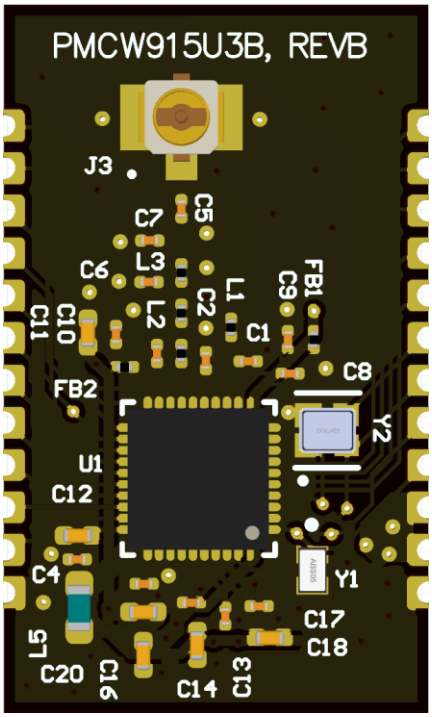
PMCW915U3B

915 MHz Wireless Sensor Gateway Module

User Manual

Firmware v2.1.0 · UART protocol 1.3.0 · Document issue 2.1 · May 2026

Power Micro Controls



Every byte sequence and console transcript in this manual was captured from a running module rather than composed by hand. Where behavior is version-specific or not yet final, it is called out explicitly.

Contents

1. What this module is

PMCW915U3B is a 915 MHz radio module that arrives with its firmware already loaded. You interact with it entirely over UART, and everything the module can do is reachable through the interfaces in this manual: identify itself, configure the radio, poll a sensor, retrieve stored data, persist settings and update firmware. There are two interfaces, one built for your host application and one built for a person at a terminal, and both are meant to be easy to drive.

The operating model is poll/response. Sensors never transmit on their own. Your host asks the module for something; the module performs one radio transaction and reports the result asynchronously. That gives you complete control over channel traffic, which is what makes dense deployments practical.

The module is stateless about device identity. It holds no list of paired sensors. Every request carries the target sensor's 16-character EUI inline. Enrollment, pairing, scheduling and de-duplication all live in your system, and the module simply executes one transaction at a time.

Three things to know up front

17. **One request at a time.** A second request while one is in flight returns `ERR_BUSY`.
18. **Answers arrive asynchronously.** A sensor request is acknowledged immediately, then the actual data arrives later as an unsolicited *indication*.
19. **Configuration is staged, then committed.** `CFG_SET` edits a RAM working copy; `CFG_COMMIT` validates the whole set, writes it to non-volatile memory, and applies it to the radio.

2. Before you start

You need:

Item	Detail
Power	3.3 V at `BAT+` (Header 2, pins 10 and 11) and `GND` (Header 1 pin 12, Header 2 pin 12)
Antenna	A 50 Ω 915 MHz antenna on `J3`. **Never transmit without a load.**
Serial connection	460800 baud, 8N1, no flow control, 3.3 V logic. Module I/O is not 5 V tolerant.
Sensor identities	The 16-character EUI of each sensor you intend to poll (§11.7)

Wiring for bench work. Connect a 3.3 V USB-to-UART adapter to the console port:

Module pin	Signal	Adapter
Header 1, pin 10	`COM0_TX`	adapter RX
Header 1, pin 11	`COM0_RX`	adapter TX
Header 1, pin 12	`GND`	adapter GND

Wiring for production. Connect your host processor to the machine port:

Module pin	Signal	Host MCU
Header 1, pin 8	`COM1_TX`	host RX
Header 1, pin 9	`COM1_RX`	host TX
Header 2, pin 1	`RESET`	host GPIO (open-drain), recommended

Both ports can be used simultaneously.

> If your module is mounted on a carrier board, the console port is often already routed to an on-board USB-serial bridge. In that case simply open the serial port that bridge presents; no wiring is needed. Confirm which port reaches the module by opening it at 460800, pressing Enter, and looking for the `gw>` prompt.

3. Two interfaces

The module has two UARTs. They speak different grammars for different audiences.

	COM1 — machine port	COM0 — console port
Header 1 pins	8 (TX) / 9 (RX)	10 (TX) / 11 (RX)
SoC pins	PA05 / PA06	PA07 / PA08
Grammar	Framed binary protocol only	Text console, plus binary (auto-detected)
Use it for	Production integration	Bench work, bring-up, field diagnostics
Stability	Version-frozen contract	Convenience grammar, may change between releases

3.1 How COM0 serves both grammars

The console port inspects the first byte of each new line: if it is `0xA5` (the binary start-of-frame marker) the traffic is parsed as a binary frame; anything else is treated as console text. This lets you drive the machine API from a PC without rewiring. Binary responses arrive interleaved with ASCII console output, including the local echo of typed characters and one-line summaries of every indication. Your parser must scan for the A5 5A marker and ignore the surrounding text.

> A real capture makes the point. A binary `SENSOR_STATUS` on COM0 returns the binary ACK, the binary indication, and the console's ASCII summary of the same event:

```
A5 5A 03 00 16 7E 16 22 32 92                      <- binary ACK
A5 5A 17 00 00 31 45 30 ... 7A 10 A0 FF 19 00 7C B2  <- binary STATUS_IND
0D 0A 5B 49 4E 44 5D 20 73 74 61 74 75 73 ...      <- ASCII: "[IND] status ..."
```

> **For production integration, use COM1.** It carries binary only, with no console chatter.

4. Quick start

Open the console port at 460800 8N1 and press Enter. You should see:

```
gw>
```

Ask the module to identify itself:

```
gw> info
PMCW915U3B Radio Gateway 1.3.0
  protocol:  1.3.0
  local eui: E0798DFFFE733FB2
  channel:   0   tx_power: -100 ddBm
  forward:   1   decode: 1   hop: 1 (engine off)
  timings:   delay=500ms timeout=7000ms retries=5
  boots:     1   radio: idle   last rssi: -90 dBm
```

Set the channel your sensors use, and persist it:

```
gw> channel 10
ok
gw> commit
ok
```

Poll a sensor for its status (substitute your sensor's EUI):

```
gw> status E0798DFFFE73393F
ok

[IND] status E0798DFFFE73393F batt=4217mV rssi=-96dBm events=25
```

`ok` means the request was accepted and is on the air. The `[IND]` line that follows is the sensor's actual reply, carrying battery voltage, link quality and the number of stored events. Retrieve the most recent event's first page:

```
gw> latest E0798DFFFE73393F
ok

[IND] data E0798DFFFE73393F blk=24 pg=0 recs=45 | ts=8640.0 ppg_g=101376 mlx=2943 acc=16384/0/0 die=6400
```

That is the whole round trip: configure, poll, receive.

