

# DSA Project Roadmap & Task Breakdown

**Damn Simple Architecture** — Full ecosystem development plan including emulator, assembler, compiler, debugger, and tooling infrastructure.

---

## Table of Contents

- 1. Phase 1: Foundation & Core Infrastructure
  - 1.1 Binary Format & Linking System
  - 1.2 Assembler Rewrite
  - 1.3 Documentation Updates
- 2. Phase 2: Compiler Development
  - 2.1 Language Design & Implementation
  - 2.2 Standard Library
- 3. Phase 3: Build System & Package Management
  - 3.1 Build System
  - 3.2 Package Management System
- 4. Phase 4: Debugger & Development Tools
  - 4.1 Debug Symbol System
  - 4.2 Debugger Implementation
  - 4.3 Enhanced Editor Integration
- 5. Phase 5: Integration & Polish
- 6. Phase 6: Future Enhancements (NTH)
- 7. Summary Timeline
- 8. Critical Path
- 9. Recommended Work Order

---

## Phase 1: Foundation & Core Infrastructure

**Estimated Duration: 3–4 weeks**

---

### 1.1 Binary Format & Linking System

**Priority: CRITICAL** — Everything depends on this.  
**Total Estimate: 1.5 weeks**

---

### 1.1.1 Design New Binary Format Specification

**Estimate:** 2 days

**Dependencies:** None

**Deliverable:** `docs/binary-format-spec.md`

- ☐ Research existing object file formats (ELF, COFF, Mach-O) for inspiration
  - ☐ Design `.dsb` object file format specification
  - ☐ Symbol table structure
  - ☐ Relocation table format
  - ☐ Section definitions (code, data, rodata, bss)
  - ☐ Debug information structure
  - ☐ Metadata headers
  - ☐ Design `.dse` executable format specification
  - ☐ Entry point definition
  - ☐ Memory layout requirements
  - ☐ Linking metadata
  - ☐ Document format specifications in markdown
  - ☐ Create format version strategy for future compatibility
- 

### 1.1.2 Implement DSB Object File Writer

**Estimate:** 3 days

**Dependencies:** 1.1.1

**Deliverable:** `dsa-binary-format` crate v0.1.0

- ☐ Create new crate: `dsa-binary-format`
  - ☐ Implement object file structures
  - ☐ Header structure
  - ☐ Symbol table builder
  - ☐ Section manager
  - ☐ Relocation entry creator
  - ☐ Write serialization logic
  - ☐ Add validation and error handling
  - ☐ Write unit tests for each structure
  - ☐ Integration tests for complete object files
- 

### 1.1.3 Build Linker Program

**Estimate:** 4 days

**Dependencies:** 1.1.2

**Deliverable:** `dsa-link` executable

- ☐ Create new crate: `dsa-linker`
  - ☐ Implement symbol resolution
  - ☐ Global symbol table
  - ☐ Symbol conflict detection
  - ☐ Weak symbol handling
  - ☐ Implement relocation processing
  - ☐ Address calculation
  - ☐ Patch generation
  - ☐ Cross-section references
  - ☐ Build executable generator
  - ☐ Combine sections
  - ☐ Generate final memory layout
  - ☐ Write `.dse` output
  - ☐ Add linker script support (basic)
  - ☐ Comprehensive error messages
  - ☐ Test suite with complex linking scenarios
- 

## 1.2 Assembler Rewrite

**Priority: HIGH** — Required for all compiled code.

**Total Estimate: 1.5 weeks**

---

### 1.2.1 Assembler Architecture Design

**Estimate: 1 day**

**Dependencies: 1.1.1**

**Deliverable:** `docs/assembler-architecture.md`

- ☐ Design multi-pass architecture
  - ☐ Pass 1: Symbol collection
  - ☐ Pass 2: Macro expansion
  - ☐ Pass 3: Code generation
  - ☐ Pass 4: Relocation generation
  - ☐ Plan error handling strategy
  - ☐ Design threading model for parallel file processing
  - ☐ Define module/import resolution system
  - ☐ Plan integration points with DSC compiler
- 

