2	16-bit address

There are **no signed storage types**.

Signedness is interpreted through flags (`CP`, condition codes), not through type.

Example: Scalar Declarations

globals

```
  counter: byte
  total: word
  buffer_ptr: ptr
end
```

Each variable occupies fixed-width storage in RAM.

2.3 Composite Types

ZAX supports:

- Arrays
- Records
- Unions

However, v0.2 introduces a key rule:

All composite storage sizes are rounded to the next power of two.

This is not optional.

2.4 Power-of-Two Storage Rule

Natural Size vs Storage Size

Consider:

```
type Sprite
  x: byte
  y: byte
  tile: byte
  flags: word
end
```

Natural size calculation:

Field	Size

x	1
y	1
tile	1
flags	2
Total	5 bytes

ZAX storage size:

Rounded up to next power of two → 8 bytes

Compiler Warning

When compiled:

Warning: Type Sprite size 5 padded to 8.
  Storage uses 8 bytes per element (3 bytes padding).

Padding becomes part of the actual layout.

2.5 Why Power-of-Two?

ZAX must avoid hidden multiplication.

Runtime indexing scales element size using shifts:

Element Size	Scaling Sequence
1	none
2	`ADD HL,HL`
4	`ADD HL,HL` ×2
8	`ADD HL,HL` ×3
16	`ADD HL,HL` ×4

This guarantees:

- Small, bounded code
- No multiplication routines
- No hidden helper calls

2.6 Explicit Padding

To suppress padding warnings:

```
type Sprite
  x: byte
  y: byte
  tile: byte
  flags: word
  _pad: byte[3]
end
```

Now natural size = 8 bytes.

No warning is issued.

2.7 Arrays

General Form

```
type
  Name = ElementType[Count]
```

or in globals:

```
globals
  arr: byte[16]
end
```

Example

```
globals
```

```
  table: Sprite[4]
end
```

Each element occupies 8 bytes (not 5).

Total storage:

4 × 8 = 32 bytes

2.8 `sizeof`

`sizeof(Type)` returns the storage size.

Example

```
const
  sprite_size = sizeof(Sprite)
end
```

`sprite_size` = 8

Compile-Time Example

```
export func main(): void
  ld hl, sizeof(Sprite)
  ret
end
```

When Compiled and Executed…

`HL` will contain:

0008h

2.9 Records

Records are composite types with named fields.

General Form

```
type Name
  field1: type
  field2: type
end
```

Fields are laid out in declaration order.

Offsets are computed using storage size of previous fields.

Example

```
type Point
  x: byte
  y: byte
end
```

Natural size = 2
 Storage size = 2 (already power-of-two)

2.10 Unions

Unions overlay fields at offset 0.

General Form

```
type Name union
  field1: type
  field2: type
end
```

Storage size = maximum field size (rounded to next power-of-two if needed).

2.11 `offsetof`

Returns byte offset of a field within a record.

Example

```
const
  offset_flags = offsetof(Sprite, flags)
end
```

Given layout:

Field	Offset
x	0
y	1
tile	2
flags	3

`offset_flags` = 3

Padding after `flags` ensures total size = 8.

2.12 Nested Layout

For nested types:

```
type Scene
  sprites: Sprite[4]
end
```

Each `Sprite` = 8 bytes
`sprite[0]` at offset 0
`sprite[1]` at offset 8
`sprite[2]` at offset 16
`sprite[3]` at offset 24

Total Scene storage = 32 bytes

2.13 Important Points to Note

- Composite storage sizes are always powers of two.
- Padding is storage-visible.
- `sizeof` returns padded size.
- `offsetof` returns byte offsets.
- There is no packed-layout mode.
- Signedness is not encoded in storage.
- Layout is deterministic and mechanical.

2.14 Summary

ZAX storage is:

- Width-based
- Deterministic
- Power-of-two aligned
- Shift-scalable
- Visible in `sizeof` and indexing

This storage model exists to guarantee bounded lowering for address computation.

Excellent.
Now we enter the chapter that defines ZAX's identity.

Following the Pascal Quick Guide pattern:

Concept framing

Formal syntax

Reference tables

Clear examples

"When compiled and executed…" demonstrations

ZAX Quick Guide

Chapter 3 — Addressing and Indexing

3.1 Introduction

ZAX is an assembler-first language.

Addressing must remain:

Predictable

Bounded

Transparent

In v0.2, addressing is governed by two invariants:

1. Power-of-two storage scaling

2. Single runtime-atom rule

This chapter explains:

Array indexing forms

Register indexing

Indirect forms

Nested indexing

The runtime-atom limit

3.2 General Index Syntax

The general form of indexed access is:

array[index]

Where index may be:

A compile-time constant

An 8-bit register

A 16-bit register

A typed scalar variable

An indirect Z80 form (HL), (IX+d), etc.

3.3 Index Forms Reference Table

Form	Meaning	Runtime Atom Count
arr[5]	Constant index	0
arr[A]	8-bit register index	1
arr[HL]	16-bit register index	1
arr[idx]	Typed scalar variable	1
arr[(HL)]	Byte at address HL used as index	1
arr[(IX+3)]	Byte at IX+3 used as index	1
arr[CONST + 3]	Constant expression	0

3.4 The v0.2 Semantic Change

In v0.1:

arr[HL]

meant:

> Indirect byte at (HL)

In v0.2:

arr[HL]

means:

> 16-bit direct index value in HL

Indirect byte-at-HL is written as:

arr[(HL)]

This removes ambiguity between grouping parentheses and Z80 indirect syntax.

3.5 8-bit Register Indexing

Example:

```
globals
  table: byte[256]
end

export func main(): void
  ld a, table[B]
  ret
end
```

Here:

B provides 8-bit index

No scaling required (element size = 1)

When Compiled…

Lowering is equivalent to:

```
; base address of table
; B used as offset
```

No multiplication is generated.

