binho

APPLICATION NOTE

Programming STM32 microcontrollers over SPI with the Binho Pulsar

AN0003

Document information

Keywords	AN0003, Pulsar, Supernova, SPI, STM32, STM32H5, STM32H503, system bootloader, in-system programming, firmware update
Abstract	The system bootloader programmed into the ROM of STM32 microcontrollers offers an SPI interface on many families, allowing internal flash memory to be erased, programmed and verified with no debug probe attached. This application note describes that interface, the connections required between a Binho Pulsar USB host adapter and the target, and the procedure for programming a device. It gives particular attention to the framing rule that separates a working host implementation from one that corrupts data, and to a method of driving reset from spare chip selects when no general-purpose pins are free. Measured results are given for an STM32H503 on a NUCLEO-H503RB board.

Binho Inc.
42 Dore Street, Unit A, San Francisco, CA 94103

REVISION 1.0
30 AUGUST 2026

1 Introduction

The system bootloader programmed into the ROM of every STM32 microcontroller provides a means of programming internal flash memory over a serial interface, without a debug probe and without any user code present on the device. The interfaces offered by the bootloader vary by device and are listed per part number in AN2606.

SPI appears on that list for many families. It is the fastest of the three interfaces this series has covered when writing, and it is the one whose host implementation is easiest to get subtly wrong. The reason is a single design decision: the value the target sends when it has nothing to say is an ordinary data value, not a reserved one. A host that treats it as padding passes every identification command and then corrupts flash reads.

This note is the SPI counterpart to AN0001, which covers I3C, and AN0002, which covers I2C. The command set is nearly the same on all three. What differs is the framing, and on this bus the framing is what matters.

1.1 Scope

This document describes:

- the structure of the SPI system bootloader protocol, including the acknowledgement handshake and the rule that governs how a reply is read
- the connections required between a Binho Pulsar and an STM32 target, including a method of driving NRST and BOOT0 from spare chip selects
- the procedure for identifying, erasing, programming and verifying a device
- a method for confirming from outside the programming tool that the target is running the newly programmed image
- the accompanying reference utility, which drives either a Pulsar or a Supernova

It does not cover programming over UART, USB, CAN, I2C or I3C; option-byte programming; readout protection, which the ROM bootloader on the tested part does not offer over this interface; or production programming fixtures. For I3C see AN0001, and for I2C see AN0002.

1.2 Applicable devices

The SPI system bootloader is not specific to one family. The pins it uses differ per part and per SPI instance, and all of them are given in AN2606. They are not the pins a board routes to an Arduino-style SPI header, which is an application mapping of the same peripheral.

Table 1. Applicable devices

Family	Example part	Bootloader SPI interfaces	Basis
STM32H5	STM32H503	SPI1, SPI2, SPI3	Tested for this document
STM32H5	STM32H563	Per AN2606	AN2606, not tested
STM32G0, G4	—	Per AN2606	AN2606, not tested
STM32L4, L5	—	Per AN2606	AN2606, not tested
STM32U5	—	Per AN2606	AN2606, not tested

Only the STM32H503 was exercised on hardware, and only its SPI2 instance. The pins and the supported command list must be read from AN2606 for the part in hand before any of this document's commands are issued.

1.3 Related documents

Table 2. Related documents

Document	Title
AN2606	STM32 microcontroller system memory boot mode
AN4286	SPI protocol used in the STM32 bootloader
RM0492	STM32H503 reference manual
AN0001	Programming STM32 microcontrollers over I3C with the Binho Supernova
AN0002	Programming STM32 microcontrollers over I2C with the Binho Pulsar

2 System overview

2.1 Roles on the bus

The host adapter is the SPI controller and drives the clock and the chip select. The target, held in system bootloader mode, is an SPI target configured for **mode 0** (clock idle low, data sampled on the leading edge), MSB first, 8-bit words.

Unlike I2C there is no address, so exactly one target can be programmed per chip select, and unlike I3C there is no dynamic addressing. What SPI offers instead is that the host owns the clock completely, which removes clock stretching from the picture and, as Section 5.7 shows, moves the performance question somewhere else entirely.

2.2 Status reporting

Every command is acknowledged. The host clocks dummy bytes until the target answers `0x79` for ACK or `0x1F` for NACK.

