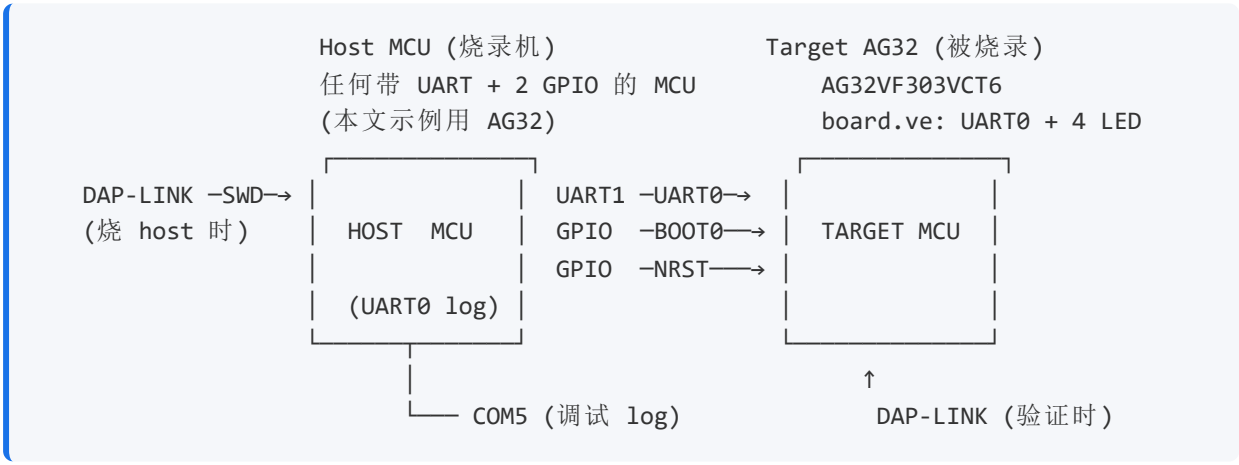
AG32 串口升级 — Host + Target mask ROM 协议

适用场景: 在没有 DAP-LINK / JTAG 工具时,通过任何带 **UART + 2 GPIO** 的 host MCU(本文以 AG32 为参考实现),使用 AG32 target 内置的 mask ROM bootloader 协议,把 code + logic 烧到 target flash。

常用于:
- **量产烧录**(脱离 JTAG,纯 UART 链路)
- **现场升级**(用现成的 host 板当烧录机)
- **双模式 logic 烧录**(压缩 vs 非压缩,host 上 button 切换)

---

1. 系统框图



5 根物理连线 (host ↔ target,任何带 UART 的 MCU 都可以是 host):

Host pin	信号	Target pin	备注
任意 GPIO (推荐 PIN_47)	NRST 控制	PIN_14 (NRST)	active-low, host 拉低再拉高触发 reset
任意 GPIO (推荐 PIN_48)	BOOT0 控制	PIN_94 (BOOT0)	高 = mask ROM, 低 = main flash
任意 UART TX (推荐 PIN_51)	TX	PIN_69 (UART0_RX)	19200 8N1 mask ROM auto-baud
任意 UART RX (推荐 PIN_52)	RX	PIN_68 (UART0_TX)	
GND	GND	GND	共地

注意:
- Target BOOT1 (PIN_37) **固定下拉到 GND**,不进其他 boot mode
- Host UART0 (PIN_68/69,AG32 默认) 是调试 log,通过 DAP-LINK 桥接到 PC COM5
- Target UART0 (PIN_68/69) **不接 DAP-LINK**,只接 Host UART1 (复用引脚编号但物理路径不同)
- **Host MCU 可以是任何带 UART + 2 GPIO 的单片机**(STM32 / ESP32 / NXP / 树莓派 Pico 等),协议是标准 mask ROM DFU 协议,与 host 芯片无关

---

2. 烧录产物:Target 的 3 段

Target 烧录产物 = **option bytes (128B) + logic (压缩/非压缩) + MCU code** 合并成一个 batch.bin。

段	目标地址	256KB 压缩	256KB 非压缩
Option bytes	0x81000000	128 B (实际 72 B 有效)	128 B (实际 72 B 有效)
Logic (含 decompressor)	0x80034000 (FLASH_DECOMP)	decompressor (2,720 B)	—
Logic (raw)	0x80034b00 (FLASH_LOGIC = FLASH_CONFIG_ADDR + 0xb00)	4,893 B	—
Logic (raw)	0x80027000 (FLASH_CONFIG_ADDR)	—	99,944 B
MCU code	0x80000000	~11 KB	~11 KB

逻辑大小对比 (实测,target_demo):
- 压缩: 4,893 B (`logic/board.bin`)
- 非压缩: 99,944 B (`logic/board.bin`)
- 阈值: `COMPRESSED_SIZE = 0xc000` (48 KB)
- `length < 0xc000` → 压缩 → 写 0x80034000 (decompressor) + 0x80034b00 (logic)
- `length >= 0xc000` → 非压缩 → 写 0x80027000 (logic)

