# STM32L432KC Bare■Metal Notes — Startup, Linker Script, SysTick, UART

Summary of follow■up topics from our chat (generated 2026-01-26).

# 1) What you built (big picture)

- Created a minimal bare■metal firmware (no HAL, no libc startup) that boots via a vector table in Flash, runs a custom _**reset** routine, then calls **main()**.
- Used a **linker script (link.ld)** to define the Flash/SRAM memory map, place sections (.vectors/.text/.rodata/.data/.bss), and emit symbols that startup code uses.
- Confirmed build outputs (**.o**, **.elf**, **.bin**) and flashed to the Nucleo■L432KC. Got GPIO LED control working (steady ON, then blinking).
- Added **SysTick** (ARM core peripheral) to generate a 1ms timebase and used a small **timer_expired()** helper to blink + print periodically.
- Started adapting the guide's UART code (written for STM32F4) to your STM32L4: register naming differences (ISR/RDR/TDR) and different clock/AF/pin mappings.

# 2) Linker script: what it is and why you need it

A **linker script** tells the linker how to lay out your program in the MCU address space. Unlike a desktop program, an MCU has fixed Flash and SRAM locations; your code must end up at the addresses the CPU expects at boot (vectors in Flash), and writable data must end up in SRAM.

## Key jobs your link.ld performs:

- Defines physical memory regions (Flash, SRAM) with origin addresses and sizes.
- Places output sections into those regions (e.g., .vectors/.text into Flash; .data/.bss into SRAM).
- Creates **symbols** (_sbss, _ebss, _sdata, _edata, _sidata, _estack…) that your startup code references with **extern**.
- Sets the program entry point (ENTRY(_reset)) so debuggers/tools know where execution starts.

## Flash vs SRAM placement of .data

In C, initialized globals (e.g., int x = 5;) belong to the **.data** section. They must be writable at runtime (so they live in SRAM), but their initial values must be stored in non■volatile memory (Flash). The linker therefore creates two "images":

- **Load image** in Flash (where initial bytes live in the .bin/.elf).
- **Run image** in SRAM (where the variables actually reside while the program runs).

# 3) Startup/reset code: what _reset actually does

On Cortex■M, reset starts by reading the first two 32■bit words from the vector table:

- Word 0: initial stack pointer value (_**estack**)
- Word 1: reset handler address (_**reset**)

Your _**reset** then does the classic bare■metal runtime setup:

- Zero **.bss** (uninitialized globals/statics are required by C to start as 0).
- Copy **.data** initial bytes from Flash (load image) into SRAM (run image).
- Call **main()**.
- If main ever returns, loop forever.

```
// Typical reset flow (conceptually)
extern uint32_t _sbss, _ebss, _sdata, _edata, _sidata;

for (uint32_t *p = &_sbss; p < &_ebss; p++) *p = 0;          // .bss = 0
for (uint32_t *dst = &_sdata, *src = &_sidata; dst < &_edata; ) *dst++ = *src++;  // copy .data

main();
for (;;) {}
```

**Why copy .data?** Not because RAM “forgets” after power■off (that’s true), but because SRAM comes up with undefined contents after reset. C requires initialized globals to start with the initializer values, so you must copy them in on every boot/reset.

# 4) Sections and artifacts: .o vs .elf vs .bin

**main.o** is a compiled object file (machine code + relocation info). You don’t “open” it like a document; you inspect it with tools like objdump or readelf.

**firmware.elf** is the linked executable with section layout, symbols, and debug info (if enabled). This is what tools like debuggers and objdump -h use to show you where sections landed.

**firmware.bin** is the raw bytes to program into Flash (no headers/symbols), usually produced from the ELF via objcopy.

When you ran arm-none-eabi-objdump -h firmware.elf, the output told you:

- Which sections exist (.vectors, .text, .rodata, .data, .bss, …).
- Each section’s size, file offset, and the runtime address it will occupy.
- That vectors and code were placed at the Flash base (0x08000000) as intended.

# 5) GPIO recap for STM32L4: banks, registers, and BSRR

On STM32, each GPIO port (bank) is a peripheral with a block of memory■mapped registers. For STM32L4, GPIO blocks start at 0x4800_0000 and are typically spaced 0x400 apart.

```
#define GPIO(bank) ((struct gpio *)(0x48000000u + 0x400u * (bank)))
// bank = 0 => GPIOA at 0x48000000
// bank = 1 => GPIOB at 0x48000400
```

The **MODER** register is 32 bits: it stores **16 fields of 2 bits**, one field per pin (0..15). So pin *n* uses bits (2n+1:2n). To change only pin n's mode:

- Clear that 2■bit field with an AND mask.
- Set the new 2■bit value with an OR.

```
gpio->MODER &= ~(3U << (n * 2));          // 3 = 0b11 selects the 2-bit field
gpio->MODER |= ((mode & 3U) << (n * 2));    // write new 2-bit mode into that field
```

Important: these operations affect **only** the 2 bits for pin n. All other pins' mode bits remain unchanged because the mask has 1s everywhere except the target field.