Table 3. Values the target sends

Value	Meaning
`0x79`	ACK, the step was accepted
`0x1F`	NACK, the step was rejected
`0xA5`	Busy. The SPI peripheral's underrun pattern, sent whenever nothing is queued
`0x5A`	Sync, sent once when the session opens

The busy value is the important row, and Section 2.3 explains why.

2.3 The busy value is also a data value

`0xA5` is not a status the bootloader chooses to send. It is the underrun pattern the SPI peripheral emits whenever the target has nothing loaded in its transmit register. Nothing distinguishes it from a byte of flash that happens to hold `0xA5`.

This rules out the implementation most readers would write first. Skipping `0xA5` while looking for the start of a reply works on every identification command, because none of their replies begins with that value. It then corrupts any flash read whose first byte is `0xA5`, and never returns at all on a page that is entirely `0xA5`. Both failures were reproduced deliberately during the preparation of this document.

Section 5.6 gives the framing that works. It is positional: the host stops clocking at the acknowledgement, lets the target load, and then reads exactly the number of bytes it expects, filtering nothing.

> **IMPORTANT** A host that searches for the start of a reply will pass every test performed with identification commands and still corrupt firmware reads. Verify a read implementation against data that contains `0xA5`, not against Get and Get ID.

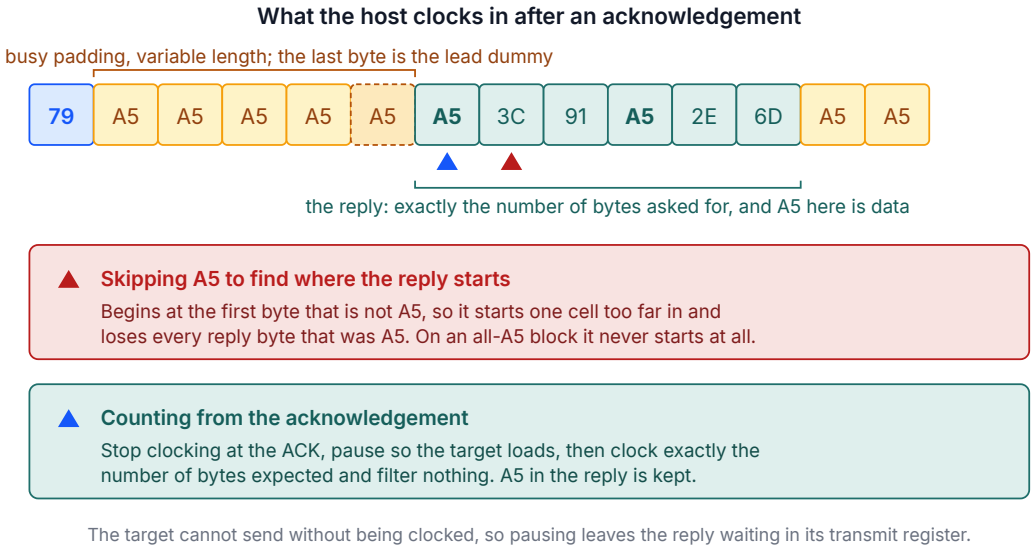

Figure 1. The byte stream a host clocks in after an acknowledgement. Searching for the first byte that is not the busy value loses reply bytes that happen to hold it; counting from the acknowledgement does not.

2.4 Host adapter options

Both configurations below use the same utility and the same procedure.

Table 4. Host adapter configurations

Adapter	Status
Binho Pulsar	Verified. Used for the timing measurements of Section 9
Binho Supernova	Verified. Same cycles and the same hostile-image test

3 Hardware setup

3.1 Required equipment

Table 5. Required equipment

Item	Used here
Host adapter	Binho Pulsar, hardware revision C, firmware 4.4.0
Target board	NUCLEO-H503RB
Wiring	Seven wires: four for the bus, one ground, two for reset control
Host	Python 3.12

3.2 Connections

The STM32H503 offers three SPI instances to its bootloader and no alternate pin mappings for any of them. SPI2 was used here because its four pins are all on port B.