### 1.2.2 Implement Core Assembler

**Estimate:** 5 days

**Dependencies:** 1.1.2, 1.2.1

**Deliverable:** `dsa-asm` executable v2.0.0

- ☐ Create new crate: `dsa-assembler-ng` (next-gen)
  - ☐ Implement lexer with better error recovery
  - ☐ Build parser with detailed error messages
  - ☐ Instruction parsing
  - ☐ Directive handling
  - ☐ Macro system
  - ☐ Include resolution
  - ☐ Symbol table management
  - ☐ Code generator outputting to DSB format
  - ☐ Multi-threading for file parsing
  - ☐ Comprehensive test suite
  - ☐ Error message testing
- 

### 1.2.3 Import System & DSC Integration

**Estimate:** 2 days

**Dependencies:** 1.2.2, 2.1.2

**Deliverable:** Working import system

- ☐ Design import protocol between DSC and assembler
  - ☐ Implement symbol table merging
  - ☐ Handle pre-compiled object imports
  - ☐ Test DSC → Assembly → Object pipeline
  - ☐ Document integration process
- 

## 1.3 Documentation Updates

**Priority:** MEDIUM — Can be done alongside development.

**Total Estimate:** 3 days (distributed)

---

### 1.3.1 Update Assembly Documentation

**Estimate:** 1 day

**Dependencies:** 1.2.2

**Deliverable:** Updated `docs/dsa-assembly-reference.md`

- ☐ Review all instruction documentation
  - ☐ Document new pseudo-instructions
  - ☐ Update calling convention docs
  - ☐ Add examples for new features
  - ☐ Document assembler directives
  - ☐ Macro system documentation
- 

### 1.3.2 Architecture Documentation

**Estimate:** 1 day

**Dependencies:** None (can start anytime)

**Deliverable:** `docs/dsa-architecture.md`

- ☐ Document ISA specification
  - ☐ Memory model documentation
  - ☐ Interrupt handling
  - ☐ Hardware peripheral specs
  - ☐ Timing/performance characteristics
- 

### 1.3.3 Build Tools Documentation

**Estimate:** 1 day

**Dependencies:** 1.2.2, 1.1.3, 3.1.2

**Deliverable:** `docs/build-tools-guide.md`

- ☐ Assembler usage guide
  - ☐ Linker usage guide
  - ☐ Build system guide
  - ☐ Tutorial: Building a simple program
  - ☐ Tutorial: Multi-file projects
- 

## Phase 2: Compiler Development

**Estimated Duration:** 3–4 weeks

---

### 2.1 Language Design & Implementation

**Priority:** HIGH — Core functionality.

**Total Estimate:** 2.5 weeks

---

#### 2.1.1 Language Syntax Design

**Estimate:** 2 days

**Dependencies:** None

**Deliverable:** `docs/language-spec.md`

- ☐ Define syntax goals (simplicity, systems programming)
  - ☐ Design type system
  - ☐ Primitive types
  - ☐ Pointers/references
  - ☐ Structs
  - ☐ Arrays
  - ☐ Function types
  - ☐ Control flow syntax
  - ☐ Function declaration syntax
  - ☐ Module/import system
  - ☐ Operator precedence
  - ☐ Write EBNF grammar
  - ☐ Create example programs
- 

### 2.1.2 Lexer & Parser Implementation

**Estimate:** 4 days

**Dependencies:** 2.1.1

**Deliverable:** Parser in `dsc-compiler` crate

- ☐ Adapt existing C lexer to new syntax
  - ☐ Implement new parser for designed syntax
  - ☐ AST node definitions
  - ☐ Error recovery mechanisms
  - ☐ Comprehensive parser tests
  - ☐ Syntax error message quality testing
- 

### 2.1.3 Code Generation Improvements

**Estimate:** 5 days

**Dependencies:** 2.1.2, 1.2.2

**Deliverable:** Working code generator

- ☐ Review and fix existing codegen issues
  - ☐ Implement missing language features
  - ☐ Structs
  - ☐ Arrays
  - ☐ Pointers/memory operations
  - ☐ For loops
  - ☐ Switch statements
  - ☐ Break/continue
  - ☐ Optimize register allocation further
  - ☐ Implement proper function calling conventions
  - ☐ Add constant folding optimization
  - ☐ Dead code elimination
  - ☐ Test each feature thoroughly
- 

