STM32F103 Flash Loader Protocol: Complete Technical Reference

Ashok Ashwin

February 8, 2026

Contents

1 Introduction

This document provides a comprehensive technical reference for the STM32F103C8 UART flash loader protocol. It covers the embedded bootloader firmware architecture, communication protocol mechanics, CRC-16 error detection, frame structure, handshake procedures, and the integration of STM32 Hardware Abstraction Layer (HAL), flash programming, and UART drivers. The document targets firmware engineers implementing or extending flash loading capabilities on STM32 microcontrollers.

1.1 Scope and Architecture Overview

This flash loader system comprises three main components:

1. **Bootloader Firmware** : Runs on the STM32F103C8 at reset, implements UART communication, flash control, and application validation
2. **Flash Loading Protocol** : Custom XModem-based protocol with CRC-16 error detection operating over UART at 115200 baud
3. **Host − Side Tools** : Python scripts implementing the protocol client, serial communication, and application image management

The integration of these components ensures robust firmware updates even in hostile environments (electrical noise, intermittent connections, power interruptions).

2 Core Concepts: HAL, UART, Flash, and CRC

2.1 1. Hardware Abstraction Layer (HAL) Architecture

Definition: HAL is the standardized interface layer between firmware and microcontroller hardware. STMicroelectronics provides HAL libraries abstracting registers, interrupts, and timing details.

2.1.1 Why HAL Matters for Flash Loading

Rather than manipulating raw registers, the bootloader uses HAL functions:

- **Portability:** Code remains compatible if moving from STM32F1 to STM32H7 or STM32L4
- **Abstraction:** Details of flash controller timing, cache behavior, and synchronization are hidden
- **Correctness:** Critical sequences (e.g., wait states, flash unlocking) are pre-tested by STMicroelectronics

2.1.2 HAL in the Flash Loader

Key HAL functions used:

- **HAL_RCC_*:** Clock configuration (HSE 8MHz, PLL to 72MHz, peripheral clocks)
- **HAL_UART_Transmit / Receive:** Synchronized UART I/O with timeout
- **HAL_FLASH_Unlock / Lock:** Enable/disable flash programming mode
- **HAL_FLASH_Program:** Write 16-bit or 32-bit words
- **HAL_FLASHEx_Erase:** Erase full pages (1 KB each on STM32F1)
- **SystemClock_Config():** Initialize clock tree during bootloader startup

2.1.3 Clock Configuration

The bootloader initializes:

- HSE (8 MHz crystal) as primary oscillator
- PLL with factor 9 (HSE/1 × 9 = 72 MHz core clock)
- Flash wait states set to 2 (required for 72 MHz operation)
- AHB, APB1, APB2 dividers configured for proper peripheral timing

This 72 MHz clock provides fast flash operations, deterministic UART timing, and precise timeouts.

2.2 2. UART Communication Layer

Definition: UART (Universal Asynchronous Receiver Transmitter) is a serial communication standard. The STM32F103C8 contains multiple UART instances; the bootloader uses USART1 (PA9/PA10).

2.2.1 UART Configuration

The bootloader configures USART1 as:

- Baud rate: 115200 bits/second
- Data bits: 8
- Stop bits: 1
- Parity: None
- Flow control: None (no RTS/CTS)

At 115200 baud, each bit takes ~8.68 microseconds; one byte takes ~87 microseconds; one 128-byte XModem frame takes ~11 milliseconds.

2.2.2 UART Timeout Handling

The bootloader defines critical timeouts:

- **BOOTLOADER_TIMEOUT (5 seconds):** If no START command received, auto-jump to application
- **HEARTBEAT_INTERVAL (500 ms):** Send "BOOT" message to inform host of readiness
- **UART_TIMEOUT_MS (5 seconds):** Per-byte receive timeout during frame transfer

These timeouts use HAL_GetTick() system timer for millisecond precision.

2.3 3. Flash Memory Organization and Programming

Definition: Flash is non-volatile electrically erasable programmable memory. The STM32F103C8 contains 64 KB of flash divided into 1 KB pages.

2.3.1 STM32F1 Flash Organization

```
Total Flash: 64 KB (0x08000000 - 0x0800FFFF)
Page Size: 1 KB (0x400 bytes)
Number of Pages: 64
Page 0 (0x08000000 - 0x080003FF): Bootloader
Pages 1-7 (0x08000400 - 0x08001FFF): Bootloader continuation
Pages 8-63 (0x08002000 - 0x0800FFFF): Application area
```

2.3.2 Flash Memory Controller (FMC) Registers

Key registers controlled through HAL:

- **FLASH_ACR:** Access control register (sets wait states, enables prefetch)
- **FLASH_KEYR:** Key register for unlocking flash programming
- **FLASH_CR:** Control register (enables erase/program operations)
- **FLASH_SR:** Status register (indicates busy, operation complete, errors)
- **FLASH_AR:** Address register (target address for erase)

2.3.3 Flash Write and Erase Mechanics

Program (Write) Operation:

- Flash must be unlocked (HAL_FLASH_Unlock)
- Data written in 16-bit or 32-bit words (half-word or word aligned)
- Flash controller autonomously writes and verifies
- Firmware polls SR register via HAL until BSY=0
- Takes approximately 10-40 microseconds per word

Erase Operation:

- Flash must be unlocked
- Set page address in AR register
- Set PER (Page Erase) bit in CR register
- Set STRT (Start) bit to begin erase
- Firmware polls SR.BSY until complete
- Takes approximately 20-40 milliseconds per 1 KB page

2.3.4 Detailed HAL FLASH API Reference

The STM32 HAL provides high-level functions to abstract flash hardware complexity. Here are the critical functions used in the bootloader:

1. HAL_FLASH_Unlock()

```
void HAL_FLASH_Unlock(void)
```

Purpose: Unlock flash memory for programming and erase operations.
Details:

