binho

APPLICATION NOTE

Interfacing an ams OSRAM time-of-flight sensor over I3C with the Binho Supernova

AN0007

Document information

Keywords	AN0007, Supernova, I3C, MIPI I3C, HDR-DDR, ams, OSRAM, TMF8829, LIGHTRANGER 14 Click, dToF, time-of-flight, ranging, multi-zone, SPAD, ENTDAA, CCC, BCR, DCR, FIFO, bootloader
Abstract	The ams OSRAM TMF8829 is a direct time-of-flight ranging sensor with a MIPI I3C interface alongside I2C and SPI. It reports a distance for each zone of a SPAD array rather than one distance for the field of view, up to 1536 zones per measurement, and it differs from a register-mapped sensor in three respects that determine how a host is written: its ranging firmware is downloaded over the bus at every power-up, results are returned through a FIFO rather than a register file, and a measurement is kilobytes rather than bytes. This application note shows how to bring one up from a Binho Supernova USB host adapter without writing any firmware. It covers enumeration and dynamic address assignment, forcing the bootloader and downloading the ranging application, selecting a zone configuration, the frame header field that determines the size of every zone, the transfer-length limit that a full-resolution frame exceeds, the row interleaving used by the two largest configurations, rendering the results, and an HDR-DDR frame transfer checked against the single data rate read of the same frame. The worked example uses the TMF8829 on a LIGHTRANGER 14 Click, and a reference Python utility that produced every output in this document is supplied.

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

REVISION 1.0
1 SEPTEMBER 2026

1 Introduction

The ams OSRAM TMF8829 is a direct time-of-flight ranging sensor with a MIPI I3C interface alongside I2C and SPI, on the same two pins. It measures distance by timing returned photons against an emitted laser pulse, across an array of single-photon avalanche diodes divided into zones, and reports a separate result for every zone rather than one distance for the field of view. The largest configuration reports 1536 zones, and each zone can report up to four returns at different distances.

I3C keeps the two-wire footprint and adds a substantial feature set on top of it.

Table 1. What I3C adds to a two-wire sensor interface

Feature	What it gives a design
Dynamic address assignment	The controller hands out addresses at run time, so static address collisions stop being a board-design constraint and identical parts can share a bus
Identification from the bus	Every target reports a 48-bit provisional identifier, a device class and its bus capabilities, so a controller can enumerate a bus it was not built for
Common command codes	A standard command set, broadcast and direct, for enumeration, identification, configuration, interrupt control and reset, the same on every compliant target
In-band interrupts	The sensor interrupts the host on the data line, carrying a byte that says why, so no interrupt pin has to be routed or allocated
High data rate modes	Double data rate transfers move two bits per clock edge pair, doubling the signalling rate without raising the clock
Higher signalling rates	Push-pull clocking, to 12.5 MHz in single data rate mode, against the open-drain limits of I2C
Lower power	Push-pull signalling avoids the static pull-up current I2C spends continuously, and activity states let a controller tell targets what bus latency to expect so they can idle harder
Hot-join	A target powered up after the bus is running can ask to be enumerated
Standard reset and recovery	A defined target reset and defined error detection and recovery, rather than per-vendor sequences
I2C compatibility	Legacy I2C devices can share the bus

A target implements the subset of that a sensor needs, and the subset differs between parts. This note works through a TMF8829 from a Binho Supernova USB host adapter: enumerating it, identifying it from the bus, entering its bootloader, downloading and starting its ranging firmware, selecting a zone configuration, reading and decoding measurement frames at every resolution the part offers, rendering the results, transferring a frame in HDR-DDR, and establishing which of the features above the target actually implements.

Three properties of this part determine how a host has to be written, and all three differ from a register-mapped sensor.

Table 2. What differs from a register-mapped sensor

Property	Consequence for the host
The ranging firmware is not resident	The part powers up in a bootloader or a ROM application, neither of which ranges. The application is downloaded over the bus into RAM at every power-up, so it forms part of the boot time of any product built around the part
Results are returned through a FIFO	There is no distance register. A measurement is read as one contiguous transfer whose length is described by the frame itself, and a short read leaves the remainder in the FIFO to corrupt the next frame
A measurement is kilobytes, not bytes	A 48x32 configuration returns 4674 bytes per measurement at the smallest result format. Section 7.3 establishes the transfer limit this runs into and Section 11 gives the size of each configuration

Nothing here requires a microcontroller, firmware, or a driver port. Every step is a command from a host PC.