Table 6. SPI bootloader pins on the STM32H503

Instance	MOSI	MISO	SCK	NSS
SPI1	`PA7`	`PA0`	`PA8`	`PB8`
SPI2	`PB1`	`PB14`	`PB10`	`PB12`
SPI3	`PC12`	`PC11`	`PC10`	`PD2`

Table 7. Connections between the Pulsar and the target

Pulsar	Target	Required
SCK	`PB10`	Yes
MOSI	`PB1`	Yes
MISO	`PB14`	Yes
CS 0	`PB12` (NSS)	Yes
GND	GND	Yes
CS 1	`NRST`	Optional, for automatic bootloader entry
CS 2	`BOOT0`	Optional, for automatic bootloader entry

> **IMPORTANT** These are not the pins a board routes to an Arduino-style SPI header. On the NUCLEO-H503RB that header carries SPI1 on `PA5`, `PA6`, `PA7` and `PC9`, which is the application mapping of the peripheral and not the one the ROM configures. Only `PA7` is common to both. Confirm against AN2606 for the part in hand.

3.3 Driving reset from chip selects

NRST and BOOT0 are shown above on chip selects rather than on general-purpose pins. This is worth describing because it relies on a property of chip selects that is rarely used this way, and because it frees the GPIO pins for other work.

No call in either adapter package sets a chip select to a level. What chip selects do offer is polarity: an active-low select idles high and is driven low for exactly as long as a transfer lasts. Two consequences follow:

- Selecting the line wired to NRST and clocking a slow transfer produces a **reset pulse of a known width**. The utility clocks 200 bytes at 10 kHz, which holds NRST low for about 160 ms.
- The line wired to BOOT0 **sits high while it is not the selected one**, which is the level that puts the part into the bootloader.

The same trick cannot select the application, because that needs BOOT0 held low while NRST is pulsed and only one line can be selected at a time. The bootloader's Go command covers that case, and the utility falls back to it automatically. Section 6.5 gives the practical consequence.

Where general-purpose pins are available instead, `--reset-via gpio` uses them and behaves exactly as in AN0001 and AN0002.

3.4 Bus considerations

SPI needs no pull-ups. The four lines are configured push-pull with no pull on the target side, which means **NSS must be actively driven and never left floating**.

> **CAUTION** Set the bus voltage to match the target before connecting. The utility sets 3.3 V, which suits a NUCLEO-H503RB. Driving 3.3 V into a target running at a lower supply can damage it.

4 Software setup

4.1 Installing the host package

The utility talks to the adapter through its vendor package. Install the one matching the adapter in use:

```
pip install binhopulsar        # for a Pulsar
pip install binhosupernova     # for a Supernova
```

Versions used for this document: `binhopulsar` 1.2.0, Python 3.12. No other package is required; the utility parses Intel HEX itself.

4.2 Obtaining the reference utility

`stm32_flash.py` is supplied in the assets archive that accompanies this document:

```
https://cdn.binho.io/application-notes/AN0003/AN0003-assets.zip
```

The same files, together with the Markdown source of this document, are published at `https://github.com/binhollc/application-notes`. Section 12 lists the archive contents.

The archive expands to an `assets` directory holding the utility, and a `firmware` subdirectory holding the verification images of Section 7. Every command in this document is shown as it would be run from inside that `assets` directory.

It is the same file supplied with AN0001 and AN0002, so one copy covers all three buses.

> **IMPORTANT** An adapter whose SPI interface is already initialised answers `spiControllerInit` with `INTERFACE_ALREADY_INITIALIZED` and **discards the settings passed with it**. The clock rate, mode and chip select then stay as they were, and every transfer goes out misconfigured while the call itself looks successful. A host must re-apply the same settings through `spiControllerSetParameters` whenever init reports anything other than success. The reference utility does this. Written without it, the symptom is a bus that reads back a constant and looks exactly like a wiring fault.

5 Bootloader protocol

5.1 Entering system bootloader mode

`BOOT0` is held high while `NRST` is pulsed low and released. Section 3.3 describes how the utility does this from chip selects.

5.2 Session start

The host clocks the sync byte `0x5A`. The target replies with a sync byte of its own and then an acknowledgement. There is no further handshake.

5.3 Command framing

Unlike the other two buses, the sync byte prefixes **every** command, not only the first:

```
host    5A <op> <~op>
target  ... 79
host    79
```

The target discards anything that is not `0x5A` while it waits for a command, so clocking dummy bytes between commands is harmless.