3.6 16-bit Register Indexing

Example:

```
globals
  table: word[512]
end

export func main(): void
  ld a, table[HL]
  ret
end
```

Here:

Element size = 2

HL contains 16-bit index

Scaling requires one shift

Lowering sequence:

ADD HL,HL
ADD HL, base_address

Bounded cost.

3.7 Typed Variable Indexing

Example:

```
globals
  table: byte[128]
end

export func main(): void
  var
    idx: byte
  end

  ld a, table[idx]
  ret
end
```

Lowering rule:

Byte variable zero-extended into HL

Address computed

HL preserved with push/pop if required

Typed variables behave as values.

There is no implicit dereference model.

3.8 Indirect Index Forms

Z80-style indirect patterns are allowed:

arr[(HL)]
arr[(IX+5)]
arr[(IY-2)]

These forms:

Treat the byte at the effective address as the index

Count as one runtime atom

3.9 Runtime Atom Rule

An address expression may contain at most one runtime-varying source.

A runtime atom is:

A register

A typed variable

An indirect form

Constants do not count.

Allowed Examples

Expression	Atom Count	Allowed
arr[5]	0	Yes
arr[idx]	1	Yes
arr[CONST+4][idx]	1	Yes
arr[idx].field	1	Yes

Rejected Examples

Expression	Atom Count	Result
arr[i][j]	2	Compile error
arr[i + j]	2	Compile error
grid[row][col]	2	Compile error

Diagnostic Example

ld a, grid[row][col]

Compiler emits:

Error: Address expression contains more than one runtime atom.

3.10 Staged Addressing Pattern

To resolve multi-atom access:

```
; Instead of:
ld a, grid[row][col]

; Do:
ld hl, grid[row]
; compute second offset explicitly
```

This keeps lowering explicit and bounded.

3.11 Nested Arrays and Storage Scaling

Consider:

```
type Sprite
  x: byte
  y: byte
  tile: byte
  flags: word
end

globals
  grid: Sprite[4][6]
end
```

Sprite size = 8 (padded)

Row size:

6 × 8 = 48 → rounded to 64

Each row occupies 64 bytes.

Total grid size:

4 × 64 = 256 bytes

Padding compounds by design.

3.12 Important Points to Note

arr[HL] is 16-bit index in v0.2.

arr[(HL)] is indirect form.

Only one runtime atom per address expression.

Composite types scale by shifts only.

Padding is storage-visible.

Index scaling never generates multiplication.

3.13 Summary

ZAX addressing is:

Shift-based

Bounded

Single-moving-part

Assembler-transparent

The addressing model ensures no hidden code growth.

What Comes Next

Chapter 4 introduces:

Constants

Compile-time expressions

imm

Expression folding

How atom counting interacts with arithmetic expressions

This bridges storage and control flow.

Good — now we follow the Pascal rhythm: after structure and mechanics comes **constants and expressions**, with formal syntax, tables, and runnable demonstrations.

ZAX Quick Guide

Chapter 4 — Constants and Compile-Time Expressions

4.1 Introduction

ZAX distinguishes between:

- **Runtime values** (registers, variables)
- **Compile-time values** (constants and folded expressions)

Compile-time expressions:

- Do not generate code
- Must be fully resolvable by the compiler
- Participate in layout and addressing

This chapter defines:

- `const`
- Compile-time arithmetic
- `sizeof`
- `offsetof`
- Atom interaction rules

4.2 Constant Declarations

General Form

```
const
  name = expression
end
```

The expression must be fully computable at compile time.

Example

```
const
  screen_width  = 32
  screen_height = 24
  screen_size   = screen_width * screen_height
end
```

All three are compile-time constants.

When Compiled…

No code is emitted.
Values are substituted directly into instruction operands.

4.3 Compile-Time Expression Rules

A compile-time expression may include:

- Integer literals
- Previously defined constants
- Arithmetic operators
- `sizeof(Type)`
- `offsetof(Type, field)`
- Parentheses for grouping

It may NOT include:

- Registers
- Variables
- Indirect forms
- Function calls

4.4 Arithmetic Operators (Compile-Time)

Arithmetic Operators Table

Operator	Meaning
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Integer division
`%`	Remainder
`<<`	Shift left
`>>`	Shift right
`&`	Bitwise AND
`` ` ``	`` ` ``
`^`	Bitwise XOR

All operations use integer arithmetic.

Example

```
const
  mask  = (1 << 3)
  value = (5 + 2) * 4
end


mask = 8
 value = 28
```

4.5 `sizeof`

Returns storage size of a type (power-of-two rounded).

Example

```
type Point
  x: byte
  y: byte
end

const
  point_size = sizeof(Point)
end


point_size = 2
```

Example with Padding

```
type Sprite
  x: byte
  y: byte
  tile: byte
  flags: word
end

const
  sprite_size = sizeof(Sprite)
end


Natural size = 5
 Storage size = 8

sprite_size = 8
```

4.6 `offsetof`

Returns byte offset of a field within a record.

General Form

```
offsetof(Type, field_path)
```

Example

```
const
  offset_tile = offsetof(Sprite, tile)
  offset_flags = offsetof(Sprite, flags)
end
```

Layout:

Field	Offset
x	0
y	1
tile	2
flags	3

Offsets reflect natural layout order.
 Total size still rounded to 8.

4.7 Nested `offsetof`

Allowed when indices are constant:

```
type Scene
  sprites: Sprite[4]
end

const
  second_sprite_offset = offsetof(Scene, sprites[1])
end
```

`second_sprite_offset` = 8

(Each Sprite occupies 8 bytes.)

4.8 Atom Interaction with Expressions

Compile-time arithmetic does **not** contribute runtime atoms.

Example:

ld a, table[CONST1 + CONST2 * 4]

Atom count = 0
 Fully resolved at compile time.

Rejected Example

ld a, table[i + j]

Atom count = 2
 Compile error.

Diagnostic:

Error: Address expression contains more than one runtime atom.

4.9 Grouping Parentheses

Parentheses inside `[ ]` may mean:

1. Z80 indirect form `(HL)`
2. Expression grouping `(3+5)`

If parentheses contain only constants:

arr[(3+5)]

Compiler may emit a warning:

Warning: Redundant grouping parentheses.