### 2.1.4 Type Checking & Semantic Analysis

**Estimate:** 3 days

**Dependencies:** 2.1.2

**Deliverable:** Type checker integrated in compiler

- ☐ Implement type checker
  - ☐ Symbol table for scoping
  - ☐ Type inference where applicable
  - ☐ Const checking
  - ☐ Definite assignment analysis
  - ☐ Comprehensive semantic error messages
  - ☐ Test suite for type errors
- 

## 2.2 Standard Library

**Priority:** MEDIUM — Needed for useful programs.

**Total Estimate:** 1 week

---

### 2.2.1 Core Runtime Library (in Assembly)

**Estimate:** 3 days

**Dependencies:** 1.2.2

**Deliverable:** `lib/runtime/` directory

- ☐ Memory allocation (malloc/free)
  - ☐ String operations
  - ☐ Math functions
  - ☐ I/O functions (improved print, read)
  - ☐ System call interface
  - ☐ Tests for each function
- 

## 2.2.2 Standard Library (in DSC)

**Estimate:** 2 days

**Dependencies:** 2.1.3, 2.2.1

**Deliverable:** `lib/std/` directory

- ☐ String module
  - ☐ Collections (array utilities, maybe simple list)
  - ☐ File I/O module
  - ☐ Math utilities
  - ☐ Tests and examples
- 

## Phase 3: Build System & Package Management

**Estimated Duration:** 2–3 weeks

---

### 3.1 Build System

**Priority:** HIGH — Required for complex projects.

**Total Estimate:** 1.5 weeks

---

#### 3.1.1 Build System Design

**Estimate:** 1 day

**Dependencies:** None

**Deliverable:** `docs/build-system-design.md`

- ☐ Define project structure conventions
  - ☐ Design build manifest format (`dsa-project.toml` or similar)
  - ☐ Dependency resolution strategy
  - ☐ Build cache design
  - ☐ Incremental build strategy
  - ☐ Multi-target support
-



### 3.1.2 Build Tool Implementation

**Estimate:** 5 days

**Dependencies:** 3.1.1, 1.2.2, 1.1.3, 2.1.3

**Deliverable:** `dsa-build` executable

- ☐ Create crate: `dsa-build`
  - ☐ Manifest parser
  - ☐ Dependency graph builder
  - ☐ Task orchestrator
  - ☐ Compilation tasks
  - ☐ Assembly tasks
  - ☐ Linking tasks
  - ☐ Build cache implementation
  - ☐ Parallel build support
  - ☐ Clean, rebuild commands
  - ☐ Watch mode for development
  - ☐ Comprehensive tests
- 

### 3.1.3 Project Management Commands

**Estimate:** 2 days

**Dependencies:** 3.1.2

**Deliverable:** Enhanced `dsa-build` with project management

- ☐ `dsa new <project>` — Create new project
  - ☐ `dsa init` — Initialize in existing directory
  - ☐ `dsa add <dependency>` — Add dependency
  - ☐ Binary vs library project types
  - ☐ Template system for project scaffolding
  - ☐ Documentation for each command
- 

## 3.2 Package Management System

**Priority:** MEDIUM — Enables code sharing.

**Total Estimate:** 1.5 weeks

---

### 3.2.1 Package Registry Design

**Estimate:** 2 days

**Dependencies:** 3.1.1

**Deliverable:** `docs/package-registry-design.md`

- ☐ Decide: Git monorepo vs custom hosting
  - ☐ Design package naming conventions
  - ☐ Version resolution strategy (semver)
  - ☐ Package manifest format
  - ☐ Security considerations
  - ☐ Package storage format (source/binary/both)
  - ☐ API design for registry server
- 