核心公式: `FLASH_CONFIG_ADDR = 0x80000000 + flash_size - FLASH_CONFIG_SIZE`
- 256KB + 压缩 (48KB): 0x80034000
- 256KB + 非压缩 (100KB): 0x80027000
- 1MB + 压缩: 0x800F4000
- 1MB + 非压缩: 0x800E7000

---

3. 软件架构

3.1 Target (被烧录的 AG32)

普通 AG32 工程,跟正常开发一样:
- `board.ve` 配引脚 + 内部信号
- `logic/user_ip.v` 写自定义 CPLD (可选)
- `src/main.c` 写应用代码
- `platformio.ini` 配 `logic_compress = true|false` + `logic_address` + `ip_name`

烧录产物:
- `logic/board.bin` — CPLD bitstream
- `.pio/build/debug/target_demo.bin` — MCU code

3.2 Host (烧录机,任何 MCU)

关键点: Host 不需要是 AG32,可以是任何 MCU。但作为参考实现,本文档以 AG32 host 为例(对应 `E:\AI_AG32\ROMBOOT\romboot_host\`)。

协议层要求 (任何 host 都要实现):
- 1 个 UART (跟 target UART0 通信)
- 2 个 GPIO (NRST + BOOT0 控制)
- 1 个可选 GPIO (button 触发,可省略 — 上电自动烧)

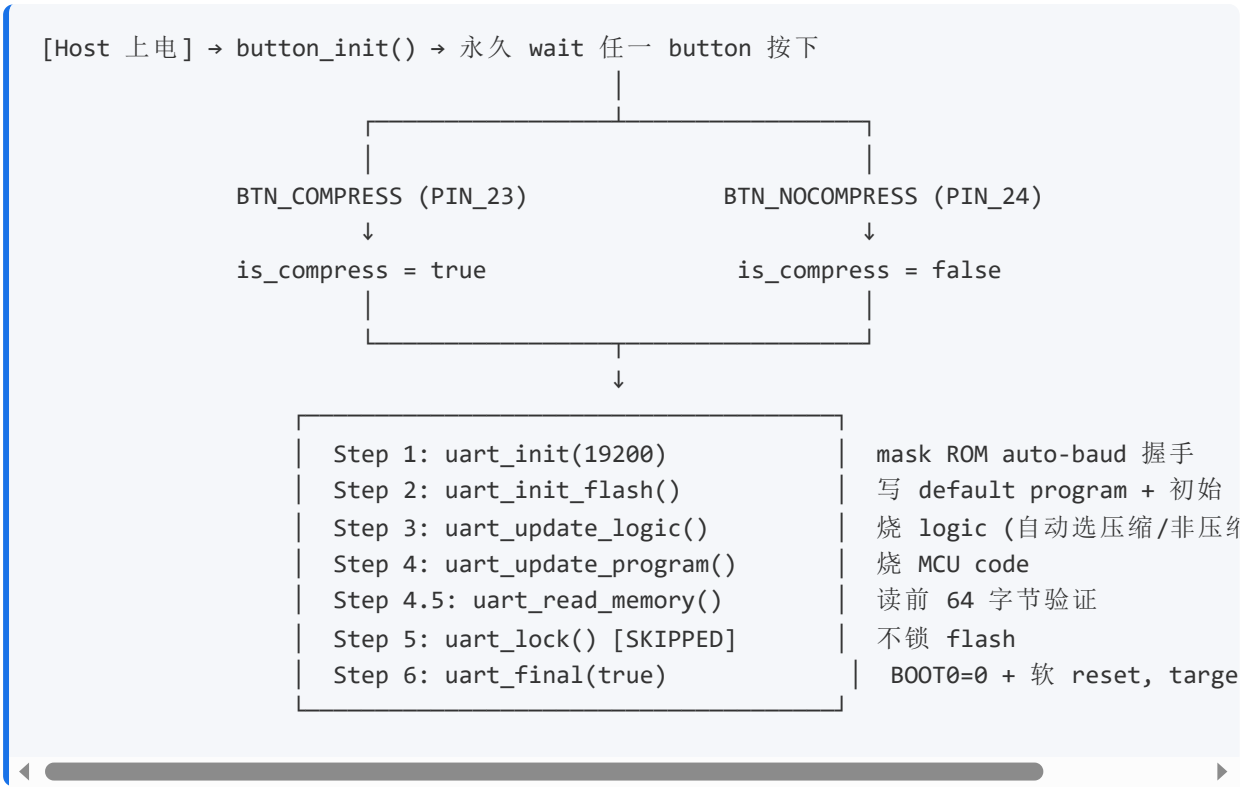
AG32 host 实现 (参考):
- `board.ve` 配 UART1 (PIN_51/52) + GPIO4_2/3 (NRST/BOOT0) + GPIO6_2/4 (button) + UART0 (log)
- `host_asf.asf` 配 WEAK_PULL_UP_RESISTOR for PIN_23/24 (button 内部上拉)
- `src/main.c` 7 步烧录流程
- `src/port.c` GPIO + UART + button 控制