- Flash is locked by default after reset for safety
- Internally writes magic keys to FLASH_KEYR register in sequence:
- First key: 0x45670123
- Second key: 0xCDEF89AB
- Sets flash controller to allow erase/program commands
- Must be called before any flash programming

2. HAL_FLASH_Lock()

```
void HAL_FLASH_Lock(void)
```

Purpose: Lock flash memory after programming/erase to prevent accidental writes.
Details:

- Sets LOCK bit in FLASH_CR register to 1
- Disables all erase and program operations
- Provides protection against software bugs
- Should be called immediately after HAL_FLASH_Program/HAL_FLASHEx_Erase
- Safe to call even if flash is already locked

3. HAL_FLASHEx_Erase()

```
HAL_StatusTypeDef HAL_FLASHEx_Erase(
    FLASH_EraseInitTypeDef *pEraseInit,
    uint32_t *PageError)
```

Purpose: Erase one or more pages of flash memory.
Parameters:

- **pEraseInit:** Pointer to erase configuration structure
- **TypeErase:** FLASH_TYPEERASE_PAGES (page erase, 1 KB granularity)
- **PageAddress:** Starting address of first page to erase
- **NbPages:** Number of consecutive pages to erase
- **PageError:** Output parameter: page number where error occurred (if any)

Return value:

- **HAL_OK:** Erase successful
- **HAL_ERROR:** Erase failed (flash is locked or other error)
- **HAL_TIMEOUT:** Erase operation timed out

Bootloader usage example:

```
FLASH_EraseInitTypeDef erase_init;
uint32_t page_error = 0;

erase_init.TypeErase = FLASH_TYPEERASE_PAGES;
erase_init.PageAddress = 0x08002000;  /* Application base */
erase_init.NbPages = 1;               /* Erase 1 page (1 KB) */

HAL_FLASH_Unlock();
status = HAL_FLASHEx_Erase(&erase_init, &page_error);
HAL_FLASH_Lock();

if (status != HAL_OK)
{
    printf("Erase failed
}
```

Important Notes:

- Erase time: ~20-40 ms per page on STM32F103 at 72 MHz
- Entire page is erased to 0xFFFFFFFF (all 1's)
- Bootloader pages (0-7) should NEVER be erased by application code

- After erase, page must be written to complete the cycle

4. HAL_FLASH_Program()

```
HAL_StatusTypeDef HAL_FLASH_Program(
    uint32_t TypeProgram,
    uint32_t Address,
    uint64_t Data)
```

Purpose: Write data to flash memory at specified address.
Parameters:

- **TypeProgram:** FLASH_TYPEPROGRAM_HALFWORD (16-bit) or FLASH_TYPEPROGRAM_WORD (32-bit)
- **Address:** Flash address where data will be written (must be aligned to word size)
- **Data:** Data to write (16-bit or 32-bit value)

Return value:

- **HAL_OK:** Program successful
- **HAL_ERROR:** Flash not unlocked, address misaligned, or write error
- **HAL_TIMEOUT:** Flash operation timed out

Bootloader usage (16-bit writes):

```
uint32_t flash_addr = 0x08002000;  /* Application base */
uint8_t firmware_data[128];        /* 128-byte chunk from XModem */

HAL_FLASH_Unlock();

for (int i = 0; i < 128; i += 2)
{
    /* Combine two bytes into 16-bit word (little-endian) */
    uint16_t word = (uint16_t)firmware_data[i] |
                    ((uint16_t)firmware_data[i + 1] << 8);

    status = HAL_FLASH_Program(FLASH_TYPEPROGRAM_HALFWORD,
                               flash_addr + i,
                               word);

    if (status != HAL_OK)
    {
        HAL_FLASH_Lock();
        return -1;  /* Write error */
    }
}

HAL_FLASH_Lock();
```

Key Characteristics:

- Write time: ~10-40 microseconds per 16-bit word
- Can only program 0 bits to 1 bits (not 1 to 0)
- Page must be erased first before programming
- STM32F1 supports only 16-bit and 32-bit writes (no 8-bit)
- Misaligned writes will silently fail or produce undefined behavior

2.3.5 Complete Flash Write Function from Bootloader

Here is the complete **flash_write_page()** function as implemented in main.c, demonstrating proper HAL FLASH API usage:

```
static int flash_write_page(uint32_t addr, const uint8_t *data, uint16_t size)
{
  uint32_t page_error = 0;
  FLASH_EraseInitTypeDef erase_init;
  HAL_StatusTypeDef status;
  uint16_t i;

  /* Validate address range */
  if (addr < APP_BASE_ADDR || (addr + size) > (BOOT_BASE_ADDR + FLASH_SIZE))
  {
    return -1;
  }

  /* Validate size is word-aligned */
  if (size & 1)
  {
    return -1;
  }

  /* Step 1: Unlock flash for programming */
  HAL_FLASH_Unlock();

  /* Step 2: Configure erase operation for 1KB page */
  erase_init.TypeErase = FLASH_TYPEERASE_PAGES;
  erase_init.PageAddress = addr;
  erase_init.NbPages = 1;

  /* Step 3: Erase the page */
  status = HAL_FLASHEx_Erase(&erase_init, &page_error);
  if (status != HAL_OK)
  {
    HAL_FLASH_Lock();
    return -1;
  }

  /* Step 4: Program data as 16-bit words */
  for (i = 0; i < size; i += 2)
  {
    uint16_t word = (uint16_t)data[i] | ((uint16_t)data[i + 1] << 8);
    status = HAL_FLASH_Program(FLASH_TYPEPROGRAM_HALFWORD, addr + i, word);
    if (status != HAL_OK)
    {
      HAL_FLASH_Lock();
      return -1;
    }
  }

  /* Step 5: Lock flash after programming */
  HAL_FLASH_Lock();
  return 0;
}
```

Analysis of the function:

1. **Address Validation** : Ensures write stays within application range (0x08002000 - 0x0800FFFF)
2. **Size Validation** : Checks that size is even (required for 16-bit writes)
3. **Unlock** : Enables flash programming via HAL_FLASH_Unlock()
4. **Erase** : Clears target page to 0xFF via HAL_FLASHEx_Erase()
5. **Program** : Writes data in 16-bit words with error checking
6. **Lock** : Re-protects flash immediately after write via HAL_FLASH_Lock()

This page-buffered approach minimizes erase cycles, reduces wear on flash, and fits within 20 KB RAM.