### 3.2.2 Local Package Manager Tool

**Estimate:** 4 days

**Dependencies:** 3.2.1, 3.1.2

**Deliverable:** `dsa-pkg` tool integrated with `dsa-build`

- ☐ Create crate: `dsa-pkg`
  - ☐ Package index synchronization
  - ☐ Dependency resolver
  - ☐ Package download/cache system
  - ☐ Integration with build system
  - ☐ Commands:
    - ☐ `dsa install <package>`
    - ☐ `dsa publish`
    - ☐ `dsa search <query>`
    - ☐ `dsa update`
  - ☐ Lock file generation
  - ☐ Test with mock registry
- 

### 3.2.3 Package Registry Implementation

**Estimate:** 3 days

**Dependencies:** 3.2.1

**Deliverable:** Package registry (URL or repo)

- ☐ If **Git monorepo** approach:
  - ☐ Set up repository structure
  - ☐ CI/CD for validation
  - ☐ Submission process
  - ☐ Package browser website
  - ☐ If **custom hosting**:
  - ☐ Simple web server (Rust + Axum/Actix)
  - ☐ Package upload API
  - ☐ Package search API
  - ☐ Basic web UI
  - ☐ Database for metadata
  - ☐ Documentation for publishing
- 

## Phase 4: Debugger & Development Tools

**Estimated Duration: 3–4 weeks**

---

### 4.1 Debug Symbol System

**Priority: HIGH** — Foundation for debugging.  
**Total Estimate: 1 week**

---

#### 4.1.1 Debug Symbol Format Design

**Estimate: 1 day**

**Dependencies:** 1.1.1

**Deliverable:** `docs/debug-symbol-format.md`

- ☐ Design symbol table format
  - ☐ Function addresses → names
  - ☐ Line number → address mapping
  - ☐ Variable location information
  - ☐ Type information
  - ☐ Define symbol table file format
  - ☐ Plan for embedding in DSE/DSB files
- 

#### 4.1.2 Symbol Generation in Tools

**Estimate: 3 days**

**Dependencies:** 4.1.1, 1.2.2, 2.1.3

**Deliverable:** Debug symbols in build output

- ☐ Modify assembler to emit debug symbols
  - ☐ Modify compiler to emit debug symbols
  - ☐ Source file/line tracking
  - ☐ Variable scope tracking
  - ☐ Linker merges debug symbols
  - ☐ Test symbol generation pipeline
- 

### 4.1.3 Symbol Table Loader in Emulator

**Estimate:** 2 days

**Dependencies:** 4.1.2

**Deliverable:** Symbol loading in emulator crate

- ☐ Implement symbol table parser
  - ☐ Build address → symbol lookup (HashMap)
  - ☐ Build symbol → address lookup
  - ☐ Memory efficient storage
  - ☐ Tests for symbol resolution
- 

## 4.2 Debugger Implementation

**Priority:** HIGH — Major productivity boost.

**Total Estimate:** 2 weeks

---

### 4.2.1 Core Debugger Features

**Estimate:** 5 days

**Dependencies:** 4.1.3

**Deliverable:** Debugger backend

- ☐ Execution control
- ☐ Step instruction
- ☐ Step over function calls
- ☐ Continue to breakpoint
- ☐ Run to cursor
- ☐ Breakpoint system
- ☐ Address breakpoints
- ☐ Conditional breakpoints
- ☐ Watchpoints (memory access)
- ☐ Register inspection
- ☐ Memory inspection
- ☐ Stack trace generation
- ☐ Test debugger commands

---

## 4.2.2 Disassembler with Symbol Resolution


**Estimate:** 3 days

**Dependencies:** 4.1.3

**Deliverable:** Enhanced disassembler

- ☐ Instruction decoder
  - ☐ Format with labels instead of addresses
  - ☐ Show function names at call sites
  - ☐ Inline comments with variable names
  - ☐ Color coding for instruction types
  - ☐ Tests for disassembly output
- 