5. Text console reference

Commands are case-sensitive, whitespace-separated, terminated by Enter. Characters are echoed as you type; backspace works.

5.1 Command list

This is the module's own `help` output:

```
gw> help
commands:
  info                        module identity + live config
  status <eui>                poll sensor status
  latest <eui> [page]         poll latest event page (0-63)
  read <eui> <block> <page>   poll block 0-2047, page 0-63
  get <param>                 read a config parameter
  set <param> <value>         stage a config parameter
  commit | factory | reload   persist / defaults / discard edits
  channel <0-25>              alias: set operating_channel
  decode <0|1>                alias: set decode_records
params:
  role
  operating_channel
  tx_power_ddbm
  rx_forward_enable
  decode_records
  tx_delay_ms
  tx_timeout_ms
  no_response_retries
  hop_enable
```

```
hop_table_len
hop_table
hop_dwell_ms
frame_size (ro)
local_eui (ro)
```

5.2 Command detail

Command	Effect
`info`	Module identity, firmware and protocol versions, own EUI, live configuration, boot count, radio state, last received signal strength.
`status <eui>`	Polls one sensor for battery, signal strength, and stored-event count. Replies as `[IND] status`.
`latest <eui> [page]`	Retrieves a page of the sensor's most recent event. `page` is 0–63 and defaults to 0.
`read <eui> <block> <page>`	Retrieves a specific stored event (`block`, 0–2047) and page (0–63).
`get <param>`	Prints the working value of a configuration parameter.
`set <param> <value>`	Stages a value in the RAM working copy. Not persistent until `commit`.
`commit`	Validates all staged values, writes them to non-volatile memory, applies channel and power immediately.
`factory`	Restores factory defaults and commits them.
`reload`	Discards staged edits and reloads the committed values.
`channel <n>`	Shortcut for `set operating_channel <n>`. Still requires `commit`.
`decode <0|1>`	Shortcut for `set decode_records <0|1>`.
`gpio`	Prints the state of all twelve header GPIOs, mode and level per pin (§9).
`gpio <pin> <mode>`	Configures pin 0–11 to mode 0–5. Applies immediately, but not persistent until the mode is also staged into `gpio_mode` and committed.
`gpio <pin> <0|1>`	Drives an output pin low or high. `error 0x0050 (PIN_MODE)` if the pin is not an output.
`gpio <pin>`	Reads and prints one pin's level.

5.3 Responses

Response	Meaning
`ok`	Command accepted. For sensor polls, the transaction is on the air and data follows as an indication.
`<param> = <value>`	Result of `get`.
`error 0xNNNN (NAME)`	Command rejected. See §6.6.
`usage: <syntax>`	Argument count or length wrong. The command was not attempted.
`unknown command '<word>' — try 'help'`	Unrecognized command.
`[IND] status <eui> batt=… rssi=… events=…`	A sensor status reply arrived.
`[IND] data <eui> blk=… pg=… recs=… | …`	A sensor data page arrived (first record summarized).
`[IND] no response from <eui>`	The sensor did not answer within the retry budget.
`[IND] flash reset <eui> done`	A flash-reset acknowledgement arrived.

Captured examples:

```
gw> channel 30
error 0x0013 (CERT_LIMIT)

gw> status ZZZ
usage: status <eui16>

gw> get local_eui
local_eui = E0798DFFFE733FB2

gw> bogus
unknown command 'bogus' — try 'help'
```

An EUI of the wrong length is answered with `usage.` A 16-character EUI containing non-hexadecimal returns `error 0x0030 (TARGET)`.

6. Binary protocol reference

This is the version-frozen machine contract. Implement against it for production integration.

6.1 Frame format

```
+------+------+-----+--------+---------+-------+
| SOF  | LEN  | SEQ | OPCODE | PAYLOAD | CRC16 |
| 2 B  | 2 B  | 1 B |  1 B   |   n B   |  2 B  |
+------+------+-----+--------+---------+-------+
```

Field	Detail
SOF	Always `0xA5 0x5A`.
LEN	`uint16` little-endian. Counts OPCODE + PAYLOAD only, excluding SOF, LEN itself, SEQ and CRC. Minimum value 1.
SEQ	You choose it and the module echoes it in the response. `0x00` is reserved for module-originated indications.
OPCODE	See §6.4.
PAYLOAD	Opcode-specific, `LEN − 1` bytes.
CRC16	CRC-16/CCITT-FALSE (polynomial `0x1021`, initial value `0xFFFF`, no reflection, no final XOR) computed over LEN, SEQ, OPCODE and PAYLOAD, which is everything between SOF and CRC. Transmitted little-endian.

Multi-byte numeric fields are little-endian except the sensor record timestamp, which is big-endian (see §8).

Bound your receive buffer at 2200 bytes, the maximum frame size.

6.2 Request/response model

- Every request receives either a response frame, an `ACK`, or a `NACK`, all echoing your `SEQ`.
- Sensor requests are answered twice: an immediate `ACK` (accepted, on the air), then later an indication carrying the actual result.
- Indications always carry `SEQ = 0x00`.
- A mid-frame stall of more than ~100 ms resets the parser, so a truncated or aborted frame cannot wedge the interface.

6.3 Worked example — module identity

Request `MOD_INFO` with `SEQ = 0x11`:

```
TX: A5 5A 01 00 11 01 17 D2
    |SOF| LEN |SQ|OP| CRC |
```

`LEN = 0x0001` (opcode only, no payload). Actual response:

```
RX: A5 5A 73 00 11 01
    50 4D 43 57 39 31 35 55 33 42 20 52 61 64 69 6F 20 47 61 74 65 77 61 79 20 31 2E 33 2E 30
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    01 03 00
    02 01 00 00
    45 30 37 39 38 44 46 46 46 45 37 33 33 46 42 32
    50 48 59 5F 53 74 75 64 69 6F 5F 39 31 35 4D 5F 32 47 46 53 4B 5F 35 30 4B 62 70 73 5F 32 35 4B
    00 00 00 00 00 00 00 00
    19
    C8 00
    F1 BB
```

Decoded:

Field	Bytes	Value
`banner`	ascii[48], NUL-padded	`PMCW915U3B Radio Gateway 1.3.0`
`protocol_version`	`01 03 00`	1.3.0
`fw_version`	`02 01 00 00`	2.1.0.0
`local_eui`	ascii[16]	`E0798DFFFE733FB2`
`phy_id`	ascii[40], NUL-padded	`PHY_Studio_915M_2GFSK_50Kbps_25K`
`channel_max`	`19`	25
`tx_power_max_ddbm`	`C8 00`	200 (= +20.0 dBm)

Check `protocol_version` at startup. Refuse to operate if the major version differs from the one your host was built against.