Commands taking an address send four bytes big-endian followed by a one-byte XOR of those four, and take another acknowledgement before their payload.

5.4 The host must acknowledge the acknowledgement

After sending its ACK the target **waits for the host to send `0x79` back** before it continues. A host that reads the acknowledgement and moves on leaves the target blocked, and nothing on the bus indicates why: the session stops responding.

This is the single most likely reason for an implementation to hang on first contact.

5.5 One dummy byte leads each reply

The first byte clocked out after an acknowledgement is the stale busy pattern, because the target's shift register still holds it when the first reply byte is queued behind. AN4286 states this as sending a dummy byte before a read.

It happens **once per reply, not once per transfer**. Requesting another dummy partway through a reply drops a real byte, which during development shifted the reported command list by one in each direction before the cause was identified.

5.6 Reading a reply

Combining Sections 2.3 and 5.5, the framing that works is:

1. Clock until the acknowledgement appears, and **stop there**.
2. Send `0x79` to release the target.
3. Pause briefly so the target loads its first reply byte.
4. Clock exactly the number of bytes expected, discarding the one leading dummy and filtering nothing else.

Step 3 is safe because the target cannot send without being clocked: pausing leaves the reply waiting in its transmit register. Pausing partway through a reply is equally safe, because the target queues its next byte and waits rather than padding.

What is not safe is clocking past the acknowledgement, because the reply begins immediately after it with no delimiter, and the busy value cannot be told apart from data.

5.7 Supported commands

Table 8. Commands supported over SPI on the STM32H503

Opcode	Command
`0x00`	Get
`0x01`	Get Version
`0x02`	Get ID
`0x11`	Read Memory
`0x21`	Go
`0x31`	Write Memory
`0x44`	Extended Erase
`0x50`	Special Command
`0x63`	Write Protect
`0x73`	Write Unprotect

Ten commands, and the list is identical to the one this part offers over I3C. It is five short of what the same part offers over I2C, which adds four no-stretch variants and Checksum. The no-stretch variants have no meaning here, since an SPI target has no way to stretch a clock it does not drive.

Readout Protect (`0x82`) and Readout Unprotect (`0x92`) are absent, as they are on the other two buses.

5.8 Erase

Extended Erase takes a page count followed by a list of page numbers, or a special value. Mass erase is `0xFFFF` with no page list.

For a page list the count field carries **the number of pages**, as on I3C, and not the number of pages minus one as on I2C. A flash page on this part is 8 KB.

6 Programming procedure

6.1 Identifying the target

```
python stm32_flash.py --bus spi --reset-via cs info

  Adapter: pulsar  fw 4.4.0
  Bus: SPI
  Resetting target into bootloader (NRST=CS1, BOOT0=CS2)
  SPI: 1000 kHz, mode 0, chip select 0
  Bootloader session established

  Protocol version : 2.0
  Device ID        : 0x474  (STM32H503)
  Commands         : 10
      0x00  Get
      ...
      0x73  Write Unprotect
```

The ten supported commands are listed in full between those lines; they are the ones given in Table 8. The adapter's serial number is printed on the first line and is omitted here.

This is the recommended first step on any new board, as it confirms wiring, bus configuration and bootloader entry before any destructive operation is attempted.

6.2 Reading the existing contents

```
python stm32_flash.py --bus spi --reset-via cs read backup.bin --address 0x08000000 --
length 0x20000
```

6.3 Programming an image

The `flash` command performs the complete sequence: erase, program, read back and compare, then start the application.

```
python stm32_flash.py --bus spi --reset-via cs flash firmware/an0003_image_a.bin --address
0x08000000
```

An Intel HEX file carries its own addresses, so the option is omitted. `--erase pages` erases only the pages the image covers, which is faster for a small image:

```
python stm32_flash.py --bus spi --reset-via cs flash firmware/an0003_image_a.bin --address 0x08000000 --erase pages
```

`--spi-speed` sets the clock, and Section 9.3 explains why raising it buys less than expected. Session options may be given before or after the subcommand.

6.4 Verification

Verification is performed by default. The programmed range is read back over the same interface and compared byte for byte with the source image. It is disabled with `--no-verify`.

Given Section 2.3, verification carries more weight on this bus than on the others: it is the step that would catch a read framing error before it reached a customer.