4.10 Compile-Time vs Runtime Example

```
const
  index = 5
end

globals
  table: byte[16]
end

export func main(): void
  ld a, table[index]
  ret
end
```

Here:

- `index` is compile-time constant
- Address is computed at compile time
- No runtime scaling needed

When Compiled and Executed…

Instruction will directly reference:

table + 5

No extra instructions emitted.

4.11 Important Points to Note

- `const` values exist only at compile time.
- `sizeof` returns storage size.
- `offsetof` returns byte offset.
- Compile-time expressions generate no runtime code.
- Runtime atoms only apply to dynamic sources.
- Constant arithmetic is unlimited in depth.

4.12 Summary

Compile-time expressions in ZAX:

- Define layout
- Parameterize addressing
- Avoid magic numbers
- Interact safely with the runtime-atom rule

They form the bridge between storage and structured control flow.

What Comes Next

Chapter 5 introduces:

- Structured control flow
- `if`
- `while`
- `repeat`
- `select`
- Flag-based semantics
- Lowering behavior

This is where ZAX moves from layout to execution structure.

If you want to keep following Pascal's rhythm exactly, next chapter should be **Control Flow** before Functions — because that mirrors the traditional learning progression.

Good.
Following the Pascal-style progression, Chapter 5 now introduces **structured control flow** — clear syntax blocks, small runnable examples, and visible output behavior.

ZAX Quick Guide

Chapter 5 — Decision Making and Control Flow

5.1 Introduction

ZAX provides structured control flow constructs while preserving Z80 semantics.

Control structures:

- Do not introduce hidden runtime support
- Lower directly to conditional branches
- Rely on explicit flag state

ZAX does not evaluate high-level boolean expressions automatically.
Branching depends on the processor's condition flags.

5.2 The `if` Statement

General Form

```
if condition
  statements
end
```

Or with alternative branch:

```
if condition
  statements
else
  statements
end
```

The condition maps directly to a flag test.

Example 1 — Simple Branch

```
export func main(): void
  ld a, 5
  cp 5

  if Z
    ld b, 1
```

```
  end

  ret
end
```

Explanation

- `cp 5` sets flags
- `if Z` checks the zero flag
- If equal, `B` becomes 1

When Compiled and Executed…

If A = 5:

- Zero flag set
- B = 1

Otherwise:

- B unchanged

Lowering resembles:

```
CP 5
JR NZ, skip
LD B,1
skip:
```

5.3 Supported Condition Codes

Condition	Meaning
`Z`	Zero flag set
`NZ`	Zero flag clear
`C`	Carry set

`NC`	Carry clear
`P`	Sign positive
`M`	Sign negative
`PE`	Parity even
`PO`	Parity odd

Conditions are passed directly to branch instructions.

5.4 The `while` Loop

General Form

```
while condition
  statements
end
```

The loop continues while the condition is true.

Example — Countdown Loop

```
export func main(): void
  ld a, 5

while NZ
  dec a
  cp 0
end

ret
end
```

Explanation

- Loop continues while Zero flag is not set
- Each iteration decrements A

- When A reaches 0, loop exits

Lowering Pattern

```
loop_start:
DEC A
CP 0
JR Z, loop_end
JR loop_start
loop_end:
```

5.5 The `repeat` Loop

General Form

```
repeat
  statements
until condition
```

Unlike `while`, `repeat` executes body first.

Example

```
export func main(): void
  ld a, 0

repeat
  inc a
  cp 5
until Z

ret
end
```

Explanation

- A increments from 0

- Loop exits when A equals 5

5.6 The `select` Statement

Provides structured multi-branch behavior.

General Form

```
select expression
  case value:
    statements
  case value:
    statements
  default:
    statements
end
```

The expression must resolve to a value suitable for comparison.

Example

```
export func main(): void
  ld a, 2

  select a
    case 1:
      ld b, 10
    case 2:
      ld b, 20
    default:
      ld b, 0
  end

  ret
end
```

When Compiled and Executed…

If A = 2:

- B = 20

Lowering resembles sequential comparisons with bounded branching.

5.7 `break` and `continue`

Within loops:

- `break` exits the loop immediately
- `continue` jumps to next iteration

Example

```
export func main(): void
  ld a, 5

while NZ
  dec a
  cp 3

  if Z
    break
  end

end

ret
end
```

Loop exits when A becomes 3.

5.8 Important Points to Note

- Control flow lowers directly to conditional jumps.
- No hidden evaluation of complex boolean expressions.

- Flags must be prepared explicitly before conditions.
- Nested control flow is supported.
- Lowering remains bounded and predictable.

5.9 Summary

ZAX control flow:

- Adds structure without abstraction overhead
- Uses direct flag semantics
- Produces transparent branch sequences
- Avoids hidden runtime behavior

What Comes Next

Chapter 6 introduces:

- Functions in detail
- Parameters
- Return values
- Call boundary contracts
- Preservation-safe rules

This formalizes modular program structure.

If you'd like, we can now proceed to Chapter 6 — or tighten this chapter with more Pascal-style repetition and output blocks.

Good — now we follow the Pascal progression precisely.

After control flow, Pascal introduces **procedures and functions**.
In ZAX, this is where modular structure and call boundaries become formal.

We will follow the same rhythm:

- Concept framing
- General syntax
- Parameter forms
- Return behavior
- Runnable examples

- “When compiled and executed…”
- Important notes

ZAX Quick Guide

Chapter 6 — Functions and Modular Structure

6.1 Introduction

Functions in ZAX provide:

- Code reuse
- Logical structure
- Explicit call boundaries

ZAX functions:

- Do not inject hidden stack frames
- Do not automatically preserve registers
- Follow explicit boundary contracts

There is no function overloading.

Each function name must be unique.

6.2 Function Declaration

General Form

```
func name(parameter_list): return_type
  statements
end
```

Or exported:

```
export func name(parameter_list): return_type
```

```
  statements
end
```

Return Types

Return Type	Meaning
`void`	No return value
`byte`	Returns value in HL (low byte significant)
`word`	Returns value in HL
`ptr`	Returns 16-bit address in HL

All non-void returns use **HL** as the return channel.