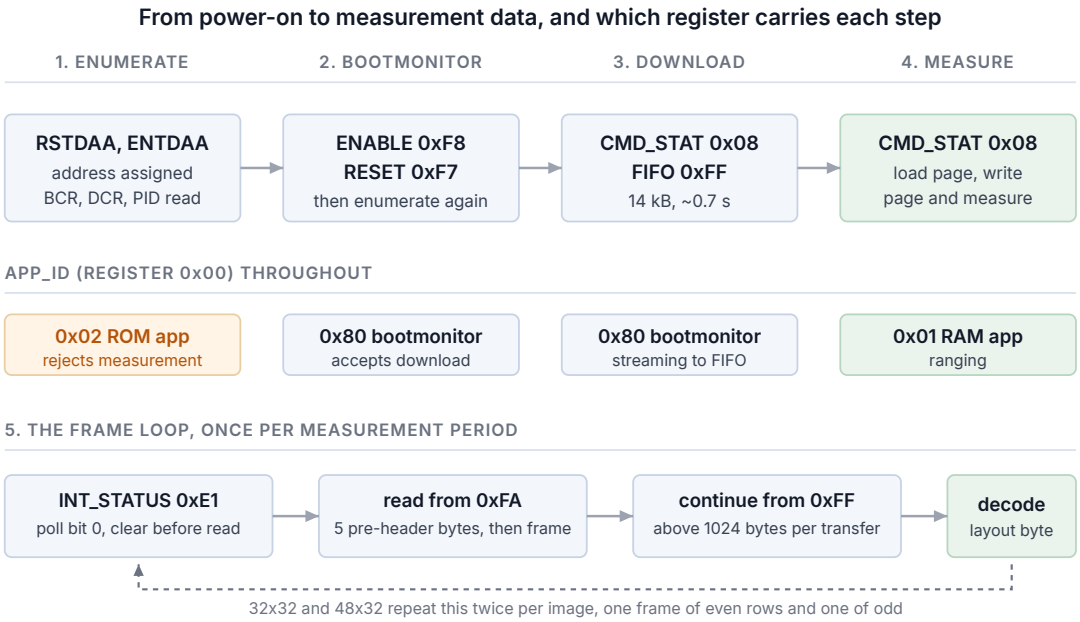

Figure 1. From power-on to measurement data, and which register carries each step.

1.1 Scope

By the end of this document a reader with the hardware in front of them will have enumerated a TMF8829 onto an I3C bus, forced it into its bootloader, downloaded and started its ranging firmware, configured each of its zone grids, read and decoded measurement frames including the two configurations that arrive as two frames, rendered the results, transferred a frame in HDR-DDR and verified it against the single data rate read of the same frame, and established what the target implements from observable effects.

The part worked through is the TMF8829 on a LIGHTRANGER 14 Click. It is exercised on hardware and every output in this document is a real capture.

This note does not cover optical calibration, crosstalk compensation against a cover glass, the histogram output mode, or the point-cloud correction that converts per-zone distances into Cartesian coordinates. Those depend on the optical stack a design puts in front of the sensor and are documented by the sensor vendor.

1.2 Applicable devices

The worked example uses a TMF8829 on a LIGHTRANGER 14 Click. The bus procedures apply to any I3C target; the bootloader, configuration and frame procedures apply to the TMF882x and TMF88xx family, which share the host interface described here.

1.3 Related documents

Table 3. Related documents

Document	Where
TMF8829 datasheet	ams OSRAM, DS001140 v2-00, 2025-Nov-12; the revision whose interface section is quoted in Section 10.2
TMF8829 Host Driver Communication	ams OSRAM; defines the frame structures the datasheet refers to but does not contain
TMF8829 Python driver	`https://github.com/ams-OSRAM/tmf8829_driver_python`, MIT licensed; the source of the RAM application image and of the frame structure definitions
MIPI I3C Specification	MIPI Alliance
Binho Supernova documentation	`https://docs.binho.io`

2 Required equipment and software

2.1 Hardware

Table 4. Required equipment

Item
Binho Supernova USB host adapter
LIGHTRANGER 14 Click, or another board carrying a TMF8829
USB-C cable

2.2 Connections

Table 5. Signals used

Signal	Purpose
SCL	I3C clock, adapter to target
SDA	I3C data, bidirectional
3.3 V	Target supply
GND	Return

Two wires and a supply. No interrupt pin, reset line or strapping resistor is connected for anything in this document.