6.4 Opcodes

Code	Name	Direction	Request payload	Response
`0x01`	`MOD_INFO`	Host → Module	*(none)*	Identity block (§6.3)
`0x02`	`MOD_STATUS`	Host → Module	*(none)*	`rf_state u8`, `last_rssi_dbm i16`, `last_error u16`, `boot_count u32`, `hop_enabled u8`, `operating_channel u8`
`0x10`	`CFG_GET`	Host → Module	`param_id u16`	`param_id u16`, `type u8`, `value[]`
`0x11`	`CFG_SET`	Host → Module	`param_id u16`, `value[]`	`ACK` / `NACK`
`0x12`	`CFG_COMMIT`	Host → Module	*(none)*	`ACK` / `NACK`
`0x13`	`CFG_FACTORY_RESET`	Host → Module	*(none)*	`ACK` / `NACK`
`0x14`	`CFG_RELOAD`	Host → Module	*(none)*	`ACK` / `NACK`
`0x20`	`SENSOR_READ_LATEST`	Host → Module	`eui ascii[16]`, `page u8`	`ACK`, then `SENSOR_DATA_IND`
`0x21`	`SENSOR_READ_BLOCK`	Host → Module	`eui ascii[16]`, `block u16`, `page u8`	`ACK`, then `SENSOR_DATA_IND`
`0x22`	`SENSOR_STATUS`	Host → Module	`eui ascii[16]`	`ACK`, then `SENSOR_STATUS_IND`
`0x23`	`SENSOR_FLASH_RESET`	Host → Module	`eui ascii[16]`, `ack_flag u8`	`ACK` / `NACK`
`0x30`	`SENSOR_DATA_IND`	Module → Host	—	See §6.5
`0x31`	`SENSOR_STATUS_IND`	Module → Host	—	`eui ascii[16]`, `battery_mv u16`, `rssi_dbm i16`, `block_counter u16`
`0x32`	`SENSOR_NO_RESPONSE_IND`	Module → Host	—	`eui ascii[16]`

0x40	FW_UPDATE_ENTER	Host → Module	unlock_key u32	Reserved. Returns ERR_STATE in v2.1.0.
0x50	GPIO_CONFIG	Host → Module	pin u8 (0–11), mode u8 (0–5), initial u8 (0/1, outputs only)	ACK / NACK
0x51	GPIO_WRITE	Host → Module	pin u8 (0–11), value u8 (0/1)	ACK / NACK
0x52	GPIO_READ	Host → Module	pin u8 (0–11)	pin u8, value u8
0x53	GPIO_WRITE_MASK	Host → Module	mask u16 (bit *n* = pin *n*), values u16	ACK / NACK
0x54	GPIO_READ_ALL	Host → Module	*(none)*	values u16, configured_mask u16
0x55	GPIO_CHANGE_IND	Module → Host	—	pin u8, value u8, timestamp u32
0x7E	ACK	Module → Host	—	seq u8, opcode u8
0x7F	NACK	Module → Host	—	seq u8, opcode u8, error_code u16

Two MOD_STATUS fields are easy to misread. hop_enabled returns the stored value of the hop_enable parameter rather than the state of the radio: the hop engine is dormant in this release, so the radio stays on operating_channel whatever this field reports (§13.2). boot_count increments once per boot and is held in non-volatile memory, so it advances across both a power cycle and a module reset, and it is not reset by CFG_FACTORY_RESET.

Captured examples:

```
MOD_STATUS
TX: A5 5A 01 00 12 02 27 B7
RX: A5 5A 0C 00 12 02 01 A6 FF 32 00 01 00 00 00 01 00 B5 2D
              rf_state=01 (receiving)
              last_rssi=FFA6 = -90 dBm
              last_error=0032 = ERR_NO_RESPONSE (from an earlier transaction)
              boot_count=00000001
              hop_enabled=01   operating_channel=00

CFG_GET operating_channel (0x0002)
TX: A5 5A 03 00 13 10 02 00 8A 65
RX: A5 5A 05 00 13 10 02 00 00 00 D6 44
              param_id=0002  type=00 (u8)  value=00

CFG_SET operating_channel = 0            -> accepted
TX: A5 5A 04 00 14 11 02 00 00 27 BE
RX: A5 5A 03 00 14 7E 14 11 08 1F        (ACK: seq 0x14, opcode 0x11)

CFG_SET operating_channel = 30           -> rejected, outside certified plan
TX: A5 5A 04 00 15 11 02 00 1E 89 E7
RX: A5 5A 05 00 15 7F 15 11 13 00 EB DD  (NACK: error 0x0013 ERR_CERT_LIMIT)

SENSOR_STATUS
TX: A5 5A 11 00 16 22 45 30 37 39 38 44 46 46 46 45 37 33 33 39 33 46 44 7E
                        "E 0 7 9 8 D F F F E 7 3 3 9 3 F"
RX: A5 5A 03 00 16 7E 16 22 32 92                        (ACK)
    A5 5A 17 00 00 31 45...46 7A 10 A0 FF 19 00 7C B2    (STATUS_IND, SEQ=00)
              battery=107A = 4218 mV   rssi=FFA0 = -96 dBm   events=0019 = 25
```

6.5 Indications

SENSOR_DATA_IND (0x30) payload:

Field	Type	Meaning
target_eui	ascii[16]	Which sensor replied
block	u16	Event index (0xFFFF if the module could not attribute it)
page	u8	Page within the event, 0–63

`record_count`	u8	Number of records in this indication
`format`	u8	`0` = raw page, `1` = decoded records
`payload`	—	Depends on `format`

`format = 0` (raw): payload is the sensor's 2048-byte flash page exactly as stored. Decode it yourself using §8.

`format = 1` (decoded): payload is `record_count` records of 53 packed little-endian bytes each.

> **A full page arrives as two indications.** 45 decoded records exceed the 2200-byte frame limit, so the module splits each page into consecutive indications of at most 23 records, all carrying the same `eui`, `block` and `page`. Reassemble them in arrival order. Raw mode (`format = 0`) always fits in a single indication.