6.5 Starting the application

Where NRST and BOOT0 are on chip selects, BOOT0 cannot be driven low, so the utility issues the Go command instead of a reset. The output reports this as `Jumping to 0x08000000`.

One consequence follows from Section 3.3 and is worth planning for: while the adapter is connected and idle, the line wired to BOOT0 sits high, so a power cycle or an external reset puts the target back into the bootloader rather than into the application. Disconnect that line, or use `--reset-via gpio`, when the board is expected to start on its own.

7 Confirming the update on the target

The `flash` command reads the programmed range back and compares it byte for byte, which proves that the contents of flash match the image supplied. It does not prove that the device is executing the new firmware.

Two small firmware images are supplied with this document to close that gap. They are identical except for the letter they report and the rate at which they blink the user LED.

7.1 The verification images

Table 9. Verification images supplied with this document

Image	File	Reports	LED
A	`firmware/an0003_image_a.bin`	`IMAGE A`	1 Hz
B	`firmware/an0003_image_b.bin`	`IMAGE B`	5 Hz

The images are 864 and 860 bytes and are linked to run from `0x08000000`. Source, a linker script and a makefile are supplied alongside them.

Neither image uses any of the SPI2 pins, so they can be loaded and observed without moving any wiring.

7.2 Procedure

Open a serial terminal on the ST-LINK virtual COM port at 115200 8N1, then program Image A:

```
python stm32_flash.py --bus spi --reset-via cs flash firmware/an0003_image_a.bin --address
0x08000000
```

The terminal shows the banner and then one heartbeat line per second:

```
========================================
  Binho  |  AN0003
  Programming STM32 microcontrollers
  over SPI with the Binho Pulsar
========================================
  Running: IMAGE A
  Kernel clock: 32 MHz
  LED blink: 1 Hz
========================================
  IMAGE A  alive, 1 s
  IMAGE A  alive, 2 s
```

Program Image B and the reported letter changes and the LED blinks five times faster.

The transition from `IMAGE A` to `IMAGE B` is the end-to-end result: the image was erased, programmed and verified over SPI, and the device is running what was programmed.

> **IMPORTANT** Programming these images erases the application already on the device. Archive the existing contents first, as described in Section 6.2.

> **NOTE** The supplied binaries are built for the NUCLEO-H503RB. They drive `PA4` as USART3_TX and `PA5` as the user LED. On any other board the serial output will not appear. The source is supplied, so retarget it by changing the pin and peripheral definitions at the top of `firmware/main.c` and rebuilding.

8 Reference utility

8.1 Commands

Table 10. Utility commands

Command	Action
`list`	List connected Binho adapters
`info`	Identify the target and report its capabilities
`flash`	Erase, program, verify and run an image
`read`	Read memory into a file
`erase`	Mass-erase user flash
`reset`	Reset the target via GPIO

8.2 Options

Session options may be given either before or after the subcommand. Only those relevant to SPI are listed; AN0001 and AN0002 list the I3C and I2C options.

Table 11. SPI session options

Option	Default	Meaning
`--bus`	auto	`spi`, `i2c` or `i3c`
`--spi-speed`	1000000	SPI clock in Hz, 10000 to 50000000
`--spi-mode`	0	SPI mode
`--spi-cs`	0	Chip select carrying NSS
`--reset-via`	`gpio`	Drive NRST and BOOT0 from GPIO pins or from spare chip selects
`--nrst-cs`	1	Chip select wired to `NRST`, with `--reset-via cs`
`--boot0-cs`	2	Chip select wired to `BOOT0`, with `--reset-via cs`
`--manual`	off	Prompt for bootloader entry instead of driving it
`-v`, `-q`	off	Log every transfer, or suppress progress output

Table 12. Per-command options

Option	Applies to	Meaning
`--address`	`flash`, `read`	Base address for a raw binary
`--length`	`read`	Number of bytes to read
`--erase`	`flash`	`mass`, `pages` or `none`
`--page-size`	`flash`	Flash page size for `--erase pages`, default 8192
`--erase-timeout`	`flash`, `erase`	Seconds to allow for an erase
`--chunk-size`	`flash`, `read`	Bytes per transfer. Default 256, the most the bootloader accepts
`--no-verify`	`flash`	Skip the read-back comparison
`--no-run`	`flash`	Leave the device in the bootloader
`--run`	`info`, `read`, `erase`	Start the application afterwards