> **NOTE** no external pull-up resistors were fitted for any procedure in this document. I3C runs its data phases push-pull, and the adapter drives the open-drain phases that dynamic address assignment and the broadcast command codes need. Enumeration, the firmware download and every frame read in Section 11 were measured on a bus with nothing on SCL and SDA but the adapter and the target.

2.3 Bus settings

Table 6. Bus settings used

Setting	Value
Push-pull rate	5 MHz at 50% duty cycle
Open-drain rate	1 MHz
Drive strength	Fast mode
Bus voltage	3.3 V

2.4 Software

```
pip install binhosupernova
```

Python 3.12 and the `binhosupernova` package are the only dependencies.

The ranging firmware travels in the assets archive as `assets/firmware/tmf8829_application.hex`. It is the sensor vendor's, published under the MIT license, and that license accompanies it as `assets/firmware/LICENSES-MIT.TXT`. The current image is always available from `https://github.com/ams-OSRAM/tmf8829_driver_python`.

The renderer draws with block characters and 24-bit colour. On a terminal that cannot carry them it falls back automatically, so a plain console produces a coarser rendering rather than an error. Every listing in this document is the fallback, because a PDF cannot show colour.

Confirm the setup without any hardware attached:

```
python i3c_tof.py demo --mode 16x16 --style mono
```

3 First contact: enumerate the bus

```
python i3c_mems.py scan
```

```
1 target(s) on the bus at 3300 mV, PUSH_PULL_5_MHZ_50_DC, OPEN_DRAIN_1_MHZ

  dynamic address 0x08
    PID  02 10 53 CD 47 22   device id 0x53CD
    BCR  0x23   DCR 0x00
    HDR  claimed   IBI capable
    profile  none in this tool
```

Two common command codes did that. **RSTDAA** clears any dynamic address already assigned, so the bus starts from a known state. **ENTDAA** then runs dynamic address assignment: the controller broadcasts, every target arbitrates using its 48-bit provisional identifier, and each is handed an address in turn. With one part present, that address is `0x08`.

No address was configured anywhere, and the part's static address was neither used nor needed.

The bus characteristics register is worth reading closely, because it is a set of claims that the rest of this note tests.

Table 7. BCR 0x23, bit by bit

Bit	Value	Meaning
0	1	Maximum data speed limitation: the target has one, so a controller should ask before pushing the clock
1	1	IBI capable: the target can raise an in-band interrupt
2	0	No mandatory data byte with an IBI
5	1	Advanced capabilities, which on this part means HDR

Section 10 tests both against measured behaviour.

4 Application state and bootloader entry

Register `0x00` reports which of the part's three applications is executing. Reading the first three registers on a part fresh from a power cycle:

```
  target at 0x08, APP_ID 0x80 v33.0
```

Table 8. Application identifiers

Value	What is running	What it will do
`0x80`	Bootmonitor	Accepts bootloader commands: read and write RAM, write FIFO, start an application
`0x02`	ROM application	Reports its version, rejects measurement commands
`0x01`	RAM application	Ranges

Which one a part powers up in is selected by its fuses, through the `powerup_select` field of `ENABLE` when that field is left at its default of *no override*. The part measured here comes up in the bootmonitor, reporting bootloader version 33.0 — so on this board there is no ROM application to displace, and the first thing a host sees is already waiting for a download.

A part that comes up as `0x02` needs the step below; one that comes up as `0x80` does not. A part that reads `0x01` is running firmware a previous session downloaded, which a soft reset alone does not undo, so the step below applies to it as well.

The awkward part is that **neither application implements the bootloader command set**. A part running an application cannot be told to accept a download, because the command that would do so is not one it understands. The way in is through the host interface registers, which stay live whatever is executing. `powerup_select` is a two-bit field, and value 1 is what forces the bootmonitor:

```
PON            = 1 << 2          # keep the part powered while it reboots
FORCE_BOOTMON  = 1 << 4          # powerup_select = 1: ignore the fuses
SOFT_RESET     = 1 << 6

device.write(ENABLE, [PON | FORCE_BOOTMON])      # 0xF8
device.write(RESET,  [SOFT_RESET])               # 0xF7
```

Table 9. `ENABLE` bit 4–5, `powerup_select`

Value	Meaning
0	No override; the fuses select what runs
1	Force the bootmonitor, ignore the fuses, wait for host commands
2	Set the vector table to the base of RAM and run whatever is there

The reset drops the target off the bus, so it has to be enumerated again before anything else will work. The utility does that automatically and reports the result:

```
an application is running; forcing the bootmonitor
  bootloader at 0x08
```

The dynamic address came back the same here, but nothing guarantees that on a bus with several targets. Re-read it rather than assuming it.