Captured (block 1, page 1, decoded):

```
TX: A5 5A 14 00 17 21 45 30 37 39 38 44 46 46 46 45 37 33 33 39 33 46 01 00 01 35 4F
RX: A5 5A 03 00 17 7E 17 21 D4 E7                              (ACK)
    A5 5A D9 04 00 30                                          (DATA_IND, LEN=1241)
    45 30 37 39 38 44 46 46 46 45 37 33 33 39 33 46            eui
    01 00                                                       block = 1
    01                                                          page = 1
    17                                                          23 records in this frame
    01                                                          format = decoded
    6D 0B 00 00  00  10 84 01 00  80 84 01 00 ...              first record
    …
    (a second DATA_IND follows with the remaining 22 records)
```

6.6 Error codes

Code	Name	Meaning and what to do
`0x0000`	`MOD_OK`	Success.
`0x0001`	`ERR_CRC`	Frame checksum mismatch. Retransmit.
`0x0002`	`ERR_LENGTH`	`LEN` inconsistent or frame too large. Check your framing.
`0x0003`	`ERR_OPCODE`	Unknown opcode, most likely a protocol version mismatch.
`0x0010`	`ERR_PARAM_ID`	No such configuration parameter.
`0x0011`	`ERR_RANGE`	Value outside the parameter's allowed range.
`0x0012`	`ERR_READONLY`	Parameter cannot be written.
`0x0013`	`ERR_CERT_LIMIT`	Value would exceed the certified RF envelope. Not applied and not stored.
`0x0020`	`ERR_NVM`	Non-volatile write failed. Retry; persistent failure indicates hardware fault.
`0x0030`	`ERR_TARGET`	Malformed EUI. It must be exactly 16 hexadecimal characters.
`0x0031`	`ERR_BUSY`	A radio transaction is already outstanding. Wait for its indication, then retry.
`0x0032`	`ERR_NO_RESPONSE`	Sensor did not answer within the retry budget.
`0x0040`	`ERR_STATE`	Operation invalid in the current state (also returned by `FW_UPDATE_ENTER` in v2.1.0).
`0x0050`	`ERR_PIN_MODE`	The GPIO pin is not configured for the operation you asked for, such as writing a pin that is an input or disabled, or reading one still in mode 0. Configure the pin first (§9).

7. Configuration parameters

Configuration is a two-step process. `CFG_SET` (or console `set`) stages a value in a RAM working copy where it is range-checked immediately. `CFG_COMMIT` validates the entire set, writes it to non-volatile memory, and applies channel and transmit power to the radio. Values that are staged but never committed are lost on reset or `CFG_RELOAD`.

ID	Name	Type	Range	Default	Persists	Description
`0x0001`	`role`	u8	0–1	1	Yes	Radio role. Leave at 1.
`0x0002`	`operating_channel`	u8	0–25	10	Yes	Channel *N* = (902 + *N*) MHz. Must match your sensors.
`0x0003`	`tx_power_ddbm`	i16	−100 … 200	200	Yes	Transmit power in deci-dBm (200 = +20.0 dBm).
`0x0010`	`rx_forward_enable`	u8	0–1	1	Yes	Forward received sensor traffic to the host. See §11.9.
`0x0011`	`decode_records`	u8	0–1	1	Yes	`1` = decoded records, `0` = raw page.
`0x0020`	`tx_delay_ms`	u16	0–65535	500	Yes	Minimum spacing between transmissions.
`0x0021`	`tx_timeout_ms`	u16	0–65535	7000	Yes	How long to wait for a sensor reply before retrying.
`0x0022`	`no_response_retries`	u16	0–65535	5	Yes	Attempts before declaring no-response.
`0x0040`	`hop_enable`	u8	0–1	1	Yes	Stored hopping flag. Ships as 1, but the hop engine is dormant and the radio stays on `operating_channel` (§13.2).
`0x0041`	`hop_table_len`	u8	0–25	25	Yes	Hop sequence length.
`0x0042`	`hop_table`	u8[25]	each 0–25	zeros	Yes	Hop sequence.
`0x0043`	`hop_dwell_ms`	u16	0–65535	400	Yes	Dwell per hop.
`0x0050`	`gpio_mode`	u8[12]	each 0–5	zeros	Yes	Power-on mode of each header GPIO, indexed by logical pin 0–11. `0` = hi-Z (§9).
`0x0051`	`gpio_out_default`	u16	0–4095	0	Yes	Bitmask, bit *n* = pin *n*. Level

						applied at power-on to pins whose mode is an output.
`0x0052`	`gpio_irq_mask`	u16	0–4095	0	Yes	Bitmask, bit *n* = pin *n*. Input pins that raise `GPIO_CHANGE_IND`. `0` disables all change notification.
`0x00F0`	`frame_size`	u16	2096	2096	—	Read-only. Fixed by the certified PHY.
`0x00F1`	`local_eui`	ascii[16]	—	device ID	—	Read-only. This module's own identity.

The certification-bound parameters (`operating_channel`, `tx_power_ddbm`, `hop_table*`, `hop_dwell_ms`, `frame_size`) are validated against compile-time limits. No command sequence, parameter combination or host action can operate the radio outside its certified envelope. Attempts return `ERR_CERT_LIMIT` and change nothing.

Timeout tuning. `tx_timeout_ms` defaults to 7000 ms because a sensor can be busy writing flash for several seconds (§11.3). Shortening it makes no-response detection faster but risks abandoning transactions the sensor would have answered.

Bench power setting. The +20 dBm default is correct for deployment. On a bench with modules close together, reduce transmit power (`set tx_power_ddbm -100` = −10 dBm) to avoid saturating the receiver.

8. Raw data and decoding

	`decode_records = 0` **(raw)**	`decode_records = 1` **(decoded, default)**
Payload	2048-byte page verbatim	53-byte packed records
Indications per page	1	2 (23 + 22 records)
Timestamp endianness	Big-endian (as stored)	Little-endian (normalized)
24-bit fields	3 bytes, MSB-first	Widened to 32-bit little-endian
Best for	Archival, minimum host work, byte-exact fidelity	Direct consumption without bit manipulation

Raw mode is recommended when you intend to store or forward data unmodified. It preserves a sensor's exact bytes and halves the indication count.

9. GPIO control

Twelve pins on the module's headers are general-purpose I/O that your host drives and senses over the same UART interface it already uses for the radio. There is no extra wiring and no second protocol to learn: a carrier-board host can hang an LED, a relay, an enable line or a push-button off the module and control it with the frames in this section.