6.3 Example — Void Function

```
func set_a_to_ten(): void
  ld a, 10
  ret
end
```

Calling:

```
export func main(): void
  call set_a_to_ten
  ret
end
```

When Compiled and Executed…

- A = 10 after call
- Control returns normally

No hidden register save occurs.

6.4 Example — Function Returning a Value

```
func add_two(): byte
  ld hl, 2
  ret
end
```

Calling:

```
export func main(): void
  call add_two
  ; result now in HL
  ret
end
```

When Executed…

- HL = 0002h

Return value always appears in HL.

6.5 Parameters

General Form

```
func name(param1: type, param2: type): return_type
```

Parameters are passed using the calling convention defined by ZAX.

Parameter handling:

- Is explicit
- May use registers or stack depending on width
- Does not generate hidden heap structures

Example — Function with Parameters

```
func add(a: byte, b: byte): byte
```

```
  ld l, a
  ld h, 0
  add a, b
  ld l, a
  ret
end
```

Caller:

```
export func main(): void
  ld a, 3
  ld b, 4
  call add
  ret
end
```

When Compiled and Executed…

- HL = 0007h

Exact lowering depends on calling convention rules, but:

- No hidden dynamic allocation occurs
- Return channel remains HL

6.6 Local Variables

Functions may define local variables:

```
func example(): void
  var
    temp: byte
  end

  ld temp, 5
  ret
end
```

Locals:

- Have fixed storage duration within call
- Do not imply heap allocation
- Are part of explicit function boundary

6.7 Preservation-Safe Boundaries (v0.2)

ZAX v0.2 defines:

For `void` functions

- No boundary-visible clobbers

For non-void functions

- Only HL is exposed as return channel

All other register effects must be documented or managed explicitly.

6.8 External Functions

Functions may be declared external:

extern func bios_print_char(a: byte): void

This declares a boundary but generates no code.

Calling such functions assumes:

- Correct ABI
- Correct register expectations

6.9 No Function Overloading

ZAX does not support:

func add(byte, byte)
func add(word, word)

Instead, use explicit names:

add_u8
add_u16

This avoids ambiguity in low-level register-heavy contexts.

6.10 Important Points to Note

- Functions are assembler-transparent.
- Return values always use HL.
- No implicit stack frames are created.
- No function overloading exists.
- Parameters are explicit.
- Boundary contracts are preservation-safe.

6.11 Summary

Functions in ZAX provide:

- Modular structure
- Predictable boundaries
- Explicit data flow
- No hidden runtime cost

They align with assembler-first design while improving organization.

What Comes Next

Chapter 7 introduces:

- The `op` system
- Inline expansion
- Pattern matchers (`r8`, `r16`, `idx16`, `cc`)
- Hygiene rules

- Overload resolution

This is where ZAX moves beyond traditional assembler macros.

If you want to continue the Pascal rhythm strictly, next would indeed be op (analogous to advanced language features), before moving to enums or more advanced type behavior.

Good.
We now reach the chapter that most clearly distinguishes ZAX from traditional assemblers.

In Pascal, this is where advanced language features appear.
In ZAX, this is the op system — structured, hygienic, inline expansion.

As before:

Concept framing

General syntax

Matcher table

Runnable examples

Lowering explanation

Important notes

ZAX Quick Guide

Chapter 7 — The op System (Structured Inline Expansion)

7.1 Introduction

The op system provides structured, type-aware inline expansion.

An op:

Expands at compile time

Generates no call overhead

Does not create stack frames

Can be overloaded by matcher type

Is hygienic (labels are isolated per expansion)

Unlike functions:

Ops do not create call boundaries

Ops do not preserve registers

Ops cannot declare var blocks

They are closer to assembler macros — but type-safe and structured.

7.2 General Form

```
op name(param1: matcher, param2: matcher)
  instructions
end
```

An op body is inserted directly at the call site.

7.3 Matcher Types

Ops match arguments by structural type.

Matcher Reference Table

Matcher Matches

r8	8-bit register (A, B, C, D, E, H, L)
r16	16-bit register pair (HL, DE, BC)
idx16	(IX+d), (IY+d), (IX), (IY)
cc	Condition code (Z, NZ, C, NC, etc.)
label	Label identifier
literal	Constant values

7.4 Simple Op Example

```
op clear(r: r8)
  xor r
end
```

Usage:

```
clear A
clear B
```

Expands to:

```
xor A
xor B
```

No call instruction emitted.

7.5 Condition Code Matcher

```
op jump_if(cond: cc, target: label)
  jp cond, target
end
```

Usage:

```
jump_if Z, done
jump_if NC, loop
```

Expands directly to:

```
jp Z, done
jp NC, loop
```

No additional logic inserted.

7.6 Indexed Matcher (idx16)

```
op load_indexed(dst: r8, src: idx16)
  ld dst, src
end
```

Usage:

```
load_indexed A, (IX+3)
load_indexed B, (IY-2)
```

Expands directly to:

```
ld A, (IX+3)
ld B, (IY-2)
```

The entire indexed pattern is captured as one matcher.

7.7 Overloading by Matcher

Ops may be overloaded based on matcher types.

```
op load(dst: r8, src: r8)
  ld dst, src
end

op load(dst: r8, src: idx16)
  ld dst, src
end
```

The compiler selects the most specific match.

7.8 Hygiene and Local Labels

Ops may contain labels.

Each expansion receives a unique internal label mapping.

Example:

```
op wait_loop(r: r8)
start:
  dec r
  jr NZ, start
end
```

Multiple invocations:

```
wait_loop B
wait_loop C
```

The compiler mangles labels so expansions do not collide.

7.9 What Ops Cannot Do

Ops cannot:

Declare var blocks

Allocate stack frames

Define nested functions

Introduce multi-atom address expressions beyond language rules

If you need locals or structured boundaries, use a function.