9 Measured results

9.1 Test configuration

Table 13. Test configuration

Item	Value
Target	STM32H503RB, NUCLEO-H503RB
Adapter	Binho Pulsar, hardware revision C, firmware 4.4.0
Host package	`binhopulsar` 1.2.0, Python 3.12
Utility	`stm32_flash.py`, as supplied in the assets archive
Bus	SPI2, mode 0, 3.3 V

9.2 Programming a 122,912-byte image

Table 14. Measured timing for a 122,912-byte image at 4 MHz

Operation	Time	Throughput
Mass erase	0.03 s	not applicable
Program	4.66 s	25.8 KiB/s
Verify	8.19 s	14.7 KiB/s

Writing is faster than reading here, which is the reverse of the other two buses and is a direct consequence of Section 5.6. Every acknowledgement in a write may be waited for in bursts, because nothing the host needs follows it. One acknowledgement in every read may not, because the reply begins immediately after it.

9.3 Read throughput against clock rate

Session setup, which covers enumeration, the reset pulse and the opening handshake, was 2.00 s and did not vary with clock rate.

Table 15. Measured 128 KiB read

Clock	Read phase	Throughput
1 MHz	10.31 s	12.4 KiB/s
4 MHz	8.92 s	14.3 KiB/s
8 MHz	8.91 s	14.4 KiB/s

All three reads returned byte-identical data.

Raising the clock beyond 4 MHz returns little, because the target's response latency after each step is counted in clock periods rather than in wall time, so a faster clock does not shorten the wait in byte terms. Read Memory also caps a block at 256 bytes, which bounds how much each command can carry.

9.4 Reliability

Table 16. Completed erase, program, verify and run cycles

Configuration	Cycles completed
Pulsar on SPI2, 1 MHz, reset from chip selects	6 of 6
Supernova on SPI2, 1 MHz, reset from chip selects	6 of 6

Cycles alternated between Image A and Image B, so each one erased, programmed and verified different contents from the cycle before it.

In addition, an image constructed so that every 256-byte block begins with `0xA5`, and one block consists entirely of `0xA5`, was programmed and verified byte for byte from both adapters. That image is the test that distinguishes a correct read implementation from one that filters padding; the implementation described in Section 5.6 passes it and the obvious alternative does not.

These figures describe one target and one host. They are evidence that the procedure is repeatable, not a reliability specification.

10 Troubleshooting

Table 17. Troubleshooting

Symptom	Probable cause and action
The session stops responding after the first acknowledgement	The host did not send `0x79` back. The target blocks until it does. See Section 5.4.
Identification works but flash reads are corrupt	The reply is being searched for rather than read positionally, and data containing `0xA5` is being taken for padding. See Sections 2.3 and 5.6.
A read never returns	The same cause, on a region that is entirely `0xA5`.
The command list is shifted by one	The leading dummy byte is being consumed once per transfer rather than once per reply. See Section 5.5.
Nothing responds at all	Wrong pins. The bootloader's SPI pins are not the board's Arduino SPI header. Confirm against AN2606.
Nothing responds, wiring confirmed	NSS is floating. The target configures it with no pull, so it must be driven.
Every transfer reads back the same constant	The adapter reported its SPI interface as already initialised and discarded the settings given to init, so the bus is running on whatever it was configured with before. Re-apply them with `spiControllerSetParameters`. See Section 4.2.
The target returns to the bootloader after a power cycle	Expected when BOOT0 is on a chip select, which idles high. See Section 6.5.
Raising the clock does not make it faster	Expected above about 4 MHz. The target's response latency is counted in clock periods, so a faster clock does not shorten it. See Section 9.3.
Garbled serial output from the verification images	The terminal baud rate does not match. The images use 115200 8N1.

11 Acronyms

Table 18. Acronyms

Term	Meaning
ACK	Acknowledge
CS	Chip select
HSI	High-speed internal oscillator
MISO	Controller in, target out
MOSI	Controller out, target in
NACK	Not acknowledged
NSS	Target select
ROM	Read-only memory
SCK	Serial clock
SPI	Serial Peripheral Interface
VCP	Virtual COM port

12 Note about the source code in the document

Example code shown in this document and supplied in the accompanying archive is provided for evaluation and reference. It is released under the MIT license, the text of which is included in the archive.