Pins are addressed by logical index, 0–11. That is the only addressing form on the wire. The protocol has no way to name a SoC port and pin, so nothing here can reach the pins the module owns (§9.7).

> The byte sequences elsewhere in this manual were captured from a running module. The examples in this section are illustrative, written from the protocol definition pending a hardware capture of the GPIO firmware.

9.1 The pin map

GPIO index	SoC pin	Header	Header pin
0	`PB01`	1	1
1	`PB00`	1	2
2	`PA00`	1	3
3	`PA04`	1	7
4	`PC00`	2	9
5	`PC01`	2	8
6	`PC02`	2	7
7	`PC03`	2	6
8	`PC04`	2	5
9	`PC05`	2	4
10	`PC06`	2	3
11	`PC07`	2	2

Note that header 2's GPIOs run descending down the header: index 4 is H2-9 and index 11 is H2-2. Use the table rather than arithmetic.

9.2 Pin modes

Mode	Meaning	Readable	Writable
`0`	Disabled. Hi-Z, no pull. Factory default on every pin.	No	No
`1`	Input, no pull	Yes	No
`2`	Input with pull-up	Yes	No
`3`	Input with pull-down	Yes	No
`4`	Output, push-pull	Yes	Yes
`5`	Output, open-drain	Yes	Yes

An operation that does not match the pin's current mode is rejected with `ERR_PIN_MODE` (`0x0050`): writing an input, or touching a pin still in mode 0. Nothing is applied and nothing is stored.

Open-drain (mode 5) pulls low and releases high; the high level must come from a pull-up on your carrier board. Use it for shared or wired-OR nets and for anything driving a rail other than the module's own 3.3 V.

9.3 Binary opcodes

Code	Name	Request payload	Response
`0x50`	`GPIO_CONFIG`	`pin u8` (0–11), `mode u8` (0–5), `initial u8` (0/1)	`ACK` / `NACK`
`0x51`	`GPIO_WRITE`	`pin u8` (0–11), `value u8` (0/1)	`ACK` / `NACK`
`0x52`	`GPIO_READ`	`pin u8` (0–11)	`pin u8, value u8`
`0x53`	`GPIO_WRITE_MASK`	`mask u16, values u16`	`ACK` / `NACK`
`0x54`	`GPIO_READ_ALL`	*(none)*	`values u16, configured_mask u16`

`0x55`	`GPIO_CHANGE_IND`	*— (module-originated)*	`pin u8, value u8, timestamp u32`

> **Edges are sampled by the module's main loop, not by a hardware interrupt.** A pulse shorter than one loop iteration can be missed entirely. Stretch short pulses on the carrier board, or poll with `GPIO_READ_ALL`, if you need to catch brief events. The choice is deliberate: the EFR32 external-interrupt channels cannot cover all twelve header pins at once, so polling gives uniform coverage rather than silently dropping edges on whichever pins lost the allocation.

`GPIO_CONFIG` (`0x50`) takes a 3-byte payload. `initial` is the level applied at the moment the pin becomes an output; it is ignored for input and disabled modes, so pass `0`. The change takes effect immediately but is not persistent (§9.6).

`GPIO_WRITE` (`0x51`) takes a 2-byte payload. `value` is `0` or `1`, and the pin must already be in mode 4 or 5.

`GPIO_READ` (`0x52`) takes a 1-byte payload. The response echoes the pin index and returns its level. It works for inputs and for outputs, where it reads back the driven level.

`GPIO_WRITE_MASK` (`0x53`) takes a 4-byte payload of two little-endian `u16`s. `mask` selects which pins to change (bit *n* = pin *n*) and `values` supplies their levels. Pins whose mask bit is `0` are untouched, and their bit in `values` is ignored. Every selected pin must be an output; if one is not, the whole operation is rejected with `ERR_PIN_MODE` and no pin changes at all, because the write is atomic. Only bits 0–11 correspond to pins, so send bits 12–15 as zero.

`GPIO_READ_ALL` (`0x54`) takes no payload. It returns two little-endian `u16`s: `values`, the level of every pin (bit *n* = pin *n*), and `configured_mask`, which has a `1` for each pin not in mode 0. Bits in `values` whose `configured_mask` bit is `0` are meaningless, because the pin is hi-Z and floating. Use this for cheap polling rather than twelve separate `GPIO_READ`s.

9.4 Change indications

`GPIO_CHANGE_IND` (`0x55`) is an asynchronous indication, delivered with `SEQ = 0x00` like every other indication. It carries the pin index, its new level and a `timestamp u32` taken from the module's own tick counter, which runs at 32768 Hz and is not wall-clock time. Use it for relative timing between edges and anchor absolute time with your host's clock on arrival.

Indications are emitted only for pins selected in `gpio_irq_mask` (`0x0052`) that are in an input mode. The default mask is `0`, so no change indications are emitted until you enable them. Setting a bit for a pin that is an output or disabled has no effect.

`gpio_irq_mask` is a committed parameter, not a live one: stage it with `CFG_SET` and issue `CFG_COMMIT`.

> A rapidly toggling input will produce one indication per edge and can flood the link. Assume no debouncing in the module: debounce mechanical contacts on the carrier board, or leave them out of `gpio_irq_mask` and poll with `GPIO_READ_ALL` instead.

9.5 Console commands

Command	Effect
`gpio`	Print all twelve pins, mode and current level.
`gpio <pin> <mode>`	Configure pin 0–11 to mode 0–5. Immediate, not persistent.
`gpio <pin> <0|1>`	Drive an output pin low or high.
`gpio <pin>`	Print one pin's level.

There is no console shortcut for persisting a pin configuration. Use the ordinary `set <param> <value>` and `commit` commands on `gpio_mode`, `gpio_out_default` and `gpio_irq_mask` (§9.6).

9.6 Persistence and power-on state

There are two layers, and the distinction matters:

	Live	Persistent
Set by	`GPIO_CONFIG`, `GPIO_WRITE`, `GPIO_WRITE_MASK`, console `gpio`	`CFG_SET` of `gpio_mode` / `gpio_out_default` / `gpio_irq_mask`, then `CFG_COMMIT`
Effect	Immediate	Applied at every power-on and reset
Survives reset	No	Yes

`GPIO_CONFIG` changes the pin now. It does not change what the pin will be after the next reset. To make a net's power-on state deterministic, stage the persistent parameters and commit them:

- `gpio_mode` (`0x0050`) is a `u8[12]` array, one mode per logical pin, index 0 first.
- `gpio_out_default` (`0x0051`) is a `u16` bitmask giving the level applied at power-on to those pins whose committed mode is an output. Bits for non-output pins are ignored.
- `gpio_irq_mask` (`0x0052`) is a `u16` bitmask selecting which input pins raise `GPIO_CHANGE_IND`.