5 Downloading the RAM application

The firmware ships as an Intel HEX file. Records contiguous with the previous one should be merged during parsing, so the download runs as a few long streams rather than one bus command per sixteen-byte record.

Each block is announced with a write-FIFO command carrying a destination address and a length **in 32-bit words**, then streamed to the FIFO register:

```
params = list(address.to_bytes(4, "little")) + list((len(block) // 4).to_bytes(2,
"little"))
device.command(BL_W_FIFO_BOTH, params)          # 0x45 to CMD_STAT
device.write(FIFO, block)                       # 0xFF
```

Because the length is expressed in words, a block has to be padded to a word boundary. A partial word cannot be described.

5.1 Downloading to both processors

The command above is `W_FIFO_BOTH` (`0x45`), not `W_FIFO` (`0x44`). The two differ in which processor the bytes are written to. The part has two, both require the image, and using the command that writes to one is not reported as an error at any point in the sequence.

A download with `W_FIFO` completes normally. Every block is acknowledged. `START_RAM_APP` is accepted and answers `OK`. Then:

```
START_RAM_APP -> OK; ENABLE 0x14 cpu_ready=False
regs0-3 FF 21 00 00
```

Register `0x00` reads `0xFF` and `cpu_ready` in `ENABLE` stays low. The first CPU started, the second had no firmware, and the part halted. Nothing in the download or the start reported a problem. The same sequence with `W_FIFO_BOTH`:

```
START_RAM_APP -> OK; ENABLE 0x94 cpu_ready=True
regs0-3 01 02 C8 00
```

The failure is intermittent across sessions rather than reproducible. RAM contents survive a soft reset, so a part that has run the application at any point retains a valid image in the second CPU, and a `W_FIFO` download over the top of it starts normally. The fault appears on the first cold start after a power cycle.

Two reads separate this from the other causes of a failed start. `ADDR_RAM` (`0x43`) sets a read pointer and `R_RAM` (`0x40`) returns bytes from it, which establishes whether the download landed. `ENABLE` reports `pon` and `cpu_ready`, which distinguishes a halted processor from a target that is no longer addressed; both return `0xFF` from register `0x00`.

Both the bootloader and the application use the same command handshake, and it is worth understanding once. `CMD_STAT` reads back as the command while it is executing, and as a status code when it is done. So the host polls for *anything other than the command it sent*, rather than for a particular value:

```
while time.monotonic() < deadline:
    status = self.read_u8(CMD_STAT)
    if status is not None and status != code:
        return status
```

Polling for `STAT_OK` instead would hang on any command that reports something else, and would race on any command that finishes before the first read.

```
python i3c_tof.py bringup --firmware firmware/tmf8829_application.hex
```

```
  target at 0x08, APP_ID 0x80 v33.0
  downloading 14,444 bytes in 1 segment(s)
    14,444 bytes in 0.64 s (22.0 kB/s)
  RAM application v2.200 running
  8x8 configured, result format 0x01
  measuring
  frames report layout 0x01: 1 peak(s) per zone
  frame 2  8x8  62/64 zones (97%)  near 128 mm  median 452 mm  far 5462 mm  0 multi-object
20 C
  the part is measuring; run 'live' to watch it
```

Fourteen kilobytes in under seven tenths of a second, at 256 bytes per bus command, into both processors. This happens at every power-up, so it is part of the boot time of any product built around the part.

Starting the application restarts the target, which drops it off the bus. It has to be enumerated again before its registers can be read; skipping that leaves every read returning `0xFF`, which looks exactly like the halt described above and is not the same thing.

6 Choosing a zone grid and starting a measurement

The sensor carries a preconfigured page for each zone grid, so selecting a resolution is one command rather than a table of register writes. The command loads the page into registers `0x22` upward, where anything worth overriding can be overridden before it is written back.