- `src/uart_master.c` mask ROM 协议层
- 嵌入的 .inc 文件 (target 产物):
- `example.inc` (target code)
- `config_compress.inc` (target 压缩 logic)
- `config.inc` (target 非压缩 logic)

---

4. 烧录流程 (7 步)

Host 上电 + 按 button 之后,执行以下序列:



每个步骤详细字节序列见 §5。

---

5. 协议层 — mask ROM DFU 字节级规格

> 本节是协议完整规格,任何 host MCU 都可以实现。**不依赖 AG32**。

5.1 物理层

项	值
Baud rate	19200 (auto-baud 探测)
Data bits	8
Parity	None
Stop bits	1
Flow control	None (裸字节)
半双工	Host 主动,target 响应

5.2 帧格式

命令帧 (H → T) — 2 字节:

```
[cmd]  [~cmd]  // e.g. 0x31 0xCE for DFU_CMD_WM
```

地址帧 (H → T) — 5 字节 (大端):

```
[addr >> 24] [addr >> 16] [addr >> 8] [addr] [XOR]
// e.g. 0x80 0x03 0x40 0x00 0xC3 for 0x80034000
```

数据帧 (H → T) — 短数据 (length ≤ 256):

```
[len-1] [data...] [XOR_checksum]
```

- `len-1` 是 1 字节 (0-255, 0 表示 length=1, 255 表示 length=256)

数据帧 (H → T) — 长数据 (length > 256):

```
[(len-1) >> 8] [(len-1) & 0xFF] [data...] [XOR_checksum]
```

- `len-1` 是 2 字节大端 (e.g. 4893 → 0x13 0x14, 99944 → 0x01 0x86 0xA7)

XOR checksum = 整个数据(不含 checksum 本身)逐字节 XOR
Address XOR = 4 字节地址逐字节 XOR

5.3 响应码 (T → H)

字节	名称	含义
`0x79`	ACK	命令/地址/数据成功接收
`0x1F`	NACK	出错 (`UART_ERROR` 触发)
`0x76`	BUSY	target 还在处理 (需等待)

5.4 DFU 命令集 (源自 AG32 `uart_master.c`)

命令	字节	方向	用途
GET	`0x00`	H→T	读 DFU version + 支持的命令列表
RM	`0x11`	H→T	读 memory (≤256 B)
RE	`0x13`	H→T	读 memory (extended, >256 B)
GO	`0x21`	H→T	跳转到指定地址执行
WM	`0x31`	H→T	写 memory (≤256 B)
WE	`0x33`	H→T	写 memory (extended, >256 B)
EE	`0x44`	H→T	extended erase (按 page 列表)
RP	`0x82`	H→T	read protect (设 RDP)
UR	`0x92`	H→T	read unprotect (解 RDP)
CRC	`0xA1`	H→T	CRC 校验
RESET	`0xA2`	H→T	软复位 target
OO	`0xA3`	H→T	option bytes 擦除

> 这是 AG32 mask ROM 自定义扩展集,跟 STM32 ROM bootloader 协议**不通用**。如果换 AG32 target,这套命令才适用。

5.5 Option bytes 格式 (128 B @ 0x81000000)

偏移	字段	有效值
0-1	RDP word	`0x5AA5` (未锁) / `0x0000` (锁定)
12-13	FLASH_CONFIG_ADDR + ~FLASH_CONFIG_ADDR	非压缩 logic 地址 + 校验 (e.g. `0x80027000` + `0x7FFD8FFF`)
14-15	FLASH_LOGIC + ~FLASH_LOGIC	压缩 logic 地址 + 校验 (e.g. `0x80034b00` + `0x7FFCB4FF`)
16-17	FLASH_DECOMP + ~FLASH_DECOMP	Decompressor 地址 + 校验 (e.g. `0x80034000` + `0x7FFFCFFF`)
其他	reserved	`0xFFFFFFFF` 默认

只写前 18 个 uint32_t (72 B),后 56 B 保持 0xFFFFFFFF。

5.6 Auto-baud 握手

Target 上电时,如果 BOOT0=高 + mask ROM 有效,target 进入 bootloader,等待 `0x7F` 探测字节。Host:

```
// 1. 时序: BOOT0=高 + NRST=高 10ms (BOOT0 稳定)
// 2. NRST 拉低 10ms
// 3. NRST 拉高 10ms (target 复位后进入 mask ROM)
// 4. 发送 0x7F 探测
uart_tx(0x7F);
// 5. 等待 0x79 (ACK)
// 6. 发送 GET (0x00) 读 DFU version
// 7. 发送 GET 命令
// 8. 解析返回,获知支持 RE/WE/CRC
// 9. 发送 unprotect (UR, 0x92) if protected
// 10. 如果 target mask ROM 版本 < 0x20,先升级 DFU firmware 到 SRAM 0x20008000
```