The module applies the committed mode and default level before it accepts its first host command, so there is no window in which a committed net is undriven. This is the mechanism to use for anything that must hold a defined level while your host is still booting.

Factory default is hi-Z on every pin. `gpio_mode` defaults to all zeros and `gpio_out_default` to `0`, so a module that has never been configured drives nothing. `CFG_FACTORY_RESET` returns every pin to that state.

Values staged with `CFG_SET` but never committed are lost on reset or `CFG_RELOAD`, exactly like every other parameter (§7).

9.7 The module owns its own pins

The module reserves several SoC pins for itself: the two UARTs (`PA05`/`PA06` for COM1, `PA07`/`PA08` for COM0), the debug port (`PA01`/`PA02`/`PA03`), and the internal PA/LNA control lines (`PD02`/`PD03`).

None of them are reachable from the host interface. The restriction is structural rather than a permission check that could be bypassed. Every GPIO opcode takes a logical index, the index table contains only the twelve header pins in §9.1, and no opcode or parameter anywhere in the protocol accepts a raw SoC port and pin. An index outside 0 to 11 addresses nothing and is rejected.

In practice this means no sequence of GPIO commands can disable the UART you are talking over, disturb the debug port, or interfere with transmit-path control.

10. Typical workflows

10.1 Enrolling a sensor

There is no pairing transaction. "Enrollment" means *your system records the sensor's EUI*. Obtain it from the sensor's own console (its `info` command) or from device labelling, store it in your registry, and start polling. To unenroll, stop polling and delete the record. The module is never involved.

10.2 Periodic health polling

For each enrolled sensor, on your chosen interval:

20. Send `SENSOR_STATUS` with the sensor's EUI.
21. Expect `ACK`, then `SENSOR_STATUS_IND` with battery, signal strength, and `block_counter` (the number of events stored).
22. If `SENSOR_NO_RESPONSE_IND` arrives instead, treat the sensor as out of range, asleep or busy, and retry on the next cycle rather than immediately.

Record every `(wall_clock_time, block_counter)` observation. This series is what lets you translate calendar times into event indices later (§11.5).

10.3 Retrieving an event

To retrieve event *N* completely, request its 64 pages sequentially, one at a time, each after the previous indication arrives:

```
for page in 0 .. 63:
    SENSOR_READ_BLOCK(eui, block = N, page = page)
    wait for SENSOR_DATA_IND (or SENSOR_NO_RESPONSE_IND)
```

Budget roughly 45 to 50 seconds per event. The most recent event is `block_counter - 1` from the latest status poll, or use `SENSOR_READ_LATEST`, which resolves it on the module.

Do not parallelize: the module accepts one transaction at a time and the channel is shared with every other module within range.

10.4 Erasing a sensor's storage

`SENSOR_FLASH_RESET` erases the sensor's event store and resets its counters.

> **This destroys data irreversibly.** Retrieve anything you need first. Set `ack_flag` to 1 to request confirmation from the sensor.

10.5 Changing channel safely

Both ends must agree. If you change the module's channel while sensors remain on the old one they become invisible, since there is no negotiation or auto-discovery.

```
gw> channel 12
ok
gw> commit
ok
```

Move the sensors first, or maintain a per-site channel plan and change both together.

10.6 Driving a carrier-board net from GPIO 4

Suppose header 2 pin 9, which is GPIO 4 on `PC00`, enables a peripheral on your carrier board. It must be off while the host boots and on once the host is running.

First, decide the power-on state and commit it. GPIO 4 is bit 4, so the mask value is `0x0010`.

23. `CFG_SET` `gpio_mode` (`0x0050`) with a 12-byte array whose element 4 is `4` (output, push-pull) and whose other elements stay `0` (hi-Z).
24. `CFG_SET` `gpio_out_default` (`0x0051`) to `0x0000`, so GPIO 4 comes up low.
25. `CFG_COMMIT`. From now on, every power-on drives that net low before your host sends its first command. Nothing floats.

Then drive it at run time.

26. `GPIO_WRITE` (`0x51`) with `pin = 4`, `value = 1` to assert the enable.
27. `GPIO_READ` (`0x52`) with `pin = 4` to read the level back, or `GPIO_READ_ALL` (`0x54`) if you are watching several pins.

From the console the same sequence is:

```
gw> gpio 4 4
gw> gpio 4 1
gw> gpio 4
```

Two things to keep straight. The `GPIO_CONFIG` and `GPIO_WRITE` pair changes the pin immediately and is forgotten at reset, while the committed `gpio_mode` and `gpio_out_default` are what the module restores at power-on. And `initial` in `GPIO_CONFIG` is a one-shot level applied as the pin becomes an output, which is not the same field as the persistent `gpio_out_default`. If you only ever call `GPIO_CONFIG`, the net will be hi-Z again after the next reset (§9.6).

If step 4 comes back `NACK` with `0x0050 ERR_PIN_MODE`, the pin is not an output. Either step 1 was never committed and the module has since reset, or the live `GPIO_CONFIG` was never issued.

11. Application notes

11.1 There is no discovery protocol

Sensors should answer only to their own EUI. Nothing should broadcast, enumerate, or announce. Every sensor's 16-character EUI has to reach your registry out of band, read from the sensor's console, a label, or a manufacturing record.

Plan for this in your commissioning process. A sensor whose EUI you do not have is unreachable, even sitting on the desk next to the module.

11.2 EUI formatting — always use 16 characters

The sensor formats its identity without leading zeros, so a sensor whose 64-bit ID begins with a zero nibble transmits fewer than 16 characters. The module normalizes this by left-padding received identities to the full 16-character form.

Always use the zero-padded 16-character uppercase form in requests and as your registry key. A 16-character EUI containing non-hexadecimal characters is rejected with `ERR_TARGET`; a shorter one is rejected by the console with `usage:`.

11.3 Overlapping modules and de-duplication

With `rx_forward_enable = 1` (the default), a module forwards every valid sensor frame it demodulates, including replies to polls issued by other modules on the same channel.

This is intentional: it is how site redundancy works. Install overlapping modules, and a sensor's reply reaches your back end through whichever ones hear it, so a single module's coverage gap does not lose data.