2.4 4. CRC-16-XMODEM Error Detection

Definition: CRC (Cyclic Redundancy Check) detects transmission errors. The bootloader uses CRC-16-XMODEM polynomial 0x1021 for frame validation.

2.4.1 CRC-16-XMODEM Algorithm

The CRC-16-XMODEM polynomial is:

$$G(x) = x^{16} + x^{12} + x^5 + 1$$

Algorithm:

1. Initialize CRC to 0x0000
2. For each byte in the frame:
   XOR byte into high byte of CRC
   Shift CRC left by 1 bit, 8 times
   If MSB was 1, XOR CRC with 0x1021 after each shift
3. Final CRC is appended as 16-bit big-endian value

2.4.2 CRC in Python

```
def crc16_xmodem(data):
    """Calculate XModem CRC-16 (polynomial 0x1021)"""
    crc = 0
    for byte in data:
        crc ^= byte << 8
        for _ in range(8):
            crc <<= 1
            if crc & 0x10000:
                crc ^= 0x1021
            crc &= 0xFFFF
    return crc
```

This produces a 16-bit value highly sensitive to any single-bit error. The probability of an undetected error is less than 1 in 65536.

2.4.3 Why CRC-16-XMODEM?

The XModem standard is well-tested and industry-proven. Its 16-bit CRC:

- Detects all 1-bit and 2-bit errors
- Detects most burst errors up to 16 bits
- Is computationally efficient (no large lookup tables)
- Is implemented identically on microcontroller and host

3 Foundational Concepts for Flash Loaders

Before implementing or understanding the flash loader protocol, you should be comfortable with these foundational topics.

3.1 1. Serial Communication and UART (Mandatory)

- Asynchronous communication (no shared clock between sender and receiver)
- Baud rate: Speed in bits per second (115200 for this loader)
- Frame format: Start bit, 8 data bits, 1 stop bit (10 bits total per byte)
- Timeout handling: Must handle device disconnection or power loss

3.2 2. Frame-Based Protocols (Mandatory)

A frame is a discrete message: [Header | Data | Trailer]. The flash loader uses XModem framing with:

- Clear boundaries (SOH delimiter)
- Block numbering (detect duplicates and losses)
- CRC trailer (detect payload corruption)

3.3 3. Flash Memory Organization (Mandatory)

- STM32F103C8 contains 64 KB flash at 0x08000000 - 0x0800FFFF
- Page size: 1 KB (erase granularity)
- Application starts at page 8 (0x08002000) after 8 KB bootloader
- Bootloader flash must never be erased by application

3.4 4. Bootloader State Machine (Important)

The bootloader implements a clear state machine:

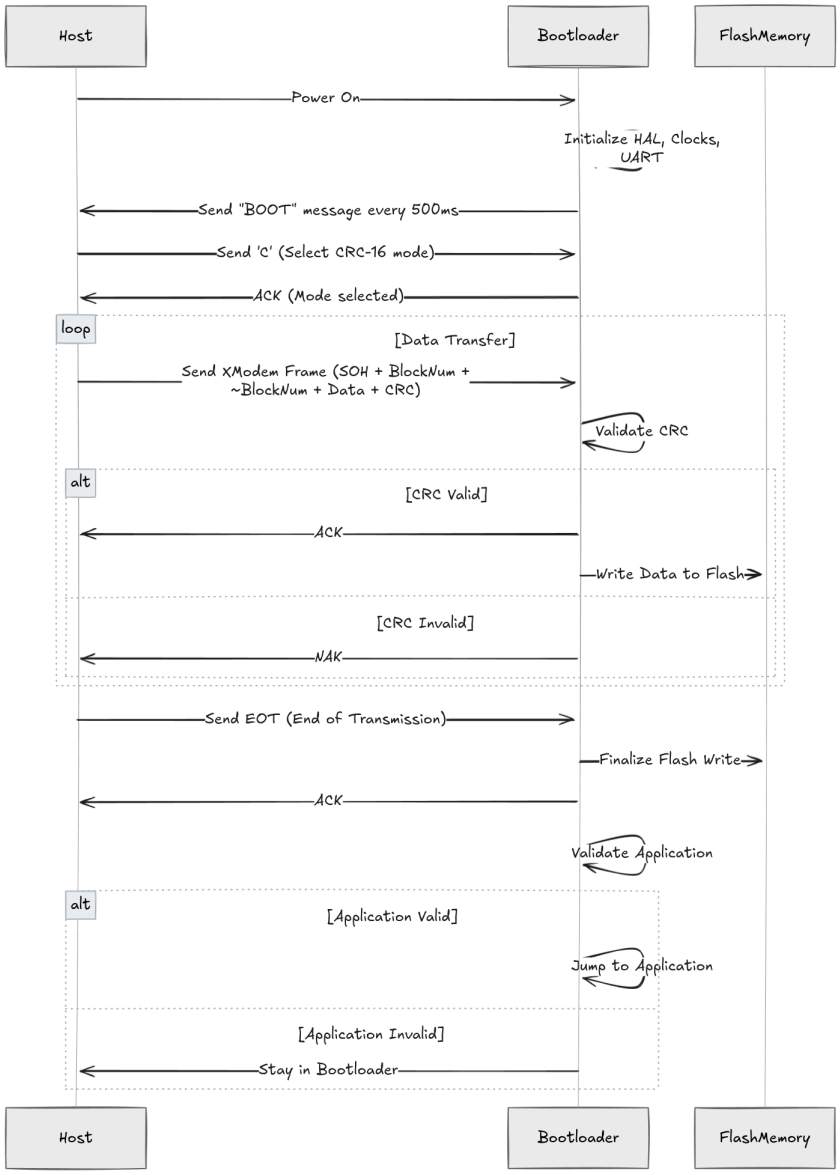
- **Heartbeat State:** Send "BOOT" every 500ms, listen for mode
- **Transfer State:** Receive XModem frames, accumulate, write flash
- **Auto-Jump:** Timeout without host response, jump to application