Table 10. Zone grids

Grid	Command	Zones	Frames per measurement
8x8	`0x40`	64	1
8x8 long range	`0x41`	64	1
8x8 high accuracy	`0x42`	64	1
16x16	`0x43`	256	1
32x32	`0x45`	1024	2
48x32	`0x47`	1536	2

The last column is not a detail. Section 7.4 is about it.

Two page registers matter for everything that follows:

```
CFG_PERIOD_MS     = 0x22        # u16, measurement period
CFG_RESULT_FORMAT = 0x2A        # what each zone contains
```

`CFG_RESULT_FORMAT` is the one with consequences. It sets how many objects each zone reports and which optional fields come with them, and it is charged per zone on every frame:

Table 11. Result format, bit by bit

Bits	Field	Cost
0–2	Objects reported per zone, 1 to 4	3 bytes each, or 5 with signal strength
3	Signal strength per object	+2 bytes per object
4	Noise floor per zone	+2 bytes
5	Crosstalk estimate per zone	+2 bytes

At one object and no optional fields, a zone is 3 bytes: `1 * (3 + 0) + 0 + 0`. At four objects with every optional field, it is 24: `4 * (3 + 2) + 2 + 2`. On a 48x32 grid that is the difference between 4674 bytes and 36,930 bytes per measurement, so the format is a bandwidth decision rather than a preference.

Writing the page back and starting the measurement is one command, and its success code is not the one every other command uses:

```
status = device.command(CMD_WRITE_PAGE_AND_MEASURE)   # 0x14
# STAT_ACCEPTED (0x01), not STAT_OK (0x00)
```

A cyclic measurement reports `STAT_ACCEPTED` and keeps running until it is stopped. Every other command on the part reports `STAT_OK`, so code that checks for it will read a running measurement as a failed one.

7 Reading a frame

7.1 Frames come from a FIFO

There is no distance register. A completed measurement sets bit 0 of `INT_STATUS` (`0xE1`), and the frame is read out of a FIFO.

The read does not start at the FIFO. It starts at the FIFO status register, `0xFA`, because the register pointer auto-increments and the four systick registers sit between `0xFA` and the FIFO data register at `0xFF`. Those five bytes arrive ahead of the frame and are not part of it, which is why the decoder takes a pre-header length:

```
0xFA  FIFOSTATUS  ─┐
0xFB  SYSTICK_0    │  five bytes that precede every frame
0xFC  SYSTICK_1    │  and belong to none of them
0xFD  SYSTICK_2    │
0xFE  SYSTICK_3   ─┘
0xFF  FIFO         —  the frame itself, and it does not auto-increment past here
```

`INT_STATUS` is write-1-to-clear, and it should be cleared *before* the frame is read rather than after. The sensor is free-running: clearing afterwards can discard the flag for a frame that arrived while the previous one was still being transferred.

7.2 The layout byte sizes every zone

A frame is sixteen bytes of header, a body of zones, and twelve bytes of footer.

Table 12. Frame header

Offset	Field	Notes
0	`id`	High nibble `0x1` marks a result frame; low nibble is the focal-plane mode
1	`layout`	Same encoding as `CFG_RESULT_FORMAT`, plus a subframe flag in bit 6
2–3	`payload`	Length of everything after this field
4–7	`fNumber`	Running frame index
8–10	`temperature`	Degrees Celsius, signed
11	`bdv`	Breakdown voltage used
12–15	`refPos`	Optical reference peak positions

The `layout` byte has to be decoded before anything else, because every byte after the header is positioned by it. A zone is between 3 and 24 bytes wide, and reading it at the wrong stride does not fail — it produces a full frame of plausible, wrong numbers.

The header also carries its own length, which is the more robust way to size the read:

```
size = PRE_HEADER + 4 + payload      # the payload field counts everything after itself
```

Computing the size from the configured mode and comparing it against what the frame reports is a cheap and worthwhile check. When the two disagree, the configuration that was written is not the one the sensor is measuring under.

Two more details, each of which produces a wrong answer rather than an error:

Distances are in quarter millimetres. That is the 0.25 mm resolution the part advertises. Read as millimetres, a wall at 2.7 m appears at 11 m, and nothing about the result looks malformed.

Except when they are not. Bit 0 of the footer's reserved byte means the firmware already converted to millimetres. The footer therefore has to be read *before* the zones are scaled, which is the reverse of the order the bytes arrive in.

