[Snapshot: 2026-01-28 — Lesson 1 complete]

Bare-Metal Forth

Implementing and Programming Forth on Microcontrollers

Preface — Scope, Philosophy, and Audience

This work is **both a course and a book**.

Audience

- Readers **new to Forth** (no prior Forth experience assumed)
- Embedded developers curious about **language–machine intimacy**
- Learners who want to understand *why* Forth looks the way it does

What this book teaches

1. **The philosophy of Forth**
    - Smallness, extensibility, and trust in the programmer
    - Interactive development and the REPL as a thinking tool
    - Words as vocabulary; vocabularies as worldviews
2. **eForth style and lineage**
    - Minimal, portable kernels
    - Clear separation of a tiny machine-dependent core and portable ideas
    - Learning by *rebuilding*, not by consuming frameworks
3. **Implementation and use, side by side**
    - How a Forth system is built on bare metal
    - How the same system is *used* interactively

What this book deliberately avoids

- Tying explanations to **one specific threading model** (ITC/STC/NTC)
- Treating Forth as a historical curiosity or niche language
- Hiding mechanisms behind opaque tooling or HALs

Threading models, memory layouts, and optimizations are treated as **design choices**, not dogma.

Historical remarks and references

- Historical notes (figForth, eForth, AmForth, Mecrisp, etc.) are included **where they clarify a design decision**
- References and corrections are woven into the narrative, not isolated in footnotes
- Code corrections and alternatives are explained as *consequences of constraints*, not as errors

Structure

- Each chapter exists as:
    - a **teaching text** (this manuscript)
    - a **self-contained assembly stage** that can be built and studied independently

The goal is not to teach "one true Forth", but to teach **how Forth comes into being on real hardware** — and how that shapes the way one thinks.

Lesson 1 — Reset, Memory, and the Empty Machine

> *Before there is a language, there is a machine. Before there is a machine, there is reset.*

Purpose of this lesson

This lesson introduces the **absolute minimum** required to bring a microcontroller from power-on into a *known, controlled state*. No Forth is executed yet. No words are defined. We deliberately stay *before* the language.

By the end of this lesson, the reader should:

- understand what happens on reset
- know why memory must be cleared deliberately
- distinguish **hardware return stack** from **software data stack**
- see memory as a *finite territory*, not an abstraction

This lesson is about **discipline**, not features.

---

1. Reset as a semantic boundary

On a microcontroller, reset is not just a restart. It is the only moment where:

- the hardware promises a defined entry point
- the programmer has full authority over initial conditions

Everything that follows inherits the quality of this moment.

On the ATtiny85, reset transfers control to address `0x0000`, the **reset vector**.

---

2. Interrupt vector table (minimal form)

In this lesson, the vector table is intentionally minimal:

- the reset vector jumps to our reset handler
- all other vectors immediately return (`RETI`)

This achieves two things:

1. the machine is safe from unintended interrupts
2. the reader sees clearly that *nothing happens unless we allow it*

---

3. Capturing the reset cause

The MCU provides a register (`MCUSR`) that records *why* the reset occurred:

- power-on
- watchdog
- brown-out
- external reset

We read this register once, at reset, and then clear it.

This teaches an important rule:

> **Information that is not captured immediately may be lost forever.**

Even if we do not use the value yet, the act of capturing it establishes good practice.

---

4. Clearing SRAM — rejecting randomness

SRAM contents after reset are **undefined**.

Any system that assumes otherwise is relying on coincidence.

We therefore explicitly clear the entire usable SRAM region:

- starting at address `0x0060`
- ending at the top of SRAM

This is not an optimization. It is a declaration:

> *There will be no ghosts from previous executions.*

---

5. Two stacks, two meanings

At this point we introduce the first crucial distinction:

- **Return stack**
  - implemented by the AVR hardware stack pointer (`SP`)
  - used implicitly by `CALL` / `RET`
- **Data stack**
  - implemented in software
  - explicitly managed by the program

In this system:

- the return stack is initialized by writing to `SP`
- the data stack pointer is stored in a register pair (`Y`)

This separation is the foundation of Forth.

---

6. Memory map as geography

Rather than treating RAM as a pool of variables, we divide it into regions:

- user area
- input buffer
- free workspace
- data stack
- return stack

Each region has:

- a start
- an end
- a purpose

The exact addresses matter less than the *act of choosing them deliberately*.

---

7. Proof of life (optional but human)

A blinking LED is not required by the machine.

It is required by the human.

A single GPIO pin configured as output and toggled in a loop serves as:

- confirmation that reset worked
- confirmation that timing exists
- confirmation that control is ours

Exercises

Exercise 1 — Memory without clearing

Comment out the SRAM clearing loop.

- Flash the program.
- Reset the device multiple times.
- Observe whether behavior remains deterministic.

Question: What assumptions did the original version refuse to make?

Exercise 2 — Moving the data stack

Change the initial data stack pointer to a lower address.

- How much space does the stack really need *at this stage*?
- What happens if it overlaps another region?

Exercise 3 — Breaking the return stack

Initialize the hardware stack pointer incorrectly.

- What fails first?
- Does the failure look predictable or chaotic?

Exercise 4 — Draw the memory map

On paper, draw the SRAM layout:

- before reset
- after reset

Label each region and its purpose.

This drawing will be reused throughout the book.

Reflection

This lesson contains no Forth code.

And yet, without it, no Forth system can exist.

Before a language can be interactive, it must be *stable*.