3.5 5. Application Validation (Important)

Before jumping to application, bootloader verifies:

- **Non-zero initial MSP:** If MSP at 0x08002000 is 0x00000000, application is invalid
- **Valid vector table:** First word must point into RAM
- **Code present:** At least one non-0xFF byte in application region

4 Flash Loader Protocol Specification



4.1 Protocol Overview

Name: STM32F103 UART Flash Loader (XModem-CRC variant)
Transport: UART at 115200 baud, 8 data bits, 1 stop bit, no parity
Physical Interface: USART1 (PA9=TX, PA10=RX) on STM32F103C8
Error Detection: CRC-16-XMODEM (polynomial 0x1021)
Frame Size: 128-byte data payload + 5-byte header + 2-byte CRC = 135 bytes
Addressing: Application space 0x08002000 - 0x0800FFFF (56 KB)

4.2 Communication Phases

4.2.1 Phase 1: Bootloader Heartbeat (Discovery)

Upon power-on or reset, bootloader enters heartbeat state:

1. Initialize clocks (72 MHz via PLL)
2. Initialize UART1 (115200 baud)

3. Validate application (check MSP and initial vector)
4. Start 5-second timeout counter
5. Send "BOOT<CR><LF>" every 500 milliseconds

```
Bootloader → Host: "BOOT<CR><LF>" (6 bytes)
Bootloader → Host: "BOOT<CR><LF>" (after 500ms)
Bootloader → Host: ... (continues until mode received or timeout)
```

Timeout Behavior: If no mode command within 5 seconds and application is valid, bootloader automatically jumps to application. This allows devices to boot without host intervention.

4.2.2 Phase 2: Mode Selection

Host receives "BOOT" message and sends mode selection:

```
Host → Bootloader: 'C' (0x43) - Select CRC-16 mode
Host → Bootloader: 'N' (0x4E) - Select checksum mode (legacy)
```

Upon receiving 'C', bootloader exits heartbeat state and begins listening for data frames.

4.2.3 Phase 3: Data Transfer Loop

```
Frame Format: SOH BlockNum ~BlockNum Data[128] CRC_H CRC_L
Host → Bootloader: 135-byte frame
Bootloader: Validate CRC, accumulate in RAM buffer
Bootloader: When page full or transfer complete, erase and write flash
Bootloader → Host: ACK (0x06) on success or NAK (0x15) on failure
Host: Wait for ACK/NAK before sending next frame
(Loop continues for each 128-byte block)
```

Error Recovery: If NAK received, host retries the same frame (same block number). Bootloader detects duplicate by block number and ignores.

4.2.4 Phase 4: End of Transfer

After all frames sent:

1. Host sends EOT (0x04)
2. Bootloader flushes any remaining buffered data to flash
3. Bootloader sends final ACK (0x06)
4. Bootloader validates complete application
5. Bootloader jumps to application at 0x08002000

```
Host → Bootloader: EOT (0x04)
Bootloader: Finalize flash write
Bootloader → Host: ACK (0x06)
Bootloader: Jump to application
```

4.3 Detailed Frame Structure

4.3.1 XModem Frame Format (CRC-16 Mode)

```
Offset   Size    Field           Purpose
-------  ------  ------          --------
0        1 byte  SOH             0x01 - marks start of frame
1        1 byte  Block#          Frame counter (1-255, wraps)
2        1 byte  ~Block#         Complement of block# (error check)
3-130    128     Payload         Application data
131-132  2       CRC-16          Big-endian CRC over Payload
Total:   135 bytes
```

4.3.2 CRC-16 Calculation

CRC is calculated over the 128-byte Payload only (not SOH, block numbers, or CRC itself).

- **Polynomial:** 0x1021 (standard XModem)
- **Initial value:** 0x0000
- **Byte order:** Big-endian (MSB first)
- **Output:** Appended as CRC_H (MSB at offset 131) then CRC_L (LSB at offset 132)

4.3.3 Block Number Wraparound

Block number increments from 1 to 255, then wraps to 1. Block zero is never used. This detects:

- Duplicate frames (same block number received twice)
- Lost frames (block number jumps)
- Out-of-order frames

```
Sequence: 1, 2, 3, ..., 254, 255, 1, 2, ...
```

4.4 Control Characters

```
Name         Value   Meaning
----------   ------  --------
SOH          0x01    Start of Header (frame delimiter)
ACK          0x06    Acknowledge (success)
NAK          0x15    Negative Acknowledge (retry)
CAN          0x18    Cancel (abort transfer)
EOT          0x04    End of Transmission
'C'          0x43    Initiate CRC-16 mode
'N'          0x4E    Initiate checksum mode
```

4.5 Timeout Specifications

```
Context                 Timeout     Purpose
----------------------  ----------  --------
Heartbeat               500 ms      Interval between "BOOT" messages
Bootloader Auto-Jump    5 seconds   Exit bootloader if no host response
Per-Frame Response      5 seconds   Wait for ACK/NAK after sending frame
Per-Byte RX             5 seconds   Timeout waiting for a single byte
Post-EOT                N/A         Bootloader immediately jumps after final ACK
```

4.6 Memory Organization During Transfer

Bootloader manages flash write as frames arrive:

- **page_buffer[1024]:** RAM-based accumulation buffer (one flash page size)
- **Buffering strategy:** 8 frames × 128 bytes = 1024 bytes = 1 complete flash page
- **Write timing:** When buffer full or EOT received, entire page is:

1. Erased (20-40 ms)
2. Written as 16-bit words (10-40 µs per word)
3. Verified by read-back

This minimizes erase cycles (erase is slow) while staying within 20 KB RAM.