Your back end must de-duplicate. The natural key is `(eui, block, page)`; identical tuples from different modules carry the same payload. Keep the copy with the strongest signal if you want a link-quality signal for site planning.

Set `rx_forward_enable = 0` only if you specifically want a module to report nothing but its own transactions.

11.4 Channel airtime is a shared, site-wide budget

Every module within range shares the sensors' channel.

Activity	Airtime cost
One status poll	≈0.7 s
One complete event retrieval (64 pages)	≈45–50 s
Channel capacity	≈1.3–1.4 transactions per second

Status polling is inexpensive: 100 sensors polled every 6 hours consumes about 0.3 % of the channel. Event retrieval is the constraint, since roughly one event per minute saturates a channel.

11.5 Confirm your sensors' firmware matches this contract

Sensor firmware predating this module release may implement different over-the-air behavior (including continuous unsolicited transmission rather than poll/response). Such units will not respond to polls as documented here.

Before deploying, verify against one sensor: a `status` poll should return an `[IND] status` line within a second or two. If a sensor is transmitting continuously or does not answer polls, its firmware needs updating to a poll-response build.

12. Troubleshooting

Symptom	Likely cause	What to do
No `gw>` prompt	Wrong pins, wrong baud rate, TX/RX swapped, or module unpowered	Verify 460800 8N1; check `COM0_TX` → adapter RX and `COM0_RX` → adapter TX; confirm 3.3 V on both `BAT+` pins
Garbled characters	Baud mismatch or 5 V logic on a 3.3 V input	Confirm 460800; level-shift any 5 V host
`error 0x0030 (TARGET)`	EUI contains non-hexadecimal characters	Use the zero-padded uppercase form (§11.8)
`usage: status <eui16>`	EUI is not exactly 16 characters long	Zero-pad to 16 characters
`error 0x0031 (BUSY)`	A transaction is already in flight	Wait for its indication before sending the next request
`error 0x0013 (CERT_LIMIT)`	Channel > 25 or power > 200 deci-dBm	Choose a value inside the certified envelope
`[IND] no response from …` for a healthy sensor	Sensor busy writing flash, asleep, out of range, or on another channel	See §11.3, §11.6, and verify channel agreement
Settings revert after power cycle	`commit` was never issued	`set …` then `commit`; verify with `info`
`error 0x0050 (PIN_MODE)` on a GPIO write	The pin is an input or still disabled (mode 0)	Configure it to mode 4 or 5 first (§9.2)
A GPIO reverts to hi-Z after reset	Only `GPIO_CONFIG` was used, which is not persistent	Stage `gpio_mode` / `gpio_out_default` and `commit` (§9.6)
No `GPIO_CHANGE_IND` from an input	`gpio_irq_mask` is 0 (the default) or the pin is not an input	Set the pin's bit in `gpio_irq_mask` and `commit` (§9.4)
Data indications never arrive	`rx_forward_enable` is 0	`set rx_forward_enable 1`, `commit`
Binary parser sees garbage on the console port	ASCII console output is interleaved	Use COM1, or scan for `A5 5A` and skip non-frame bytes (§3.1)
Duplicate pages in the back end	Multiple modules heard the same reply	Expected. De-duplicate on `(eui, block, page)` (§11.9)
`events=` never increases	Sensor is stationary; no wrist events	Expected (§11.2)
Poor or no range	No antenna on J3, or transmit power reduced from a bench session	Fit a 50 Ω antenna; check `tx_power_ddbm` and `commit`

13. Known limitations in this firmware

Firmware v2.1.0, protocol 1.3.0:

28. **`FW_UPDATE_ENTER` is reserved and not planned for this module.** The opcode is accepted, but the unlock key is **not** checked: any key returns `ERR_STATE`, and the module does not reboot. Do not treat a rejected key as an authentication result. This module is reprogrammed over its debug port by physical connection (datasheet §12.4); it carries no bootloader and accepts firmware neither over UART nor over the air.
29. **Frequency hopping is not active.** The hop parameters are accepted, validated and persisted, but the hop engine is dormant and the radio stays on `operating_channel`. Note that `hop_enable` ships set to 1 and `MOD_STATUS` echoes that value, so neither the parameter nor the status field tells you whether hopping is running in this release. Channel selection and any site-level channel plan are host responsibilities (§10.5, §11.10).
30. **Console indication mirroring is automatic.** When a binary client is active on the console port, indications are mirrored there. There is no explicit toggle yet.
31. **The console grammar is not version-frozen.** Text commands may change between releases. Machine integrations must use the binary protocol.
32. **The module is not yet certified.** See the datasheet, §11. Engineering evaluation only.