为什么先 BOOT0=高再 NRST=低再拉高:
- 如果 BOOT0 在 NRST 释放前还浮空,target 可能 latch BOOT0=0,进 main flash (空) 模式而不是 mask ROM
- 强制 BOOT0=高持续 10ms,确保 target 复位时看到稳定的 BOOT0=1

5.7 DFU 升级机制 (Step 1 内部)

如果 target mask ROM 版本 < 0x20 (DFU_MIN_VERSION),host 需要先把新版 DFU firmware 烧到 target SRAM 0x20008000,然后跳过去执行,再走标准流程。

实测 (target AG32VF303VCT6 ROM bootloader):
- 写 5,296 B 到 0x20008000 (21 段 0x31 WM + 1 段 0x33 WE)
- GO 0x20008000
- 等 10ms
- 重新 init (再发 0x7F 探测,完整 GET 流程)

5.8 7 步详细字节序列

下面以 **BTN_COMPRESS (压缩 mode) + Step 3 烧 4,893 B 压缩 logic** 为例,列出关键字节。

Step 1: uart_init (19200)

```
[硬件时序]
H 拉 NRST 高, BOOT0 高           10ms
H 拉 NRST 低                      10ms
H 拉 NRST 高                      10ms

[DFU 握手]
H → 0x7F  (auto-baud 探测)
T → 0x79  (ACK)
H → 0x00 0xFF  (GET)
T → len+1 bytes (DFU version + supported commands)
H → 0x79  (ACK)
[可选] H → 0x92 0x6D  (UR unprotect, 若 RDP lock)
T → 0x79  (ACK)

[DFU 升级 (若 version < 0x20)]
H → 0x00 0xFF  (GET, 重入)
T → version byte
H → 0x79  (ACK)
H → 0x11 0xEE  (RM read 1 byte at 0x80000000)
T → 0x79  (ACK)
H → 0x80 0x00 0x00 0x00 0x80  (addr)
H → 0x00 0x00  (memory mode byte 0x00, only for old versions)
T → 0x79  (ACK)
H → 0x00  (1 byte read)
T → 0x79  (ACK)
T → 1 byte data
H → 0x21 0xDE  (GO 0x20008000)
T → 0x79  (ACK)
H → 0x80 0x00 0x08 0x00 0x88  (addr 0x20008000)
T → 0x79  (ACK)
[20 × WM (0x31) writes 256 B each to 0x20008000]
[1 × WM writes 176 B to 0x20009400]
[Verify 21 reads]
```

Step 2: uart_init_flash

```
[读 flash size]
H → 0x31 0xCE  (WM)
H → 0x40 0x00 0x10 0x2C 0x5C  (addr 0x4000102c)
H → 0x00 0x04 0x01 0x04 0x5A  (data 0x8001045a)
T → 0x79  (ACK)
H → 0x31 0xCE  (WM)
H → 0x40 0x00 0x10 0x14 0x44  (addr 0x40001014)
H → 0x00 0x00 0x00 0x34 0x34  (data 0x00000034)
T → 0x79  (ACK)
H → 0x11 0xEE  (RM)
H → 0x40 0x00 0x10 0x10 0x40  (addr 0x40001010)
H → 0x00 0x04  (read 4 bytes)
T → 0x79  (ACK)
T → 4 bytes
... (再 read 0x4000100c + 0x40001030 读 status 和 data)

[写 option bytes]
H → 0xA3 0x5C  (OO option erase)
T → 0x79  (ACK)
H → 0x31 0xCE  (WM)
H → 0x81 0x00 0x00 0x00 0x01  (addr 0x81000000)
H → 0x47 0x47 0xB7  ...  (72 bytes option bytes, 长度 71 = 0x47)
T → 0x79  (ACK)
H → 0x11 0xEE  (RM verify)
H → 0x81 0x00 0x00 0x00 0x01  (addr)
H → 0x47  (read 72 bytes)
T → 0x79  (ACK)
T → 72 bytes
H → compare

[写 default program (16 B)]
H → 0x31 0xCE  (WM)
H → 0x80 0x00 0x00 0x00 0x80  (addr 0x80000000)
H → 0x0F 0xB7 0x02 0x00 0x03 0x71 0x43 0x23 0xAA ... (16 bytes, 长度 15 = 0x0F)
T → 0x79  (ACK)
```