4.7 Application Launch and Reset Behavior

4.7.1 Bootloader Startup and System Clock Initialization

When the STM32F103C8 exits reset, the bootloader begins executing at 0x08000000:

```
int main(void)
{
  /* Initialize HAL and system clock */
  HAL_Init();
  SystemClock_Config();

  /* Initialize UART (115200 baud for flash loader) */
  MX_USART1_UART_Init();

  /* Run flash loader protocol (heartbeat sent in flash_loader_main) */
  flash_loader_main();

  /* If we reach here, jump to application in flash */
  jump_to_application();

  /* Should never reach this point */
  while (1) { }
}
```

Key initialization steps:

1. **HAL_Init**() : Initializes HAL infrastructure (NVIC, SysTick timer, interrupt priorities)
2. **SystemClock_Config**() : Configures 72 MHz clock from 8 MHz HSE via PLL
3. **MX_USART1_UART_Init**() : Configures UART1 (PA9=TX, PA10=RX) at 115200 baud

```
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /* Enable HSE oscillator (8 MHz crystal) */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_ON;
  RCC_OscInitStruct.HSEPredivValue = RCC_HSE_PREDIV_DIV1;

  /* Configure PLL: 8 MHz * 9 = 72 MHz */
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL9;

  HAL_RCC_OscConfig(&RCC_OscInitStruct);

```

```
  /* Configure clock distribution */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK | RCC_CLOCKTYPE_SYSCLK
                              | RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;   /* AHB: 72 MHz */
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;    /* APB1: 36 MHz */
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;    /* APB2: 72 MHz */

  /* Set flash wait states for 72 MHz (requires 2 wait states) */
  HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2);
}
```

Clock Configuration Details:

- **HSE (High-Speed External):** 8 MHz crystal oscillator
- **PLL multiplication:** 8 MHz × 9 = 72 MHz system clock
- **AHB clock:** 72 MHz (no divider, full speed)
- **APB1 clock:** 36 MHz (÷2, required limit is 36 MHz max)
- **APB2 clock:** 72 MHz (full speed, used for UART)
- **Flash wait states:** 2 cycles (required at 72 MHz per datasheet)

4.7.2 Bootloader Reset Behavior

What happens during reset:

1. Hardware resets all internal state, clears RAM
2. Processor starts in reset handler defined in startup assembly (startup_stm32f103xb.s)
3. Reset handler initializes stack pointer (MSP) to 0x20005000 (top of 20 KB RAM)
4. Call SystemInit() to set initial clock configuration
5. Initialize BSS section (zero-initialized global variables)
6. Call main()

Critical reset details for flash loader:

- Flash controller is LOCKED at reset (HAL_FLASH_Unlock must be called)
- All interrupts are disabled at reset
- SysTick timer is not running (HAL_Init enables it)
- All GPIO pins default to floating input
- UART is not initialized (MX_USART1_UART_Init must be called)
- Vector table at 0x08000000 (bootloader reset vectors)

4.7.3 Application Validation Before Jump

Before jumping to application, bootloader validates:

```
static int check_application_valid(void)
{
  uint32_t app_sp = *(volatile uint32_t *)(APP_BASE_ADDR + 0);
  uint32_t app_reset = *(volatile uint32_t *)(APP_BASE_ADDR + 4);

  /* Check if stack pointer is in valid RAM range */
  if (app_sp < RAM_START_ADDR || app_sp > RAM_END_ADDR)
  {
```

```
    return 0;  /* Invalid MSP */
  }

  /* Check if reset vector is in application flash range */
  if ((app_reset < APP_BASE_ADDR) || (app_reset > (BOOT_BASE_ADDR + FLASH_SIZE)))
  {
    return 0;  /* Invalid reset vector */
  }

  /* Check if reset vector has Thumb bit set (LSB = 1) */
  if ((app_reset & 0x1) == 0)
  {
    return 0;  /* Not Thumb code */
  }

  return 1;  /* Application is valid */
}
```

Validation checks explained:

1. **Stack Pointer** (**MSP**) : Read from offset 0x08002000. Must point into RAM (0x20000000 - 0x20005000)
2. **Reset Vector** : Read from offset 0x08002004. Must point into flash (0x08002000 - 0x0800FFFF)
3. **Thumb Bit** : Reset vector LSB must be 1 (Cortex-M3 requires Thumb mode)

If any check fails, bootloader returns to heartbeat state and continues waiting for flash command.

4.7.4 Bootloader Jump-to-Application Sequence

When jumping to application, bootloader must ensure clean handoff:

```
static void jump_to_application(void)
{
  typedef void (*app_entry_t)(void);
  uint32_t app_sp;
  uint32_t app_reset;
  app_entry_t app_reset_handler;

  /* Step 1: Read application vector table */
  app_sp = *(volatile uint32_t *)(APP_BASE_ADDR + 0);
  app_reset = *(volatile uint32_t *)(APP_BASE_ADDR + 4);

  /* Step 2: Disable all interrupts before context switch */
  __disable_irq();

  /* Step 3: Stop SysTick timer */
  SysTick->CTRL = 0;

  /* Step 4: Disable all NVIC interrupts and clear pending flags */
  for (uint32_t i = 0; i < 8; i++)
  {
    NVIC->ICER[i] = 0xFFFFFFFFU;  /* Interrupt Clear Enable */
    NVIC->ICPR[i] = 0xFFFFFFFFU;  /* Interrupt Clear Pending */
  }

  /* Step 5: Deinitialize HAL peripherals (especially UART) */
  HAL_UART_DeInit(&huart1);
  HAL_DeInit();

  /* Step 6: Relocate NVIC vector table to application */
  SCB->VTOR = APP_BASE_ADDR;  /* Vector table offset register */

  /* Step 7: Set application stack pointer */
  __set_MSP(app_sp);

  /* Step 8: Jump to application reset handler */
  app_reset_handler = (app_entry_t)app_reset;
  app_reset_handler();