7.10 Example — Structured Utility Op

```
op inc_twice(r: r8)
  inc r
  inc r
end
```

Usage:

```
export func main(): void
  ld a, 5
  inc_twice A
  ret
end
```

When Compiled and Executed…

Expands to:

```
ld a, 5
inc a
inc a
ret
```

A = 7

No call overhead.

7.11 Important Points to Note

Ops expand inline.

No call instruction is generated.

Ops may be overloaded.

Ops support condition-code matching.

Ops support indexed addressing matchers.

Ops are hygienic.

Ops cannot allocate locals.

Ops do not create call boundaries.

7.12 Summary

The op system:

Replaces traditional assembler macros

Provides structured, typed inline expansion

Maintains assembler-first predictability

Prevents hidden runtime costs

What Comes Next

Chapter 8 introduces:

Enums

Namespace qualification

Structured symbolic constants

Interaction with select

This strengthens symbolic clarity in larger programs.

If you'd like, we can continue directly into Chapter 8.

Good — now we move into symbolic structure and namespacing.

In Pascal, this is where enumerations and structured types reinforce readability.
In ZAX v0.2, enums are also important because of the qualified-only rule.

We will follow the same pattern:

Concept framing

General syntax

Reference table

Runnable examples

“When compiled and executed…”

Important notes

ZAX Quick Guide

Chapter 8 — Enumerations and Namespaces

8.1 Introduction

Enumerations provide named integer constants grouped under a type.

Enums improve:

Readability

Symbol clarity

Namespace hygiene

Safety in select statements

In ZAX v0.2:

> Enum members must be namespace-qualified.

Unqualified members are compile errors.

8.2 Declaring an Enum

General Form

```
enum TypeName
  Member1
  Member2
  Member3
end
```

Each member is assigned an increasing integer value starting at 0.

Example

```
enum Mode
  Read
  Write
  Append
end
```

Implicit values:

Member Value

Mode.Read 0
Mode.Write 1
Mode.Append 2

8.3 Qualified Access

Enum members must be accessed using:

TypeName.Member

Valid Example

ld a, Mode.Write

Invalid Example

ld a, Write

Compiler error:

Error: Unqualified enum member.

8.4 Enums in select

Enums integrate naturally with select.

Example

```
enum Mode
  Read
  Write
  Append
end

export func main(): void
  ld a, Mode.Write

  select a
    case Mode.Read:
      ld b, 1
    case Mode.Write:
      ld b, 2
    case Mode.Append:
      ld b, 3
  end

  ret
end
```

When Compiled and Executed…

If A = Mode.Write:

B = 2

Lowering is equivalent to numeric comparison:

```
CP 1
JR Z, ...
```

8.5 Explicit Enum Values (Optional)

If supported by the spec:

```
enum ErrorCode
  Ok = 0
  NotFound = 4
  AccessDenied = 7
end
```

If values are not sequential, they follow the assigned constants.

8.6 Enums and Storage

Enums are stored as integers.

Default width: byte unless otherwise specified.

Enums:

Do not create separate storage categories

Do not introduce runtime type information

Exist as symbolic numeric values

8.7 Namespacing Rules

Enums introduce a namespace.

Given:

```
enum Color
  Red
  Green
  Blue
end
```

Accessible symbols:

Color.Red

Color.Green
Color.Blue

The names Red, Green, Blue do not enter the global namespace.

This prevents symbol collisions across modules.

8.8 Important Points to Note

Enum members are compile-time constants.

Members must be namespace-qualified.

Enums improve clarity in select.

Enum values are integers at runtime.

No runtime overhead is introduced.

Enums do not create new storage types.

8.9 Summary

Enums in ZAX provide:

Structured symbolic constants

Safe multi-branch selection

Clean namespace management

Zero runtime cost

They support large programs without polluting the global symbol space.

What Comes Next

Chapter 9 introduces:

Records in detail

Unions

Nested types

Field access semantics

Interaction with addressing rules

This extends storage into structured data design.

If you want to continue, we now move into structured data modeling.

Good.
We now move into structured data modeling — the equivalent of Pascal's records chapter — but grounded in ZAX's storage invariants.

As before:

Concept framing

General syntax

Layout tables

Runnable examples

"When compiled…" blocks

Important notes

ZAX Quick Guide

Chapter 9 — Records, Unions, and Structured Data

9.1 Introduction

Records and unions allow grouping related fields into structured data.

In ZAX:

Layout is deterministic

Fields are stored in declaration order

Composite types obey the power-of-two storage rule

Padding is storage-visible

There is no hidden packing mode.

9.2 Records

A record groups named fields into a single type.

General Form

```
type TypeName
  field1: type
  field2: type
end
```

Fields are placed sequentially in memory.

Example — Point

```
type Point
  x: byte
  y: byte
end
```

Layout

Field	Offset	Size
x	0	1
y	1	1

Natural size = 2
Storage size = 2 (already power-of-two)

9.3 Record with Padding

```
type Sprite
  x: byte
  y: byte
  tile: byte
  flags: word
end
```

Natural Layout

Field	Offset	Size
x	0	1
y	1	1
tile	2	1
flags	3	2
Total		5

Storage Size (v0.2)

Rounded to next power-of-two → 8 bytes

Compiler emits warning unless explicitly padded.

9.4 Field Access

Fields are accessed using dot notation:

variable.field

Example

```
globals
  p: Point
end

export func main(): void
  ld p.x, 10
  ld p.y, 20
  ret
end
```

The compiler resolves p.x to base address + offset.

When Compiled…

Equivalent to:

```
LD (p_base + 0), 10
LD (p_base + 1), 20
```

No runtime computation required for constant offsets.

9.5 Arrays of Records

```
globals
  sprites: Sprite[4]
end
```

Each Sprite occupies 8 bytes.

Layout

Index	Offset
sprites[0]	0
sprites[1]	8
sprites[2]	16
sprites[3]	24

Total size = 32 bytes

9.6 Nested Records

Records may contain other records.

```
type Rectangle
  top_left: Point
  bottom_right: Point
end
```

Layout

Point size = 2
Natural total = 4
Storage size = 4 (already power-of-two)

9.7 Unions

A union overlays multiple fields at offset 0.