## 4.2.3 Pseudo-Instruction Decompiler

 **COMPLEX TASK** — Separate pass to decompile assembly into readable pseudo-instructions.

**Estimate:** 4 days

**Dependencies:** 4.2.2

**Deliverable:** Pseudo-instruction view mode

- ☐ Pattern recognition for common sequences
  - ☐ Function prologue/epilogue
  - ☐ Multiplication using shifts/adds
  - ☐ Division
  - ☐ Conditional moves
  - ☐ Control flow reconstruction
  - ☐ If/else detection
  - ☐ Loop detection
  - ☐ Switch statement detection
  - ☐ Expression reconstruction
  - ☐ Format as higher-level pseudo-code
  - ☐ Extensive pattern testing
- 

## 4.2.4 Execution History Tracking

**Estimate:** 2 days

**Dependencies:** 4.2.1

**Deliverable:** Execution trace feature

- ☐ Circular buffer for instruction history
  - ☐ Register state snapshots over time
  - ☐ Configurable history depth
  - ☐ Efficient memory usage
  - ☐ Playback/reverse debugging (basic)
  - ☐ Export trace to file
- 

## 4.3 Enhanced Editor Integration

**Priority: MEDIUM** — UX improvement.

**Total Estimate: 1 week**

---

### 4.3.1 Tiling Window System

**Estimate: 2 days**

**Dependencies:** None (UI work)

**Deliverable:** Panel system in emulator

- ☐ Research Rust tiling libraries (`egui_tiles`, or custom)
  - ☐ Design panel layout system
  - ☐ Code editor panel
  - ☐ Disassembly panel
  - ☐ Register panel
  - ☐ Memory panel
  - ☐ Console panel
  - ☐ Implement drag-and-drop panel management
  - ☐ Save/load layouts
- 

### 4.3.2 Assembly Editor Improvements

**Estimate: 2 days**

**Dependencies:** 4.3.1

**Deliverable:** Enhanced assembly editor

- ☐ Syntax highlighting for DSA assembly
  - ☐ Auto-completion for instructions
  - ☐ Label/symbol auto-completion
  - ☐ Error highlighting
  - ☐ Inline documentation tooltips
  - ☐ Jump-to-definition for labels
- 

### 4.3.3 High-Level Language Editor

**Estimate: 2 days**

**Dependencies:** 4.3.1, 2.1.4

**Deliverable:** DSC language editor

- ☐ Syntax highlighting for DSC language
  - ☐ Basic auto-completion
  - ☐ Bracket matching
  - ☐ Error highlighting from compiler
  - ☐ Go-to-definition (using debug symbols)
  - ☐ Inline type hints
- 

#### 4.3.4 Integrate Build Tools in Editor

**Estimate: 1 day**

**Dependencies:** 4.3.1, 3.1.2

**Deliverable:** Integrated build experience

- ☐ Build button/command in UI
  - ☐ Show build output in console panel
  - ☐ Error navigation (click to jump to source)
  - ☐ Hot reload on successful build
  - ☐ Build status indicator
- 

### Phase 5: Integration & Polish

**Estimated Duration: 1–2 weeks**

---

#### 5.1 Tool Integration

**Priority: HIGH** — Everything works together.

**Total Estimate: 1 week**

---

##### 5.1.1 Unified Toolchain

**Estimate: 3 days**

**Dependencies:** All previous phases

**Deliverable:** dsa unified command-line tool

- ☐ Create meta-crate: `dsa-tools`
  - ☐ Unified CLI with subcommands
  - ☐ `dsa build`
  - ☐ `dsa run`
  - ☐ `dsa debug`
  - ☐ `dsa test`
  - ☐ `dsa pkg`
  - ☐ Shared configuration system
  - ☐ Tool interop testing
  - ☐ Documentation for workflow
- 

### 5.1.2 Emulator Integration

**Estimate:** 2 days

**Dependencies:** 5.1.1, 4.3.4

**Deliverable:** Fully integrated development environment

- ☐ Add build tools as emulator dependencies
  - ☐ In-editor build triggered from emulator
  - ☐ Debugger uses build output directly
  - ☐ Source-level debugging with line mapping
  - ☐ Test full edit → build → debug cycle
- 