```

```
  /* Never reached */
  while (1);
}
```

Detailed explanation of each step:

1. **Read Vector Table** : Fetch MSP (offset 0) and Reset_Handler (offset 4) from application base
2. **Disable Interrupts** : CPSID instruction prevents interrupts during transition
3. **Stop SysTick** : Set SysTick->CTRL to 0 to halt system tick timer
4. **Disable All NVIC Interrupts** : Clear all ICER (Interrupt Clear Enable) bits and ICPR (Interrupt Clear Pending) bits
5. **HAL Deinitialization** : Call HAL_UART_DeInit() to release UART1 (prevents interference), call HAL_DeInit() for other HAL cleanup
6. **Set VTOR** : SCB->VTOR (NVIC Vector Table Offset Register) = 0x08002000 relocates interrupt vector table
7. **Set MSP** : __set_MSP() instruction sets ARM stack pointer to application's initial MSP
8. **Jump to Code** : Function pointer cast and call executes application reset handler

Why this sequence is critical:

- **Disable interrupts first:** Prevents IRQ firing while still executing bootloader code
- **Stop SysTick:** SysTick interrupt fires every 1 ms; must be disabled before jump
- **Clear NVIC interrupt flags:** Bootloader may have pending interrupts that application doesn't expect
- **Deinitialize HAL:** Stops UART reception/transmission, closes GPIO handles
- **Set VTOR:** Application's interrupt vectors now routed to 0x08002000, not bootloader's 0x08000000
- **Set MSP last:** After VTOR is set, we update stack pointer to application's expected value
- **Jump last:** Final instruction hands control to application code

After the jump, application's reset handler executes with clean state, all bootloader code and state left behind.

5 Host-Side Implementation: Python Flash Loader

The host flash_loader.py script implements the client side of the protocol.

5.1 High-Level Flow

```
1.  Open serial port (115200 baud)
2.  Wait for "BOOT" message from bootloader
3.  Send 'C' to select CRC-16 mode
4.  Read firmware binary file
5.  Pad to 128-byte blocks (fill with 0xFF)
6.  For each 128-byte block:
    a.  Construct XModem frame (SOH + BlockNum + ~BlockNum + Data + CRC)
    b.  Transmit frame
    c.  Wait for ACK/NAK
    d.  If NAK: retry (up to 3 times)
7.  Send EOT (0x04)
8.  Wait for final ACK
9.  Close serial port
```

5.2 Key Functions

5.2.1 crc16_xmodem()

Computes CRC-16 for frame data (identical algorithm as bootloader):

```
def crc16_xmodem(data):
    crc = 0
    for byte in data:
        crc ^= byte << 8
        for _ in range(8):
            crc <<= 1
            if crc & 0x10000:
                crc ^= 0x1021
            crc &= 0xFFFF
    return crc
```

5.2.2 send_firmware()

Main function orchestrating the transfer:

- Opens serial port with retry
- Waits for bootloader heartbeat
- Sends mode selection ('C')
- Loops through firmware blocks
- Builds and transmits XModem frames
- Handles ACK/NAK responses with retry logic (max 3 retries per frame)
- Sends EOT and waits for final ACK

5.3 Error Handling and Retries

Host-side error handling includes:

- **Frame Timeout:** If no ACK/NAK within 2 seconds, retry
- **CRC Mismatch NAK:** Bootloader sends NAK; host retransmits same frame
- **Max Retries:** After 3 failed attempts, abort transfer with error message
- **Serial Port Errors:** Graceful handling of disconnection

6 Conclusion and Next Steps

The STM32F103 UART Flash Loader provides a complete, robust solution for firmware updates over UART. Key takeaways:

1. **Protocol Design:** XModem-CRC is industry-standard, reliable, and simple to implement
2. **HAL Integration:** Hardware abstraction ensures portability and correctness across STM32 families
3. **State Machine:** Clear, testable bootloader logic with defined phases
4. **Error Recovery:** Timeouts, retries, and CRC validation provide robustness against real-world communication failures
5. **Hands-On Implementation:** Both firmware (C) and host (Python) code provided as complete working references

Future enhancements could include:

- Firmware signatures for cryptographic authentication
- Dual-image OTA (over-the-air) updates with rollback

- USB DFU alternative transport layer
- Watchdog-based auto-recovery for stalled updates
- Bootloader self-update capability
- Progress indication and transfer resumption

This document provides a solid foundation for understanding, implementing, and extending flash loading solutions on STM32 microcontrollers. The combination of clear protocol specification, detailed HAL layer explanation, and working code examples ensures that firmware engineers can confidently deploy reliable firmware update systems in production environments.

Appendix: Quick Reference

Bootloader Quick-Start

1. Connect STM32F103C8 to serial port (PA9/PA10)
2. Power on device
3. Observe "BOOT<CR><LF>" message every 500ms
4. Run: python3 flash_loader.py -p /dev/ttyUSB0 -f application.bin
5. After transfer completes, bootloader jumps to application

Hardware Requirements

- STM32F103C8 (or STM32F103CB for 128 KB variant)
- USB-to-UART adapter (FTDI, CH340, etc.)
- 5V power supply
- UART connections: TX ⟶ PA9, RX ⟶ PA10, GND ⟶ GND

Memory Map Summary

```
0x08000000 - 0x08001FFF:  Bootloader (8 KB)
0x08002000 - 0x0800FFFF:  Application (56 KB)
0x20000000 - 0x20005000:  RAM (20 KB)
Page Buffer @ RAM:        1 KB for flash write buffering
```

Frame Checksum Verification

Always verify:

1. Block number consistency (byte 1 = ~byte 2)
2. CRC-16 match over 128-byte payload (bytes 3-130)
3. SOH delimiter presence at frame start