General Form

```
type TypeName union
  field1: type
  field2: type
end
```

All fields share the same base address.

Example

```
type Data union
  as_byte: byte
  as_word: word
end
```

Layout

Field	Offset	Size
as_byte	0	1
as_word	0	2

Storage size = 2 (max field size, rounded if necessary)

9.8 offsetof with Records

```
const
  flag_offset = offsetof(Sprite, flags)
end
```

Offset = 3

Padding after field does not affect offset value.

9.9 Field Access with Indexing

ld a, sprites[1].x

Atom count = 0 (constant index)

Address resolved at compile time:

base + (1 × 8) + 0

Rejected Example

ld a, sprites[i].x[j]

Two runtime atoms → compile error.

9.10 Important Points to Note

Record layout is deterministic.

Fields follow declaration order.

Composite types round to power-of-two storage sizes.

Padding is storage-visible.

Unions overlay fields at offset 0.

Field access is compile-time resolved when index is constant.

Runtime-atom rule still applies to indexed access.

9.11 Summary

Records and unions in ZAX provide:

Structured memory layout

Predictable field offsets

Shift-scalable array elements

Zero hidden runtime overhead

They extend the storage model into practical data organization.

What Comes Next

Chapter 10 introduces:

Modules and file structure

Imports

Exports

extern boundaries

Deterministic linking

This completes the program structure model.

If you'd like, we continue into modular structure next.

Good.
Now we complete the structural side of the language: modules, imports, exports, and linkage.

This mirrors the “Units” chapter in Pascal — separation of interface and implementation — but adapted to ZAX’s manifest-driven, deterministic model.

As before:

Concept framing

General forms

Reference tables

Runnable examples

Linking behavior

Important notes

ZAX Quick Guide

Chapter 10 — Modules, Imports, and Linking

10.1 Introduction

ZAX programs may be split across multiple source files.

Each file is a module.

Modules provide:

Symbol separation

Reuse across programs

Controlled public APIs

Deterministic linking

ZAX does not use a runtime loader.
All module resolution occurs at compile time.

10.2 Module Structure

A ZAX module is simply a .zax file.

A module may contain:

const

type

enum

globals

func

op

Symbols are private unless explicitly exported.

10.3 Exporting Symbols

General Form

export func name(...)
export op name(...)
export const name = ...

Only exported symbols are visible to importing modules.

Example — math.zax

```
export func add(a: byte, b: byte): byte
  ld l, a
  ld h, 0
  add a, b
  ld l, a
  ret
end
```

10.4 Importing Modules

General Form

```
import module_name
```

Importing allows access to exported symbols of that module.

Example — main.zax

```
import math

export func main(): void
  ld a, 3
  ld b, 4
  call add
  ret
end
```

10.5 Deterministic Linking

ZAX linking is:

Compile-time only

Order-independent

Deterministic

There is:

No dynamic resolution

No late binding

No runtime relocation

All symbols must be resolvable at compile time.

10.6 Public vs Private Symbols

Declaration	Visibility
func name	Private to module
export func name	Public
const name	Private
export const name	Public
op name	Private
export op name	Public

Unexported symbols are invisible outside the module.

10.7 External Symbols

Modules may declare external symbols:

extern func bios_print_char(a: byte): void

extern:

Declares symbol without definition

Assumes implementation exists elsewhere

Does not emit code

This allows integration with:

Monitor ROM routines

CP/M-style BIOS

Custom firmware

10.8 Module Example — Geometry

geometry.zax

```
export func area_rectangle(w: word, h: word): word
  ; multiplication routine would go here
  ld hl, 0
  ret
end
```

---

main.zax

```
import geometry

export func main(): void
  ld hl, 10
  ld de, 20
  call area_rectangle
  ret
end
```

When Compiled and Linked…

area_rectangle symbol resolved at compile time

Call instruction patched directly

No stub code generated

10.9 Symbol Resolution Rules

Symbols are resolved by name.

If multiple modules export the same symbol, compilation fails.

Imports do not create namespace prefixes.

Enum members remain namespace-qualified.

10.10 No Implicit Initialization

Unlike some high-level languages:

There is no module constructor.

There is no automatic global initialization routine.

Initialization must be performed explicitly from main.

10.11 Important Points to Note

Modules are simple .zax files.

Only exported symbols are public.

Linking is deterministic and compile-time only.

No runtime loader exists.

No symbol overloading.

No hidden initialization code.

extern declares boundaries without definitions.

10.12 Summary

ZAX modules provide:

Controlled visibility

Deterministic linking

Static structure

Explicit boundaries

They allow large programs without sacrificing predictability.

What Comes Next

Chapter 11 introduces:

Advanced layout patterns

Interfacing with binary data

Memory-mapped structures

Hardware register modeling

Practical layout strategies

This moves from language structure to systems-level design.

If you want, we now continue into systems-level patterns — which is where ZAX becomes very powerful for firmware and hardware work.
Excellent — now we move from language structure to systems-level layout and hardware modeling.

This chapter mirrors Pascal's "Files / Records / Real-world data" section — but adapted to ZAX's strengths: memory-mapped I/O, binary layouts, and deterministic storage.

We keep the same teaching pattern.

ZAX Quick Guide

Chapter 11 — Binary Layout, Memory-Mapped Data, and Hardware Modeling

---

11.1 Introduction

ZAX is designed for systems programming.

Typical use cases include:

Memory-mapped hardware registers

Structured binary file formats

ROM data blocks

DMA buffers

Protocol headers

ZAX provides:

Deterministic record layout

sizeof and offsetof

Explicit padding

Direct address access

No hidden packing

This chapter shows how to model real memory layouts safely and predictably.