Step 3: uart_update_logic (压缩 mode, 4,893 B)

```
[读 flash size (同 Step 2 头部)]
...

[读当前 option bytes (72 B)]
H → 0x11 0xEE  (RM)
H → 0x81 0x00 0x00 0x00 0x01  (addr)
H → 0x47  (read 72 bytes)
T → 0x79  (ACK)
T → 72 bytes
H → check 12-13 (FLASH_CONFIG_ADDR match) and 14-17 (FLASH_LOGIC+FLASH_DECOMP match)
```

```
[因为当前是 UNCOMPRESSED, 不匹配 → 走 uart_write_options() 路径]
H → 0xA3 0x5C  (OO option erase)
T → 0x79  (ACK)
H → 0x31 0xCE  (WM)
H → 0x81 0x00 0x00 0x00 0x01  (addr)
H → 72 bytes (option bytes 含 FLASH_LOGIC+FLASH_DECOMP, 长度 71 = 0x47)
T → 0x79  (ACK)
H → 0x11 0xEE  (RM verify)
H → 0x81 0x00 0x00 0x00 0x01  (addr)
H → 0x47
T → 0x79  (ACK)
T → 72 bytes

[重写 default program (16 B) @ 0x80000000]
H → 0x31 0xCE  (WM)
H → 0x80 0x00 0x00 0x00 0x80
H → 16 bytes
T → 0x79

[写 decompressor @ 0x80034000 (FLASH_DECOMP, 2,720 B)]
H → 0x33 0xCC  (WE, extended write, 因为 2720 > 256)
H → 0x80 0x03 0x40 0x00 0xC3  (addr 0x80034000)
H → 0x0A 0x9F  (length-1 = 2719 = 0x0A9F)
H → 2720 bytes decompressor
T → 0x79  (ACK)
H → 0x13 0xEC  (RE verify)
H → 0x80 0x03 0x40 0x00 0xC3
H → 0x03 0xFF  (1023, read 1024 bytes)
T → 0x79
T → 1024 bytes
[再 RE 2 次, 读剩 672 bytes]

[写 compressed logic @ 0x80034b00 (FLASH_LOGIC, 4,893 B)]
H → 0x33 0xCC  (WE, extended)
H → 0x80 0x03 0x4B 0x00 0xCC  (addr 0x80034b00)
H → 0x13 0x14  (length-1 = 4892 = 0x1314)
H → 4096 bytes
T → 0x79
H → 0x33 0xCC  (WE, 残余 797 bytes)
H → 0x80 0x03 0x5B 0x00 0xD8  (addr 0x80035b00)
H → 0x03 0x1C  (length-1 = 796 = 0x031C)
H → 797 bytes
T → 0x79

[Verify logic @ 0x80034b00 (4,893 B)]
H → 0x13 0xEC  (RE)
H → 0x80 0x03 0x4B 0x00 0xCC
H → 0x03 0xFF  (read 1024)
T → 0x79
T → 1024 bytes
[再 RE 4 次 (3 × 1024 + 1 × 797)]
```

Step 3: uart_update_logic (非压缩 mode, 99,944 B)

```
[读 flash size + 读 option bytes 同上]

[因为当前是 COMPRESSED, 不匹配 → 走 uart_write_options() 路径 (如果切换)
 或因为已经是 UNCOMPRESSED 匹配 → 走 uart_erase_flash() 路径]

[erase 100 KB logic 区]
H → 0x44 0xBB  (EE extended erase)
H → 0x09 0x09  (10 pages - 1 = 9, 大端)
H → 10 × uint16 (page 编号, 含 0x8000 表示 block erase)
T → 0x79  (ACK)

[写 uncompressed logic @ 0x80027000 (FLASH_CONFIG_ADDR, 99,944 B)]
H → 0x33 0xCC  (WE, extended)
H → 0x80 0x02 0x70 0x00 0xF2  (addr 0x80027000)
H → 0x01 0x86 0xA7  (length-1 = 99943 = 0x186A7)
H → 4096 bytes
T → 0x79
H → 0x33 0xCC × 24 次 (24 × 4096 = 98,304 bytes)
H → 0x33 0xCC  (WE 残余 1640 bytes)
H → 0x80 0x03 0xF0 0x00 0xCF  (addr 0x8003f000)
H → 0x06 0x67  (length-1 = 1639)
H → 1640 bytes
T → 0x79

[Verify logic @ 0x80027000 (99,944 B)]
H → 0x13 0xEC × 98 次 (97 × 1024 + 1 × 616, 1024 大段)
T → 0x79
```

Step 4: uart_update_program (11,136 B)

```
[erase 12 KB (3 sectors)]
H → 0x44 0xBB  (EE)
H → 0x00 0x02  (2 pages - 1 = 2, 大端)
H → 3 × uint16 (page 0, 1, 2)
T → 0x79  (ACK)

[写 code @ 0x80000000 (11,136 B)]
H → 0x33 0xCC  (WE, extended)
H → 0x80 0x00 0x00 0x00 0x80
H → 0x2B 0x7F  (length-1 = 11135 = 0x2B7F)
H → 4096 bytes
T → 0x79
H → 0x33 0xCC  (WE 4096 bytes @ 0x80001000)
H → 0x80 0x00 0x10 0x00 0x90
H → 4096 bytes
T → 0x79
H → 0x33 0xCC  (WE 残余 2944 bytes @ 0x80002000)
H → 0x80 0x00 0x20 0x00 0xA0
```

```
H → 0x0B 0x7F  (length-1 = 2943)
H → 2944 bytes
T → 0x79

[Verify code @ 0x80000000 (11,136 B)]
H → 0x13 0xEC × 11 次 (10 × 1024 + 1 × 896)
T → 0x79
```