Working Example: Real Flash Load Session

Below is a real successful flash loading session from a development board:

```
$ st-flash reset && sleep 1 && cd /home/ashwin/Desktop/Projects/FlashLoader && python3
    scripts/simple_flash.py -p /dev/ttyUSB0 -f App1/build/Simple_BootLoader.bin
st-flash 1.8.0
2026-02-01T11:13:36 INFO common.c: STM32F1xx_MD: 20 KiB SRAM, 64 KiB flash in at least 1 KiB pages.
2026-02-01T11:13:36 INFO common.c: NRST is not connected --> using software reset via AIRCR
2026-02-01T11:13:36 INFO common.c: Go to Thumb mode
Firmware size: 4432 bytes
Connected to /dev/ttyUSB0 at 115200 baud
Waiting for bootloader...
Bootloader detected.
Sending START command...
START acknowledged.
Starting transfer (554 chunks = 4432 bytes)...
Chunk     1/554 (    0 bytes): OK
Chunk   129/554 ( 1024 bytes): OK
Chunk   257/554 ( 2048 bytes): OK
Chunk   385/554 ( 3072 bytes): OK
Chunk   513/554 ( 4096 bytes): OK
Chunk   554/554 ( 4424 bytes): OK

```

```
All 554 chunks transferred successfully!
Sending END command...
Firmware transfer complete!
Bootloader will now jump to application at 0x08002000
Serial port closed.
Flash loading successful.
```

Analysis:

- Device detection: STM32F1xx with 20 KiB SRAM and 64 KiB flash
- UART connection established at 115200 baud
- Bootloader heartbeat received ("Bootloader detected")
- Protocol handshake successful (START acknowledged)
- Firmware size: 4432 bytes (fits in 56 KB application space)
- Transfer chunked into 554 blocks with progress indicators at every 128 blocks
- All chunks acknowledged (ACK received for each frame)
- Final confirmation: Application jump initiated to 0x08002000

7 ESP32 WiFi-Based Flash Loader

In addition to the Python command-line tool, this flash loader system includes an ESP32-based WiFi flash loader that provides a web interface for remotely flashing STM32 firmware over UART. This component eliminates the need for a PC with a serial connection, enabling wireless firmware updates in embedded systems.

7.1 Overview and Architecture

The ESP32 flash loader acts as a bridge between a WiFi-enabled client (web browser) and the STM32 bootloader via UART. It implements:

- **WiFi Access Point or Station Mode:** Creates its own WiFi network or connects to existing WiFi
- **Async Web Server:** Provides HTML interface for firmware upload and status monitoring
- **Intel HEX Parser:** Converts .hex files to binary format automatically
- **Simple Bootloader Protocol:** Communicates with STM32 using 8-byte chunk protocol
- **Non-Blocking Operation:** Uses FreeRTOS tasks to prevent web server blocking during flash

7.2 Hardware Configuration

7.2.1 Required Components

- ESP32 development board (any variant with WiFi)
- STM32F103C8 target board with bootloader
- 3 jumper wires for UART connection
- USB power supply for ESP32 (5V)

7.2.2 Wiring Connections

```
ESP32 Pin     | STM32 Pin     | Function
------------- | ------------- | --------
GPIO17 (TX2)  | PA10 (RX)     | ESP32 transmits to STM32
GPIO16 (RX2)  | PA9 (TX)      | ESP32 receives from STM32
GND           | GND           | Common ground
```

Important: The ESP32 uses UART2 (HardwareSerial port 2) for STM32 communication, leaving UART0 available for USB debugging. Ensure proper voltage levels—both devices operate at 3.3V logic levels.

7.3 Software Architecture

7.3.1 Key Components

1. WiFi Configuration

The ESP32 supports two WiFi modes:

- **Access Point (AP) Mode (default):** Creates WiFi network "ESP32_FlashLoader" with password "flashloader123"
- **Station (STA) Mode:** Connects to existing WiFi network (configure SSID/password in code)

```
// Configuration in main.cpp
#define WIFI_MODE_AP true  // Set to false for Station mode
const char* AP_SSID = "ESP32_FlashLoader";
const char* AP_PASSWORD = "flashloader123";
```

In AP mode, the ESP32 is accessible at **http://192.168.4.1**. In STA mode, it receives an IP from the router via DHCP.

2. Async Web Server

The flash loader uses ESPAsyncWebServer for non-blocking HTTP handling. Key endpoints:

- **GET /:** Serves index.html from LittleFS filesystem
- **GET /api/status:** Returns JSON with current flash state, progress percentage, and status message
- **POST /api/upload:** Receives firmware file (.hex or .bin), stores in RAM, and initiates flash process
- **POST /api/reset:** Resets flash state to idle

3. Intel HEX Parser

The ESP32 includes a hex-to-binary converter that processes Intel HEX format files:

```
bool hexToBinary(const uint8_t* hexData, size_t hexSize,
                 uint8_t* binBuffer, size_t* binSize)
```

This function:

- Parses Intel HEX records (data, extended linear address, EOF)
- Handles 32-bit addressing via extended linear address records (0x04)
- Converts ASCII hex characters to binary data
- Produces a flat binary image suitable for STM32 flash

This allows users to upload .hex files directly from development tools (STM32CubeIDE, Keil, etc.) without manual conversion.