11.2 Modeling a Hardware Register Block

Suppose a device has registers at address 0x8000:

Offset	Register
0	Status
1	Control
2	Data Low
3	Data High

Define the Layout

```
type DeviceRegs
  status: byte
  control: byte
  data: word
```

```
end

Natural size = 4
Storage size = 4
```

Bind to Fixed Address

```
const
  DEVICE_BASE = 0x8000
end

globals
  device: DeviceRegs @ DEVICE_BASE
end
```

(@ indicates placement at fixed address — assuming your spec supports absolute placement.)

Accessing Registers

```
ld a, device.status
ld device.control, 1
ld hl, device.data
```

Lowering uses constant offsets.

When Compiled…

Equivalent to:

```
LD A,(8000h)
LD (8001h),1
LD HL,(8002h)
```

No runtime address computation required.

11.3 Structured Binary File Format

Example: a simple file header.

Offset Field

0 Magic (2 bytes)
2 Version (1 byte)
3 Flags (1 byte)
4 Length (2 bytes)

Define Structure

```
type FileHeader
  magic: word
  version: byte
  flags: byte
  length: word
end
```

Natural size = 6
Rounded to next power-of-two → 8

Compiler emits padding warning.

Use sizeof

```
const
  HEADER_SIZE = sizeof(FileHeader)
end
```

HEADER_SIZE = 8

11.4 Explicit Packed Control

To avoid unexpected padding:

```
type FileHeader
  magic: word
  version: byte
  flags: byte
  length: word
  _pad: byte[2]
end
```

Now natural size = 8
No warning.

11.5 Overlaying Multiple Interpretations (Union)

Suppose the same memory region may be:

A 16-bit value

Two 8-bit values

Define Union

```
type Data union
  as_word: word
  as_bytes: byte[2]
end
```

Storage size = 2

Example Usage

```
globals
  d: Data
end

ld d.as_word, 0x1234
ld a, d.as_bytes[0]
```

Offset always 0 for union members.

11.6 Memory-Mapped Frame Buffer Example

Assume video RAM begins at 0x9000:

```
globals
  framebuffer: byte[1024] @ 0x9000
end
```

Write pixel:

```
ld framebuffer[10], 0xFF
```

Constant index → no runtime atom.

11.7 Indexed Hardware Access

```
ld hl, 0
ld framebuffer[HL], 0xAA
```

Element size = 1 → no scaling
One runtime atom → valid

11.8 Designing for Power-of-Two Scaling

When designing record types for arrays:

Prefer total sizes that are powers of two.

Bad:

5-byte struct → padded to 8
6-element array → padded again

Better:

8-byte struct → predictable stride

Hardware and binary layouts should be designed with this in mind.

11.9 Practical Layout Guidelines

1. Align composite types to power-of-two sizes explicitly.

2. Use sizeof for buffer calculations.

3. Use offsetof for protocol field references.

4. Avoid multi-atom address expressions.

5. Stage complex indexing explicitly.

11.10 Important Points to Note

ZAX storage model is ideal for memory-mapped I/O.

No hidden structure packing exists.

Padding is visible and controllable.

Hardware layout modeling is deterministic.

sizeof and offsetof eliminate magic numbers.

11.11 Summary

ZAX enables:

Safe modeling of binary layouts

Structured access to hardware registers

Predictable scaling for arrays

Zero-overhead abstraction

This makes it well-suited for firmware and systems programming.

What Comes Next

Chapter 12 will introduce:

Practical design patterns

Structured state machines

Combining enums, select, and ops

Building maintainable low-level systems

This completes the Quick Guide arc.

If you'd like, we continue to Chapter 12 — which serves as a capstone design chapter.

Excellent.
We now reach the capstone chapter — the equivalent of Pascal's "advanced integration" sections.

This chapter demonstrates how the pieces combine into real, maintainable systems.

We maintain the same structure:

Concept framing

Pattern explanation

Example programs

Output behavior

Practical guidelines

ZAX Quick Guide

Chapter 12 — Design Patterns and Structured Systems

12.1 Introduction

ZAX provides:

Structured control flow

Typed layout

Deterministic addressing

Inline op expansion

Namespaced enums

Explicit call boundaries

This chapter shows how to combine them into:

State machines

Device drivers

Protocol handlers

Firmware control loops

The goal is maintainable low-level systems with predictable code generation.

12.2 Pattern 1 — State Machine with enum and select

State machines are common in embedded systems.

Example — Simple Device State

```
enum DeviceState
  Idle
  Busy
  Error
end

globals
  state: byte
end
```

Main Loop

```
export func main(): void

loop:
  select state
    case DeviceState.Idle:
      ; check input
      ld state, DeviceState.Busy

    case DeviceState.Busy:
      ; perform work
      ld state, DeviceState.Idle

    case DeviceState.Error:
      ; recovery
      ld state, DeviceState.Idle
  end

  jr loop
end
```

Behavior

State transitions are explicit.

Enum improves clarity.

Lowering is bounded conditional branching.

12.3 Pattern 2 — Hardware Driver Skeleton

Combine:

Structured types

Memory mapping

Ops

Control flow

Define Register Block

```
type UART
  status: byte
  control: byte
  data: byte
end
```

Bind to Address

```
globals
  uart: UART @ 0xA000
end
```

Define Helper Op

```
op wait_ready()
wait:
  ld a, uart.status
  cp 1
  jr NZ, wait
end
```

Send Byte Function

```
func uart_send(value: byte): void
  wait_ready()
  ld uart.data, value
  ret
end
```

When Compiled…

No hidden call overhead from wait_ready

Address offsets resolved at compile time

Loop lowering uses direct jumps

12.4 Pattern 3 — Structured Table Dispatch

Example: command interpreter.

Command Enum

```
enum Command
  CmdRead
  CmdWrite
  CmdReset
end
```

Dispatch

```
func handle_command(cmd: byte): void
  select cmd
    case Command.CmdRead:
```

```
      ; read handler
    case Command.CmdWrite:
      ; write handler
    case Command.CmdReset:
      ; reset handler
  end
  ret
end
```

This replaces chains of manual cp and jr with structured clarity.