Step 4.5: verify 64 字节

```
H → 0x11 0xEE  (RM)
H → 0x80 0x00 0x00 0x00 0x80
H → 0x3F  (read 64 bytes)
T → 0x79
T → 64 bytes
H → compare byte-by-byte
```

Step 6: uart_final(true)

```
[硬件]
H 拉 BOOT0 = 0
H 拉 NRST 低  1ms
H 拉 NRST 高  10ms
[UART 关闭]
```

---

6. 双模式 (压缩 vs 非压缩) 完整对比

维度	压缩	非压缩
Logic bin 大小	4,893 B (target_demo 实测)	99,944 B (target_demo 实测)
Decompressor 地址 (FLASH_DECOMP)	**0x80034000** (2,720 B 写一次)	N/A
Logic 地址 (FLASH_LOGIC = +0xb00)	**0x80034b00**	N/A
Logic 地址 (FLASH_CONFIG_ADDR)	N/A	**0x80027000**
Flash 占用	48 KB	100 KB
Decompressor 算法	**需要** (LZW 解压)	**不需要**
Option bytes 字段	offset 14-17 (FLASH_LOGIC + FLASH_DECOMP)	offset 12-13 (FLASH_CONFIG_ADDR)
启动时间	多 30-50ms (解压)	正常
工程配置	`logic_compress = true`	`logic_compress = false`
Host .inc 文件	`config_compress.inc` (4,893B)	`config.inc` (99,944B)
实测总耗时	**30 秒**	**137 秒**

切换方法 (target 工程):

1. `target_demo/platformio.ini` 改 `logic_compress = true|false`
2. `target_demo/platformio.ini` 改 `board_upload.logic_address = 0x80034000` (压缩) 或 `0x80027000` (非压缩)
3. 跑 PIO 7 步重新 build
4. 转 `.inc` (`bin2inc.py logic/board.bin target_demo_logic.bin`)
5. 复制到 host/src/ (按命名约定: 压 → `config_compress.inc` , 非压 → `config.inc`)
6. 改 host `main.c` 引用对应 inc (或编译时已自动选 — 看 main.c 怎么写)

Host 端 完全自动 — `uart_update_logic(length, buffer)` 根据 `length < COMPRESSED_SIZE` 判断:
- 传 4,893 B (config_compress[]) → 写 0x80034000 + decompressor
- 传 99,944 B (config[]) → 写 0x80027000

option bytes 切换条件 (`uart_master.c:624-625`):

```
if (logic_compress && (!compress_addr_match || non_compress_addr_valid) ||
    !logic_compress && !non_compress_addr_match) {
  ENSURE(uart_write_options());  // 擦整个 flash + 重写 option bytes
  ...
} else {
  ENSURE(uart_erase_flash(FLASH_CONFIG_ADDR, FLASH_CONFIG_SIZE));  // 只擦 logic 区
}
```

关键 bug 修复 (2026-09-11): Host `main.c` 必须用变量 `is_compress` 锁存 button 状态,**不能在 STEP 3 时重新读 button** (因为 7 秒后 button 早已松开):

```
// 错误 (原版):
if (button_compress_pressed()) {  // 7 秒后再读, button 已松开 → 走 else
  uart_update_logic(sizeof(config_compress), config_compress);
} else {
  uart_update_logic(sizeof(config), config);  // ← 错误路径
}

// 正确 (修复版):
bool is_compress = false;
while (1) {
  if (button_compress_pressed())  { is_compress = true;  break; }
  if (button_uncompress_pressed()){ is_compress = false; break; }
}
// 7 步里:
if (is_compress) uart_update_logic(sizeof(config_compress), config_compress);
else             uart_update_logic(sizeof(config), config);
```

---

7. 编译烧录步骤 (完整)

7.1 Target 工程 (target_demo)

按 PIO 7 步标准 build (详见 `自定义logic开发流程与下载方法.md`):

```
cd E:\AI_AG32\ROMBOOT\target_demo
pio run --target prelogic --environment debug
cd logic; quartus_sh -t af_quartus.tcl; cd ..
cd logic; af --batch -F my_run.tcl; cd ..
pio run -t buildprog --environment debug
```

7.2 bin → inc 转换

```
cd E:\AI_AG32\ROMBOOT\target_demo
python bin2inc.py .pio\build\debug\target_demo.bin target_demo_code.inc
python bin2inc.py logic\board.bin target_demo_logic_compress.inc   # 压缩
# 或
python bin2inc.py logic\board.bin target_demo_logic_uncompress.inc # 非压缩
```