4. Simple Bootloader Protocol Implementation

The ESP32 implements the simple 8-byte chunk protocol:

```
Protocol Sequence:
1. Wait for "BOOT" message from STM32 bootloader (5-second timeout)
2. Clear UART buffers (600ms delay + double clear to match Python tool)
3. Send START command ('S') and wait for ACK (3-second timeout)
4. For each 8-byte chunk:
   a. Calculate 8-bit checksum (sum of all bytes)
   b. Send DATA command ('D') + 8 bytes + checksum
   c. Wait for ACK (1s normal, 3s after page completion)
   d. Retry up to 8 times on NAK or timeout
5. Send END command ('E') and wait for final ACK
```

Key implementation details:

- **Timeout Management:** Extended timeouts (3 seconds) after every 128 chunks to allow STM32 flash page write completion
- **Retry Logic:** Up to 8 retries per chunk with exponential backoff on communication errors
- **Progress Tracking:** Prints status every 128 chunks (matching Python tool behavior)
- **Memory Management:** Firmware buffer allocated dynamically, freed after transfer completion

5. FreeRTOS Task Architecture

To prevent blocking the async web server during flash operations, the ESP32 uses FreeRTOS tasks:

```
void flashTask(void* pvParameters) {
  FlashJob* job = (FlashJob*)pvParameters;
  flashSTM32(job->data, job->size);
  free(job->data);
  free(job);
  vTaskDelete(NULL);
}
```

The flash task:

- Runs pinned to CPU core 1 (avoiding core 0 which handles WiFi/TCP stack)
- Receives firmware data pointer and size via FlashJob structure
- Updates global flash state and progress variables (volatile)
- Self-deletes upon completion (automatic cleanup)

The web server can continue serving status requests while flashing proceeds in the background.

7.4 Usage Instructions

7.4.1 Step 1: Flash ESP32 Firmware

Build and upload the ESP32 firmware using PlatformIO:

```
cd esp32_FOTA_app/esp32_stm32_FlashLoader
pio run --target upload
pio device monitor  # Optional: view serial debug output
```

The ESP32 will start in Access Point mode by default.

7.4.2 Step 2: Prepare Web Interface Files

Upload the web interface (index.html) to ESP32's LittleFS filesystem:

```
pio run --target uploadfs
```

This stores the HTML interface in flash memory, making it accessible to the web server.

7.4.3 Step 3: Connect to ESP32 WiFi

- On your laptop/smartphone, connect to WiFi network "ESP32_FlashLoader"
- Enter password: "flashloader123"
- Open web browser and navigate to **http://192.168.4.1**

7.4.4 Step 4: Upload and Flash Firmware

In the web interface:

1. Ensure STM32 is powered and connected via UART to ESP32
2. Reset STM32 to enter bootloader mode (bootloader sends "BOOT" messages)
3. Click "Choose File" and select your firmware (.hex or .bin file)
4. Click "Upload and Flash"
5. Monitor progress bar and status messages
6. Wait for "Firmware transfer complete!" confirmation

The STM32 bootloader will automatically jump to the new application after successful flash.

7.5 State Machine and Status Reporting

The ESP32 maintains a global flash state machine:

```
enum FlashState {
  IDLE,         // Ready to accept firmware
  IN_PROGRESS,   // Flashing active
  SUCCESS,       // Flash completed successfully
  FAILED        // Error occurred
};
```

Status is exposed via JSON API:

```
GET /api/status
{
  "state": "in_progress",
  "progress": 45,          // Percentage (0-100)
  "message": "Transferring firmware..."
}
```

The web interface polls this endpoint every 500ms to update the progress bar and status text.

7.6 Performance Characteristics

- **Upload Speed:** ~50-100 KB/s over WiFi (depends on signal strength)
- **Flash Speed:** ~1.2 KB/s to STM32 via UART at 115200 baud
- **Total Time (4.4 KB firmware):** ~8-10 seconds (including upload, conversion, and flash)
- **Memory Usage:** ~260 KB RAM for 256 KB firmware buffer (ESP32 has 520 KB total)
- **Concurrent Connections:** Supports multiple clients viewing status, but only one flash operation at a time

7.7 Advantages of ESP32 Flash Loader

- **Wireless Operation:** No physical connection to PC required
- **Remote Access:** Flash firmware from smartphones, tablets, or laptops
- **Embedded Integration:** Can be permanently installed in products for field updates
- **Hex File Support:** Direct upload from IDE output without conversion
- **User-Friendly:** Web interface eliminates need for command-line tools
- **Status Monitoring:** Real-time progress updates and error reporting
- **Portable Power:** USB power bank can power entire system (ESP32 + STM32)

7.8 Comparison: Python Tool vs ESP32 Flash Loader

Feature	Python Tool	ESP32 Flash Loader
Connection	USB-Serial cable	WiFi wireless
Interface	Command-line	Web browser GUI
Speed	Faster (direct USB)	Slower (WiFi + UART)
Hex Support	Requires pre-conversion	Built-in hex parser
Hardware	PC + USB cable	ESP32 + 3 wires
Portability	Requires PC/laptop	Works with phone/tablet
Field Deployment	Not practical	Ideal for embedded systems
Cost	Free (software only)	~$5-10 (ESP32 board)

7.9 Troubleshooting ESP32 Flash Loader

- **Cannot connect to ESP32 WiFi:** Verify AP mode is enabled; check SSID/password; reset ESP32
- **Web page not loading:** Ensure index.html is uploaded to LittleFS (pio run –target uploadfs)
- **Upload hangs at 100%:** Check ESP32 RAM availability; reduce firmware size or increase heap
- **"Bootloader not detected":** Verify UART wiring (TX↔RX crossover); check STM32 is in bootloader mode
- **Flash fails mid-transfer:** Check UART baud rate (115200); verify ground connection; reduce WiFi interference
- **Chunk failures/retries:** Increase UART timeout values; check for EMI on UART lines
- **ESP32 crashes during flash:** Insufficient RAM—reduce firmware buffer size or use external PSRAM
- **Serial monitor shows "BOOT" but flash fails:** Buffer clearing issue—increase delay after bootloader detection to 800ms

Troubleshooting Common Issues

- **No "BOOT" message:** Check UART wiring and baud rate (115200)
- **NAK on every frame:** CRC mismatch—verify CRC algorithm matches
- **Application does not start:** Verify application linked to 0x08002000; check bootloader MSP validation
- **Device hangs mid-transfer:** Flash write timeout—check clock configuration and flash wait states
- **"BOOT" but no mode response:** Try resending 'C' manually; timeout is 5 seconds