12.5 Pattern 4 — Layered Abstraction

Use:

op for micro-operations

func for boundary-safe modules

enum for symbolic clarity

Records for layout

Example layering:

Hardware Register Access → op
Device Operation → func
System Behavior → select + state machine

Each layer preserves predictability.

12.6 Maintaining Predictability

When designing large systems:

1. Keep composite types power-of-two aligned.

2. Avoid multi-atom expressions.

3. Use sizeof for buffer lengths.

4. Use explicit names for width-specific functions.

5. Keep ops small and mechanical.

6. Use enums for all symbolic state.

12.7 Example — Complete Mini Firmware Loop

```
enum Mode
  Normal
  Panic
end

globals
  mode: byte
end

export func main(): void

main_loop:

  if Z
    ld mode, Mode.Panic
  end

  select mode
    case Mode.Normal:
```

```
      ; normal work
    case Mode.Panic:
      ; safe mode
  end

  jr main_loop
end
```

Execution Model

Flags drive branch logic.

State transitions are explicit.

No hidden scheduler.

Entire behavior visible in assembly.

12.8 Important Points to Note

ZAX encourages layered design.

Inline ops eliminate macro fragility.

Enums prevent namespace pollution.

Records provide safe layout abstraction.

Control flow improves clarity without runtime cost.

The storage model ensures bounded indexing cost.

12.9 Summary

ZAX enables:

Structured low-level systems

Deterministic memory modeling

Predictable branching

Zero-overhead abstractions

It is a language for writing maintainable assembly — not escaping it.

End of Quick Guide

At this point, the reader has seen:

1. Overview

2. Storage model

3. Addressing

4. Compile-time expressions

5. Control flow

6. Functions

7. Ops

8. Enums

9. Structured data

10. Modules

11. Hardware modeling

12. System design patterns

If you'd like, next we can:

Refine the entire Quick Guide into a single cohesive .md document

Insert more output blocks to match Pascal’s density

Or now audit the whole structure against your v0.2 spec for inconsistencies

Your move.

Good. You want:

(2) Insert more output blocks to match Pascal density

(3) Audit the whole structure against your v0.2 spec

I’ll give you a concrete, implementation-ready plan for both.

PLAN A — Increase Output Density (Match Pascal’s Rhythm)

The Pascal guide constantly reinforces concepts with:

1. Small program

2. “When compiled and executed…”

3. Concrete output block

Your current ZAX draft is structurally correct, but under-demonstrated.

A1 — Add Output Blocks to Every Chapter

Every chapter should contain at least:

1 minimal runnable example

1 multi-line example

1 explicit “Lowered form” block

1 “When executed…” result block

Storage Chapter Needs:

Example showing padding warning output

Example printing sizeof

Example showing memory offsets explicitly

Addressing Chapter Needs:

Example showing correct multi-atom rejection message

Example showing nested constant indexing

Explicit lowering trace (with ADD HL,HL shifts shown)

Control Flow Chapter Needs:

One complete loop example with visible register results

One select example showing explicit compare/jump lowering

Show flags before and after cp

Functions Chapter Needs:

Explicit stack/register diagram

Return value demonstration in HL

Boundary-clobber example

Ops Chapter Needs:

Overload resolution example

Hygiene demonstration with expanded labels

Show two expansions side by side

Modules Chapter Needs:

Two-module build example

Symbol resolution error example

Duplicate export error output

Hardware Chapter Needs:

Full driver example

Explicit compiled addresses shown

Memory table showing byte layout

A2 — Standardize Output Block Format

Pascal Quick Guide format:

When the above code is compiled and executed, it produces the following result:

For ZAX, use:

When compiled and executed in a Z80 environment, the following state is observed:

Then show:

Register state

Memory changes

Jump flow

Emitted instruction sequence (short form)

Consistency is key.

A3 — Add “Important Points to Note” After Every Major Example

Pascal does this heavily.

After each example add:

4–6 tight bullets

No philosophy

Just mechanical reinforcement

PLAN B — v0.2 Spec Audit

Now the critical part.

Your tutorial must not contradict the normative spec.

Here's the audit checklist.

B1 — Storage Semantics

Must match:

Power-of-two storage rounding

No packed size concept

sizeof returns storage size

Padding is visible

Ensure no tutorial example ever implies packed layout.

B2 — Addressing Rules

Must enforce:

One runtime atom

arr[HL] = direct 16-bit index

arr[(HL)] = indirect

Nested indexing limited

Atom counting examples consistent

We must scan every example and verify no accidental 2-atom example slipped in.

B3 — Typed Scalars

Tutorial must reflect:

Typed variables behave as values

No legacy paren-deref model

No implicit pointer deref behavior

Double-check field access examples don’t imply hidden dereference.

B4 — Function Boundaries

Ensure:

Non-void returns use HL only

Void functions expose no boundary-visible clobbers

No example accidentally suggests overloading

No implicit prologue shown

B5 — Ops Restrictions

Must reflect:

No var blocks inside ops

Hygiene required

idx16 matcher behavior correct

cc matcher allowed

No stack interaction

Verify examples don’t introduce hidden stack usage.

B6 — Enums

Ensure:

All enum member uses are qualified

No unqualified example slipped in

Enums treated as integers at runtime

B7 — HEX Output

Ensure:

No example references >64KB addresses

No extended HEX assumption

No banking model implied

Recommended Execution Order

1. First: Perform full spec consistency audit.

2. Then: Add output density improvements.

3. Then: Freeze Quick Guide v0.2.

4. Only after freeze: Write formal spec cross-references.

Final Structural Recommendation

If you want Pascal-level authority:

Add one final appendix:

> Appendix A — Common Compile Errors and Diagnostics

Include:

Multi-atom error

Unqualified enum error

Duplicate export error

Non-power-of-two padding warning

Overload resolution ambiguity

Undefined symbol

This matches the “reference density” style of Pascal’s operator tables.

If you want, next I can:

Perform the full consistency audit chapter-by-chapter

Or rewrite one chapter fully upgraded with Pascal-level output density

Or produce the full consolidated Table of Contents with section numbering

Tell me which direction.