7.3 复制到 Host 工程

```
Copy-Item target_demo_code.inc            E:\AI_AG32\ROMBOOT\romboot_host\src\example.
Copy-Item target_demo_logic_compress.inc   E:\AI_AG32\ROMBOOT\romboot_host\src\config_
Copy-Item target_demo_logic_uncompress.inc E:\AI_AG32\ROMBOOT\romboot_host\src\config.
```

7.4 Host 工程编译烧录 (DAP-LINK 接到 Host)

```
cd E:\AI_AG32\ROMBOOT\romboot_host
pio run -t buildlogic --environment debug   # 编译 ASF + VE → bitstream
pio run -t logic      --environment debug   # 烧 bitstream (含 ASF 上下拉)
pio run -t upload     --environment debug   # 烧 Host code
```

或者: `pio run -t batch_upload` 一把全烧。

7.5 触发升级

DAP-LINK 在 Host,Host 启动后:
- 监听 COM5 (DAP-LINK 桥接) 看 host banner
- 按 **BTN_COMPRESS (PIN_23)** → 烧压缩
- 按 **BTN_NOCOMPRESS (PIN_24)** → 烧非压缩

---

8. Host Button 设计 (AG32 实现示例)

8.1 硬件

- 2 个 active-low button, 一端接 PIN_xx, 另一端接 GND
- PCB **不需要**外部上拉电阻 — 通过 ASF `WEAK_PULL_UP_RESISTOR` 启用芯片内部上拉 (AG32 SDK GPIO 没有软件上拉寄存器)

8.2 软件 (AG32 host)

board.ve (host 工程的 VE 配):

```
GPIO6_2 PIN_23   # BTN_COMPRESS  (bit 2)
GPIO6_4 PIN_24   # BTN_NOCOMPRESS (bit 4)
```

host_asf.asf:

```
set_instance_assignment -name WEAK_PULL_UP_RESISTOR ON -to PIN_23
set_instance_assignment -name WEAK_PULL_UP_RESISTOR ON -to PIN_24
```

platformio.ini (VE-only 路径):

```
[setup_logic]
logic_ve     = board.ve
logic_device = AGRV2KL100
design.asf   = ./host_asf.asf
# 不设 ip_name / logic_dir → 走 VE-only 路径
```

port.c button 模块:

```
void button_init(void) {
  SYS_EnableAPBClock(APB_MASK_GPIO6);
  GPIO_SetInput(BTN_GPIO_NAME, BTN_COMPRESS | BTN_NOCOMPRESS);
  GPIO_SetSoftwareMode(BTN_GPIO_NAME, BTN_COMPRESS | BTN_NOCOMPRESS);
}
bool button_compress_pressed(void) {
  return (GPIO_GetValue(BTN_GPIO_NAME, BTN_COMPRESS) == 0);
}
```

---

9. 验证步骤

9.1 Host 侧

1. 烧完 host 后, 监听 COM5 看 banner
2. 按 button, 看 host 7 步输出:
   `[STEP 1/7] uart_init (19200 8N1)... OK [STEP 2/7] uart_init_flash ... OK [STEP 3/7] uart_update_logic ... OK [STEP 4/7] uart_update_program ... OK [STEP 4.5/7] verifying ... OK [STEP 6/7] uart_final ... OK [HOST] BURN COMPLETE in X ms`
3. host 进入 `while(1) { nop; }` 死循环 (idling)

9.2 Target 侧 (切换 DAP-LINK)

1. DAP-LINK 接到 target

2. 跑 openocd 读 option bytes:
   `powershell & "C:\Users\wangy\.platformio\packages\tool-agrv_openocd\bin\openocd.exe" ` -s "C:\Users\wangy\.platformio\platforms\AgRV\etc" ` -c "set ADAPTER cmsis-dap" -c "set ADAPTER_SPEED 10000" ` -f agrv2k.cfg -c "init; halt; sleep 100; mdb 0x81000000 16" -c "shutdown"`
3. 确认 option bytes header 正确:
   - 压缩: offset 12-13 应该是 ~0x80027000 (uncompressed pointer), offset 14-15 = 0x80034b00, offset 16-17 = 0x80034000
   - 非压缩: offset 12-13 = 0x80027000 + ~0x80027000
4. 读 code 区前 64 字节,跟 `target_demo.bin` 头对比
5. 跑 reset + resume, 听 target UART0 输出 / 看 4 LED 流水

---

10. 实测性能 (3 次测试,target_demo + AG32VF303VCT6)