Table 13. Frame footer

Offset	Field
0–3	`t0Integration`
4–7	`t1Integration`
8	`frameStatus`, bit 0 set means valid
9	`reserved`, bit 0 set means distances are already in millimetres
10–11	`eof`, always `0xE0F7`

Both `frameStatus` and `eof` are worth checking, and they check different things. The end-of-frame marker says the frame came out of the FIFO intact. `frameStatus` says the sensor considers the measurement behind it usable.

7.3 One frame can be larger than one transfer

A 32x32 frame is 1569 bytes. Reading it in one call returns nothing at all:

```
error: short frame read: 0 of 1569 bytes
```

Not a truncated read — an empty one, with the frame still sitting in the sensor's FIFO. A binary search over the read length finds the boundary exactly:

```
    largest successful read: 1024 bytes
    request  1024 ->  1024 bytes ok
    request  1025 ->     0 bytes SHORT
```

This boundary belongs to the host adapter rather than to the sensor, so a different controller should be measured rather than assumed to share it. What the target itself declares is a different number again, and it is worth reading before sizing anything around it:

```
GETMRL -> 01 FF      maximum read length 511 bytes
GETMWL -> FF FF      maximum write length 65535 bytes
```

The target declares 511 bytes and does not enforce it: an 801-byte read of a complete 16x16 frame succeeds and decodes. So neither number is a reliable guide on its own. The declared length understates what works, the adapter's limit is what actually bounds a transfer, and the combination is what the chunked read below is sized against.

A frame above the adapter's limit therefore has to be read in pieces, and the FIFO is what makes that possible. The first read starts at `0xFA` as before; every read after it starts at `0xFF`, which does not auto-increment, so reading it repeatedly continues the same frame rather than restarting it:

```
register = FIFOSTATUS if taken == 0 else FIFO
chunk = self.read(register, min(MAX_READ, size - taken))
```

Every grid above 16x16 needs this on every frame.

7.4 The two largest grids arrive interleaved

A 32x32 or 48x32 measurement is more data than one frame carries, so the sensor sends sixteen rows at a time and flags which half in bit 6 of the layout byte.

The two halves are **not** the top and the bottom of the image. The first carries the even rows and the second carries the odd ones:

```
frame with subframe flag 0  ->  image rows 0, 2, 4, 6, ...
frame with subframe flag 1  ->  image rows 1, 3, 5, 7, ...
```

Stacking them rather than interleaving them produces a result that is wrong without being detectably malformed: the scene appears twice, one above the other, at half the vertical resolution.

```
::..          ########%%%%%%%%%@      <- the same scene
:::...      %%#%#######%%%%%%%%@@
::::..  ##%%%%%%%%%%%%%%%%%%%%%@
:::::...  .#########%%%%%%%%%%%%%
::::.......
::::.......
 :....      ##########%%%%%%%%%@      <- and again, sixteen rows down
:::...  #%%%%%%%%%%%%%%%%%%%%%@@
::::..    %%%%%%%%%%%%%%%%%%%%%
:::::... ..  ########%%%%%%%%%%
::::.......
::::.......
```

Ordering by the subframe flag rather than by arrival makes a dropped or duplicated frame a detectable error rather than a silently incorrect result.

8 Rendering a measurement

A grid of distances printed as numbers contains everything the sensor reported, in a form that shows spatial structure poorly. Three properties of the renderer address that.

Two zones per character cell. A terminal cell is about twice as tall as it is wide, so one cell per zone renders a square grid as a stretched rectangle. Drawing the upper half block in one colour and letting the background show through the lower half puts two vertically adjacent zones in one cell. The aspect ratio comes out right and a 32x32 image fits in sixteen terminal rows.

Confidence is drawn, not printed. Every zone carries a signal-to-noise ratio, and a low-confidence reading drawn identically to a high-confidence one is actively misleading. Colours are mixed toward the background in proportion to confidence, so uncertain readings recede and the eye lands on the ones worth trusting.