The archive `AN0003-assets.zip` is published at `https://cdn.binho.io/application-notes/AN0003/AN0003-assets.zip` and contains the following.

Table 19. Contents of the assets archive

Path	Description
`AN0003.md`	This document in Markdown
`assets/stm32_flash.py`	Reference utility, shared with AN0001 and AN0002
`assets/PROTOCOL.md`	Wire-level notes on the SPI bootloader protocol
`assets/test_hex_parser.py`	Tests for the Intel HEX parser
`assets/LICENSE`	MIT license
`assets/firmware/main.c`	Source for both verification images
`assets/firmware/link.ld`	Linker script
`assets/firmware/Makefile`	Builds both images
`assets/firmware/README.md`	Notes on the verification images
`assets/firmware/an0003_image_a.bin`	Prebuilt Image A
`assets/firmware/an0003_image_b.bin`	Prebuilt Image B

13 Revision history

Table 20. Revision history

Revision	Date	Description
1.0	30 August 2026	Initial release.

Legal information

Definitions

Draft

A document marked draft is under internal review and subject to formal approval. Binho Inc. makes no representation or warranty as to its accuracy or completeness and accepts no liability arising from its use.

Disclaimers

Limited warranty and liability

Information in this document is believed to be accurate and reliable. Binho Inc. makes no representation or warranty, express or implied, as to its accuracy or completeness, accepts no liability for the consequences of its use, and takes no responsibility for content originating outside Binho Inc.

In no event shall Binho Inc. be liable for any indirect, incidental, punitive, special or consequential damages, including without limitation lost profits, lost savings, business interruption, or costs of removing or replacing any product, whether or not based on tort, warranty, breach of contract or any other legal theory. Notwithstanding any damages a customer might incur, the aggregate and cumulative liability of Binho Inc. is limited in accordance with its terms and conditions of commercial sale.

Right to make changes

Binho Inc. reserves the right to change the information in this document, including specifications and product descriptions, at any time and without notice. This document supersedes all information supplied prior to its publication.

Suitability for use

Binho Inc. products are not designed, authorized or warranted for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure of a Binho Inc. product can reasonably be expected to result in personal injury, death, or severe property or environmental damage. Any such inclusion or use is at the customer's own risk and Binho Inc. accepts no liability for it.

Applications

Applications described here are illustrative only. Binho Inc. makes no warranty that they are suitable for the specified use without further testing or modification. Customers are responsible for the design and operation of their own applications and products, for appropriate design and operating safeguards, and for determining whether a Binho Inc. product is suitable and fit for their purpose. Binho Inc. accepts no liability for assistance with applications or customer product design.

Third-party products and specifications

This document refers to products, boards, specifications and documentation supplied by third parties, which remain the responsibility of their respective owners. Binho Inc. makes no warranty regarding them and their behavior may change without notice. Register maps, protocol details and device identifiers reproduced here were observed on the specific hardware and firmware versions stated in this document and must be confirmed against the vendor's own documentation for the reader's part.

Example code

Source code supplied with this document is provided as an example, for evaluation and reference. It is not qualified for production use, has not been developed under a functional safety or security process, and is licensed under the terms accompanying it in the distribution archive.

Terms and conditions of commercial sale

Binho Inc. products are sold subject to the general terms and conditions of commercial sale published by Binho Inc., unless otherwise agreed in a valid written individual agreement, in which case only the terms of that agreement apply.

Export control

This document and the items it describes may be subject to export control regulations. Export may require prior authorization from the competent authorities.

Security

All products may be subject to unidentified vulnerabilities or may support security standards with known limitations. The customer is responsible for the design and operation of its applications and products throughout their lifecycle to reduce the effect of such vulnerabilities, and for compliance with all legal, regulatory and security requirements concerning its products.

Trademarks

All referenced brands, product names, service names and trademarks are the property of their respective owners. **Binho** and the Binho logo are trademarks of Binho Inc. **Arm** and **Cortex** are registered trademarks of Arm Limited. **I3C** and **MIPI** are trademarks or registered trademarks of MIPI Alliance, Inc. **STM32**, **Nucleo** and **STMicroelectronics** are trademarks or registered trademarks of STMicroelectronics N.V.