### BSRR behavior (set/reset without read■modify■write)

**BSRR** is a write■only helper register: writing 1s in its low 16 bits sets corresponding output bits; writing 1s in its high 16 bits resets corresponding output bits. This lets you toggle pins atomically.

```
// val=true  => write bit in [15:0]  => set pin
// val=false => write bit in [31:16] => reset pin
gpio->BSRR = (1U << pin_number) << (val ? 0 : 16);
```

## 6) SysTick on Cortex■M: portable across STM32 families

Yes—SysTick is part of the ARM Cortex■M core, so the base address and basic behavior are the same across STM32 parts. The two things you still must choose correctly are:

- **Clock source** (CTRL.CLKSOURCE bit): core clock vs external reference.
- **Reload value** based on your current core/system clock (what frequency the core is actually running).

A simple pattern is to increment a global tick counter in the SysTick handler:

```
static volatile uint32_t s_ticks;

void SysTick_Handler(void) {
  s_ticks++;
}
```

Then you can build cooperative "timers" like timer_expired() to blink or send UART text every N milliseconds.

### Vector table zeros

In the vector table initializer, the many 0 entries are placeholders for interrupt handlers you are not using yet. A 0 means "no handler provided". If that interrupt fires, the CPU will try to branch to address 0 and likely HardFault. A common improvement is to point unused entries to a **default handler** that loops.

## 7) UART on STM32L4: the key deltas vs STM32F4

The guide's F4 UART example uses older register names (**SR**, **DR**). On STM32L4 USART, you typically use:

- **ISR** (Interrupt & Status Register) instead of SR
- **RDR** (Receive Data Register) instead of DR for reads

- **TDR** (Transmit Data Register) instead of DR for writes
- Status bits are still conceptually similar: **RXNE** is often bit 5, **TXE** often bit 7 (confirm in the L4 reference manual).
- Enable bits: **UE** (USART enable), **RE** (receiver enable), **TE** (transmitter enable) live in CR1 (bit positions must match the L4 manual).

### Clock enable and AF/pins are family■specific

Unlike SysTick, UART clock gating and pin alternate■function numbers differ per STM32 family and even per package. For STM32L432KC you found:

- USART2 is available and commonly mapped to PA2 (TX) / PA3 (RX) with AF7 on many L4 parts.
- LPUART1 may map to PA2/PA3 with AF8 (again: verify in the datasheet's Alternate Function table for your exact part/package).
- Clock enable bits live in RCC APB1ENR1/APB1ENR2/APB2ENR and differ from the F4 guide.

### Baud rate: fCK vs CPU frequency (FREQ)

The baud generator uses the **USART kernel clock** (often called **fCK** or fCK_UART), not necessarily the CPU core frequency. If your system clock setup leaves USART2's kernel clock equal to the core clock, then BRR = FREQ / baud works. If not, you must use the actual kernel clock feeding that USART (see RCC clock tree / CCIPR selections).

## 8) Host-side serial: checking tools and finding your port

On Linux, you can check whether you have cu with:

```
which cu
cu --help
```

Common ways to identify the board's serial device:

```
ls /dev/ttyACM*
ls /dev/ttyUSB*
dmesg --follow   # plug/unplug and watch
```

To stop cu safely: use the escape sequence **~.** (tilde then dot) at the beginning of a line, or close the terminal. If you see permission errors, add your user to the dialout group and re-login.

## 9) Common pitfalls you hit (and how to debug next time)

- LED turns on but doesn't blink: often the delay loop is too short/optimized away. Using a larger count and/or nop inside the loop prevents aggressive optimization.
- Sometimes it looked like you needed to flash twice: make sure you're truly rebuilding (clean old artifacts) and that the board resets after flashing. Tools sometimes leave the target running old code until reset.
- Wrong base addresses: STM32F4 GPIO base differs from STM32L4. For L4, GPIO base is typically 0x4800_0000 (not 0x4002_0000).

- UART: AF number and pin mapping must match the datasheet’s AF table for your exact part/package; F4 assumptions don’t carry over.

# Appendix: Tiny reference snippets

Packed pin ID convention used in the guide:

```
#define PIN(bank, num) ((((bank) - 'A') << 8) | (num))
# Example: PIN('B',3) => ((1 << 8) | 3) => 0x0103
# Upper byte: bank index (A=0, B=1, ...)
# Lower byte: pin number (0..15)
```

USART read/write helpers (conceptual for STM32L4 naming):

```
static inline int uart_read_ready(struct usart *u) {
  return (u->ISR & BIT(5)) != 0;   // RXNE
}
static inline uint8_t uart_read_byte(struct usart *u) {
  return (uint8_t) (u->RDR & 0xFF);
}
static inline void uart_write_byte(struct usart *u, uint8_t b) {
  while ((u->ISR & BIT(7)) == 0) {}  // TXE
  u->TDR = b;
}
```

Note: confirm exact bit positions (RXNE/TXE/UE/RE/TE) and BRR formula details in the STM32L4 reference manual for your part.