The colour scale is rendered with the image. A depth image cannot be read as distances without knowing what the ramp is mapped to. The range is taken from the frame itself, at the 2nd and 98th percentiles rather than the extremes, so a single outlying reading does not compress the rest of the scale.

```
python i3c_tof.py frame --mode 16x16 --style mono --histogram
```

```
  frame 2  16x16  256/256 zones (100%)  near 126 mm  median 369 mm  far 3536 mm  0 multi-
object  23 C

  #########%%%%%%
  #########%%%%%%
  ########%%%%%%%
  #########%%%%%%
  ########%%%%%%%
  #######%%%%%%%%
  ######%%%%%%%%%
    ####%%%%%%%%@
    ###%%%%%%%%%%
    ##%%%%%%%%%%@
    ##%%%%%%%%%%@
     #%%%%%%%%%%@
     ##%%%%%%%%%@
     ##%%%%%%%%%@
      %%%%%%%%%%
     %%%%%%%%%%%

      13 cm ████████████████████████████████████████ 150
      41 cm ████████████████████ 82
      69 cm
      98 cm
     126 cm
     155 cm
     183 cm
     212 cm
     240 cm ▌ 6
     268 cm ████▌ 17
     297 cm
     325 cm  1
```

Near is dense, far is sparse, and a zone with no return is blank. The gradient across the frame is a surface angled away from the sensor; the blank wedge at the left is where it passes out of range.

The histogram reports the distribution of returns by distance, which the spatial view does not show: two clusters below half a metre holding 232 of the 256 zones, and 24 returns between 2.4 and 3.3 m.

A synthetic scene generator is included so the display can be developed and checked without hardware:

```
python i3c_tof.py demo --mode 32x32 --scene corner
```

It generates a wall, an object in front of it, edges where a zone straddles both, and returns that weaken with distance. That matters more than it sounds: a renderer that looks good on random numbers can be unreadable on a real scene, because real depth data has structure.

9 More than one object per zone

A zone looking at the edge of a nearby object sees past it. The part reports up to four objects per zone, ordered by distance, and a renderer that shows only the nearest hides one of the more useful things the sensor does.

```
python i3c_tof.py frame --peaks 4 --signal --noise --xtalk --style numeric
```

```
  8x8 configured, result format 0x3C

  frame 2  8x8  64/64 zones (100%)  near 136 mm  median 358 mm  far 568 mm  15 multi-
object  24 C

     49   47   57   53   18   15   14   14
     49   47   51   49   18   15   14   14
     53   50   48   44   20   17   16   14
     56   49   44   40   32   19   18   14
     42   44   41   38   32   21   21   14
     43   44   40   36   32   23   23   14
     42   43   41   36   32   26   25   14
     38   41   42   37   34   31   28   14
```

Fifteen of the sixty-four zones reported more than one surface. In the colour renderer those cells are drawn with a shaded glyph instead of a solid one, so the capability is visible rather than described.

The cost is in the result format byte: `0x3C` is four objects with signal, noise and crosstalk, which is 24 bytes per zone against the 3 the default format uses. On an 8x8 grid that is 1536 bytes against 192. On 48x32 it is 36,864 bytes against 4608.

10 Declared capabilities against measured behaviour

Section 3 read two capability claims from the BCR. Both were tested against observable behaviour rather than against result codes.

10.1 In-band interrupts: capable, not used

Bit 1 of the BCR says the target can raise an in-band interrupt, which would let a completed measurement announce itself on the data line instead of being polled for. With the controller configured to accept IBIs and the sensor free-running at 100 ms:

```
  IBIs in 3 s with the sensor free-running: 0
  frames signalled through INT_STATUS in the same 3 s: 31
```

Thirty-one frames arrived. None of them announced itself. The sensor's firmware signals frame completion through its `INT_STATUS` register and its interrupt pin, and does not raise an IBI to do it — a reading consistent with the vendor's own driver, which polls the register and contains no IBI support at all.

This is worth stating plainly rather than treating as a fault. The BCR describes what the silicon's bus interface can do; it does not promise that the application firmware running above it uses the capability. Polling a register at 100 Hz to catch a 10 Hz frame costs one byte per poll and is entirely workable. But a design that was counting on interrupt-driven frame collection over two wires should confirm it against the firmware revision it will ship.

10.2 HDR-DDR: implemented, and not documented

Bit 5 claims advanced capabilities, which in I3C means the target supports at least one high data rate mode. Two published sources describe this part's bus interface, and neither of them mentions one.

The datasheet's interface section, DS001140 v2-00, states that the device "provides I3C serial communication interface with clock speed up to 12.5 MHz" and that "after the Dynamic Address Assignment, SDR Mode is selected for private messaging". That is a correct description of the state a target is in after address assignment, since HDR is entered explicitly. But across the document there is no occurrence of HDR, of double data rate, or of high data rate in any form. The vendor's published Python driver is the same: its only I3C-specific code re-assigns a dynamic address, and it contains no HDR entry, no DDR transfer and no reference to either.