### 5.1.3 Documentation & Tutorials

**Estimate:** 2 days

**Dependencies:** 5.1.2

**Deliverable:** Complete documentation suite

- ☐ Getting started guide
  - ☐ Full tutorial: Building a simple game
  - ☐ Debugger usage guide
  - ☐ Best practices document
  - ☐ Troubleshooting guide
- 

## Phase 6: Future Enhancements (NTH)

**Priority:** LOW — Nice to have, long-term goal.

**Estimated Duration:** 4+ weeks

---

## 6.1 Command-Line Emulator



### 6.1.1 Design Phase

**Estimate:** 1 week

**Dependencies:** None

**Deliverable:** docs/cli-emulator-design.md

- ☐ UX research for terminal-based debuggers
  - ☐ Design TUI layout (using ratatui or similar)
  - ☐ Command syntax design
  - ☐ Scripting support design
  - ☐ Accessibility considerations
- 

### 6.1.2 Implementation

**Estimate:** 3+ weeks

**Dependencies:** 6.1.1, Phase 4 complete

**Deliverable:** dsa-emu-cli executable

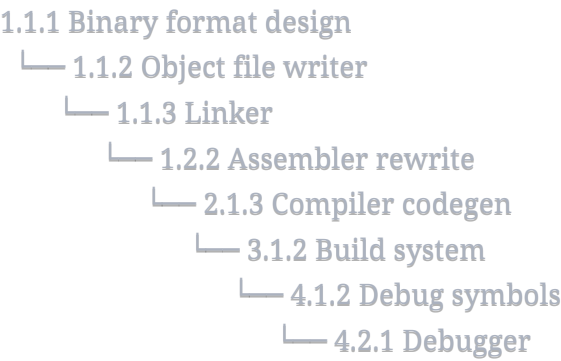
- ☐ TUI framework setup
  - ☐ Core emulator integration
  - ☐ Command parser
  - ☐ Panel rendering (code, registers, memory, etc.)
  - ☐ Keyboard shortcuts
  - ☐ Mouse support
  - ☐ Configuration system
  - ☐ Extensive usability testing
- 

## Summary Timeline

Phase	Duration	Key Dependencies
Phase 1: Foundation	3–4 weeks	None
Phase 2: Compiler	3–4 weeks	Phase 1 complete
Phase 3: Build System	2–3 weeks	Phases 1–2 complete
Phase 4: Debugger	3–4 weeks	Phases 1–3 complete
Phase 5: Integration	1–2 weeks	Phases 1–4 complete
Phase 6: CLI Emulator (NTH)	4+ weeks	Phase 4 complete

Critical Path

The following tasks are on the critical path and will block other work if delayed:



Recommended Work Order

Weeks	Focus	Tasks
1–2	Binary Format & Linker	1.1.1 → 1.1.2 → 1.1.3
3–4	Assembler Rewrite	1.2.1 → 1.2.2
5–6	Compiler Syntax & Parser	2.1.1 → 2.1.2 <i>(start 1.3 docs in parallel)</i>
7–9	Compiler Codegen & Types	2.1.3 → 2.1.4 <i>(start 2.2.1 runtime in parallel)</i>
10–11	Build System	3.1.1 → 3.1.2 → 3.1.3
12–13	Package Management <i>(if desired now)</i>	3.2.1 → 3.2.2 → 3.2.3
14–15	Debug Symbols	4.1.1 → 4.1.2 → 4.1.3
16–18	Core Debugger	4.2.1 → 4.2.2 → 4.2.4
19–20	Editor Enhancements	4.3.1 → 4.3.2 → 4.3.3 → 4.3.4
21–22	Integration & Polish	5.1.1 → 5.1.2 → 5.1.3

Notes

- Time estimates assume ~6–8 productive hours per day.
- Add **20–30% buffer** for unexpected issues.
- Testing time is included in each estimate.
- Documentation is distributed throughout rather than batched at the end.
- Package management (3.2) can be deferred if time-constrained.
- Pseudo-instruction decompiler (4.2.3) can be a stretch goal.
- CLI emulator (Phase 6) is explicitly a "nice to have" and should not block other work.