Step	描述	压缩 30s	非压缩 137s
1	uart_init (含 DFU 升级 5,296 B)	7,309 ms	7,368 ms
2	uart_init_flash (写 option bytes 72B + default program 16B)	304 ms	304 ms
3	uart_update_logic (核心)	**9,264 ms**	**116,577 ms**
4	uart_update_program (3 sectors erase + 11KB write)	13,031 ms	13,025 ms
4.5	verify 64 B	<100 ms	<100 ms
总		**30,025 ms**	**137,299 ms**

耗时差异原因:
- COMPRESSED: Step 3 用 9 秒 (写 2,720 B decompressor + 4,893 B logic)
- UNCOMPRESSED: Step 3 用 116 秒 (擦 100KB + 写 99,944 B + verify 99,944 B)

Boot 速度差异: COMPRESSED 启动多 30-50ms (LZW decompress),UNCOMPRESSED 启动直接跑。

---

11. 失败诊断 (Failure Diagnosis)

失败位置	症状	可能原因	修复
Step 1 握手	"Send cmd: 0x7f" 后无 "recv cmd: 0x79" (10 次重试都失败)	(1) BOOT0 没拉高 (2) 硬件不通 (3) baud 不匹配 (4) 5 根线接错	(1) 检查 BOOT0=1 timing (10ms 等待) (2) 检查 5 根线 (3) 默认 19200 baud (4) TX/RX/GND 物理检查
Step 1 升级 DFU	写 5,296 B 到 0x20008000 失败 (NACK)	(1) target 内存不够 (2) 协议不兼容	几乎不会失败,如果失败检查 target 型号
Step 1 升级后 re-init	"Send cmd: 0x7f" 在升级后失败	(1) JUMP_WAIT 10ms 不够 (2) 升级 firmware 损坏	(1) 增大 usleep(JUMP_WAIT * 1000) (2) 重新烧 host firmware
Step 2 写 option bytes	写 0x81000000 失败	(1) target 在 RDP 锁定状态 (2) 1F NACK 错误码	查 RDP word (offset 0),发 0x92 UR unprotect
Step 3 写 decompressor (压)	0x33 WE 0x80034000 NACK	(1) option bytes 切换不完整 (2) flash 还在擦	等几秒重试 (host 端自动重试)
Step 3 写 logic 0x80034b00 (压)	0x33 WE 0x80034b00 NACK	logic 地址跟 option bytes 不匹配	检查 uart_update_logic() 内部条件
Step 3 写 logic 0x80027000 (非压)	0x33 WE 0x80027000 NACK	(1) target 还在擦 (2) 地址算错	重试或检查 flash_size 计算
Step 3 verify	读回 byte 不匹配	(1) 写入出错 (2) flash 损坏	重新 erase + write
Step 4 写 code	0x33 WE 0x80000000 NACK	擦 sector 失败 (block 锁?)	重新 erase
Step 4.5 verify 64 B	头 64 字节不匹配	写入错误	重新跑 Step 4
Step 6 target 不跑	host 完成后 target 黑屏	(1) BOOT0 没拉低 (2) NRST 没 reset (3) option bytes 错	(1) 检查 uart_final 拉 BOOT0=0 (2) 检查 sys_nrst 拉 NRST 低 1ms 高 10ms (3) DAP-LINK 接 target 读 option bytes

失败位置	症状	可能原因	修复
target 跑但卡 PLL	PC 停在 0x800016CC (SYS_IsPllReady)	(1) OSC_CAL 丢失 (2) option bytes 里 logic address 错	**永远不要 Wipe Flash**!重跑 PIO `pio run -t batch_upload` (会自动 calibrate OSC)
Step 3 选项 byte 切换后不响应	target 不跑	option bytes 字段不匹配,boot ROM 找不到 logic	DAP-LINK 接 target, 跑 `mdb 0x81000000 16` 看 option bytes,确认 12-13 或 14-17 字段对

快速诊断流程:
1. 跑 Step 1 → 看是否收到 0x79
2. 跑 Step 2 → 看是否写 option bytes 成功
3. 跑 Step 3 → 看是否 decompressor + logic 写成功
4. 跑 Step 4 → 看是否 code 写成功
5. 跑 Step 6 → target 上电
6. 切 DAP-LINK → 读 option bytes + PC + 试 reset+resume

常用命令:

```
# 读 option bytes (128B)
& openocd ... -c "init; halt; mdb 0x81000000 16" -c "mdb 0x81000010 16" -c "mdb 0x8100

# 读 code 前 64 B
& openocd ... -c "init; halt; mdb 0x80000000 16" -c "mdb 0x80000010 16" -c "mdb 0x8000

# reset + resume
& openocd ... -f ocd_resume.cfg
```

12. 相关 skill

- `自定义logic开发流程与下载方法.md` — Target 工程的 7 步 PIO build
- `使用VE配置引脚-非自定义logic.md` — VE-only 路径 (Host 工程同款)
- `新建工程前必看-10个检查点.md` — 通用工程结构
- `flash分区和logic默认地址计算.md` — flash 布局公式
- `UART乱码时排查清单.md` — COM5 乱码排查