Appendix A — reference host code

Complete, working Python client. This is the code used to capture the traces in this manual.

```
import struct
import time
import serial

PORT = "COM18"          # whichever port reaches the module's UART
BAUD = 460800

def crc16(data: bytes) -> int:
    """CRC-16/CCITT-FALSE: poly 0x1021, init 0xFFFF, no reflection, no final XOR."""
    crc = 0xFFFF
    for b in data:
        crc ^= b << 8
        for _ in range(8):
            crc = ((crc << 1) ^ 0x1021) & 0xFFFF if crc & 0x8000 else (crc << 1) & 0xFFFF
    return crc

def build_frame(seq: int, opcode: int, payload: bytes = b"") -> bytes:
    body = struct.pack("<H", 1 + len(payload)) + bytes([seq, opcode]) + payload
    return b"\xA5\x5A" + body + struct.pack("<H", crc16(body))

def parse_frames(buf: bytearray):
    """Yield (seq, opcode, payload); tolerates interleaved ASCII console text."""
    i = 0
    while i + 8 <= len(buf):
        if buf[i] != 0xA5 or buf[i + 1] != 0x5A:
            i += 1
            continue
        length = struct.unpack_from("<H", buf, i + 2)[0]
        end = i + 4 + 1 + length + 2
        if end > len(buf):
            break                                   # incomplete: wait for more
        body = bytes(buf[i + 2 : i + 5 + length])
        rx_crc = struct.unpack_from("<H", buf, i + 5 + length)[0]
        if crc16(body) == rx_crc:
            yield buf[i + 4], buf[i + 5], bytes(buf[i + 6 : i + 5 + length])
            i = end
        else:
            i += 1                                  # false SOF; resynchronize

def collect(ser, seconds: float):
    buf = bytearray()
    deadline = time.time() + seconds
    while time.time() < deadline:
        buf.extend(ser.read(4096))
    return list(parse_frames(buf))

# ------------------------------------------------------------------ example use
ser = serial.Serial(PORT, BAUD, timeout=0.05)

# 1. Identify the module and check protocol compatibility
ser.write(build_frame(0x01, 0x01))                          # MOD_INFO
for seq, op, pl in collect(ser, 1.0):
    if op == 0x01:
        banner = pl[:48].rstrip(b"\0").decode()
        major, minor, patch = pl[48], pl[49], pl[50]
        local_eui = pl[55:71].decode()
        print(f"{banner} | protocol {major}.{minor}.{patch} | eui {local_eui}")
        assert major == 1, "incompatible protocol major version"

# 2. Poll a sensor's status
SENSOR = b"E0798DFFFE73393F"                                # 16 ASCII hex chars
ser.write(build_frame(0x02, 0x22, SENSOR))                  # SENSOR_STATUS
for seq, op, pl in collect(ser, 4.0):
    if op == 0x7F:                                          # NACK
        print("rejected, error 0x%04X" % struct.unpack("<H", pl[2:4])[0])
    elif op == 0x31:                                        # SENSOR_STATUS_IND
        eui = pl[:16].decode()
        battery_mv, rssi_dbm, events = struct.unpack("<HhH", pl[16:22])
        print(f"{eui}: {battery_mv} mV, {rssi_dbm} dBm, {events} events stored")
    elif op == 0x32:                                        # SENSOR_NO_RESPONSE_IND
        print(f"{pl[:16].decode()}: no response")

# 3. Retrieve one page of an event, decoded
ser.write(build_frame(0x03, 0x21, SENSOR + struct.pack("<HB", 1, 0)))
records = []
for seq, op, pl in collect(ser, 5.0):
    if op == 0x30:                                          # SENSOR_DATA_IND
        block, page, count, fmt = struct.unpack("<HBBB", pl[16:21])
        body = pl[21:]
        if fmt == 1:                                        # decoded, 53 B/record
            for n in range(count):
                r = body[n * 53 : (n + 1) * 53]
                ts, ms = struct.unpack_from("<IB", r, 0)
```

```
                ppg = struct.unpack_from("<4I", r, 5)
                mlx, ax, ay, az = struct.unpack_from("<4h", r, 37)
                records.append({
                    "ts": ts, "ms": ms, "ppg_green": ppg[0],
                    "temp_c": mlx / 100.0,
                    "accel_mg": (ax * 0.061, ay * 0.061, az * 0.061),
                })
        else:                                           # raw 2048-byte page
            records.append({"raw_page": body})
print(f"received {len(records)} records")     # 45 after both indications arrive

ser.close()
```

Appendix B — quick reference tables

Connections: `BAT+` = 3.3 V (Header 2 pins 10, 11) · `GND` (Header 1 pin 12, Header 2 pin 12) · `COM1` = Header 1 pins 8 (TX) / 9 (RX) · `COM0` = Header 1 pins 10 (TX) / 11 (RX) · `RESET` = Header 2 pin 1 · antenna on J3.

Serial settings: 460800 baud, 8 data bits, no parity, 1 stop bit, no flow control, 3.3 V logic (not 5 V tolerant).

Frame: `A5 5A | LEN(u16 LE) | SEQ | OPCODE | PAYLOAD | CRC16(u16 LE)` `LEN` counts OPCODE + PAYLOAD. CRC-16/CCITT-FALSE over LEN…PAYLOAD. Max frame 2200 bytes.

Opcodes: `01` MOD_INFO · `02` MOD_STATUS · `10` CFG_GET · `11` CFG_SET · `12` CFG_COMMIT · `13` CFG_FACTORY_RESET · `14` CFG_RELOAD · `20` SENSOR_READ_LATEST · `21` SENSOR_READ_BLOCK · `22` SENSOR_STATUS · `23` SENSOR_FLASH_RESET · `30` SENSOR_DATA_IND · `31` SENSOR_STATUS_IND · `32` SENSOR_NO_RESPONSE_IND · `40` FW_UPDATE_ENTER · `50` GPIO_CONFIG · `51` GPIO_WRITE · `52` GPIO_READ · `53` GPIO_WRITE_MASK · `54` GPIO_READ_ALL · `55` GPIO_CHANGE_IND · `7E` ACK · `7F` NACK

GPIO opcode payloads: `50` CONFIG `pin u8, mode u8, initial u8` · `51` WRITE `pin u8, value u8` · `52` READ `pin u8` → `pin u8, value u8` · `53` WRITE_MASK `mask u16, values u16` · `54` READ_ALL → `values u16, configured_mask u16` · `55` CHANGE_IND → `pin u8, value u8, timestamp u32`

GPIO pin map (logical index, the only addressing form on the wire):

Index	0	1	2	3	4	5	6	7	8	9	10	11
SoC pin	PB01	PB00	PA00	PA04	PC00	PC01	PC02	PC03	PC04	PC05	PC06	PC07
Header	H1-1	H1-2	H1-3	H1-7	H2-9	H2-8	H2-7	H2-6	H2-5	H2-4	H2-3	H2-2

GPIO modes: `0` disabled/hi-Z (factory default) · `1` input · `2` input pull-up · `3` input pull-down · `4` output push-pull · `5` output open-drain

GPIO parameters: `0050` `gpio_mode` u8[12] · `0051` `gpio_out_default` u16 bitmask · `0052` `gpio_irq_mask` u16 bitmask. All are staged with `CFG_SET` and applied at power-on after `CFG_COMMIT`.

Errors: `0000` OK · `0001` CRC · `0002` LENGTH · `0003` OPCODE · `0010` PARAM_ID · `0011` RANGE · `0012` READONLY · `0013` CERT_LIMIT · `0020` NVM · `0030` TARGET · `0031` BUSY · `0032` NO_RESPONSE · `0040` STATE · `0050` PIN_MODE

Timing: one radio transaction ≈0.7 s · one event (64 pages) ≈45–50 s · channel capacity ≈1.3–1.4 transactions/s

Sensor geometry: event = 64 pages · page = 45 records · record = 45 bytes on air, 53 bytes decoded

Revision history

Issue	Date	Change
1.0	March 2023	First issue.
1.1	February 2024	Revised for the current module hardware.
2.0	November 2025	Console and binary interface chapters reworked.

2.1	May 2026	Current issue. Firmware v2.1.0 and protocol 1.3.0: GPIO control, non-volatile configuration, stateless EUI addressing, and application notes.