So the capability bit and the documentation disagree, and the bench agrees with the bit. What follows was established by measurement rather than read anywhere, in two parts.

The DDR command byte is a word address. In a DDR read the command byte is not a register address and not an opaque selector. Sweeping every command from `0x80` to `0xFF` walks the register map in steps of two:

```
register = (command - 0x80) * 2
```

So command `0x80` reads from register `0x00`, and command `0xFD` reads from register `0xFA` — the FIFO status register, which is exactly where a frame read starts.

Each 16-bit word arrives byte-swapped relative to an SDR read of the same registers. That is not an endianness preference to be argued about; unswapping is what makes the two paths return identical bytes:

```
SDR read of 0x00..0x07:           01 02 C8 00 00 00 00 00
DDR, raw                          02 01 00 C8 00 00 00 00
DDR, byte-swapped                 01 02 C8 00 00 00 00 00   <- matches SDR
```

With both accounted for, an entire depth frame reads over HDR-DDR and decodes correctly. Forty frames were read alternating between the two paths, and all forty decoded valid on both — the same zones, byte for byte, whether the transfer ran at single or double data rate:

```
python i3c_tof.py frame --mode 16x16 --hdr
```

```
  frame 2  16x16  256/256 zones (100%)  near 129 mm  median 302 mm  far 2796 mm  0 multi-
object  21 C
```

HDR-DDR is declared by BCR bit 5 more often than it is implemented, and it is relevant to this class of part because the payload is kilobytes per measurement and scales with the zone count. On this target the mode is implemented and carries a complete measurement frame.

One consequence of the addressing is worth knowing. A DDR command addresses a 16-bit word, so no command lands on the FIFO data register at `0xFF` on its own. Continuing a DDR frame read from command `0xFF`, which addresses register `0xFE` and reaches the FIFO one byte later, was tried and does not reconstruct the frame, so a DDR read cannot be chunked the way the SDR one is. That confines it to grids whose entire frame fits in a single transfer: 8x8 and 16x16. The utility refuses the rest rather than returning something wrong:

```
error: a 32x32 frame is 1568 bytes, over the 1024-byte transfer limit,
       and a DDR read cannot be chunked
```

For grids above that, the SDR path with chunked reads from Section 7.3 is what carries a full frame, and the two produce identical data.

11 Measured results

11.1 Test configuration

Table 14. Test configuration

Item	Value
Adapter	Binho Supernova
Target	TMF8829 on LIGHTRANGER 14 Click
Firmware	RAM application v2.200
Bus	I3C SDR, 5 MHz push-pull, 1 MHz open-drain, 3.3 V, no external pull-ups
Result format	`0x01`: one object per zone, no optional fields
Measurement period	100 ms
Dynamic address	`0x08`

11.2 Size of a measurement, per grid

Table 15. Size of one complete measurement

Grid	Zones	Bytes per measurement	Transfers
8x8	64	225	1
16x16	256	801	1
32x32	1024	3138	4
48x32	1536	4674	6

Measured at one object per zone with no optional fields, which is the cheapest the part offers. The payload scales with the zone count, so the full-resolution grid moves twenty times the bytes of the default one, and the transfer count rises faster still because each frame above 16x16 crosses the transfer limit from Section 7.3 and arrives in two halves.

These figures are a property of the part and the result format, independent of the host reading them. The time they take depends on the bus clock and on the host.

Three levers move it, in the order worth trying:

Table 16. What to change if a full-resolution image is too expensive

Lever	Effect
Raise the push-pull clock	5 MHz was a conservative starting point, not a limit; I3C single data rate goes to 12.5 MHz
Trim the result format	Every optional field costs 2 bytes on every zone; at 1536 of them the format byte is the single largest lever
Lengthen the measurement period	Full resolution at 10 Hz is a demanding ask, and many applications do not need it

11.3 Feature summary

Table 17. What this target implements

Feature	Result
Dynamic address assignment	Yes; address `0x08` assigned by ENTDAA
Identification from the bus	Yes; PID `02 10 53 CD 47 22`, BCR `0x23`, DCR `0x00`
SDR register read and write	Yes
Declared maximum read length	`GETMRL` reports 511 bytes; not enforced, an 801-byte frame read succeeds
Declared maximum write length	`GETMWL` reports 65535 bytes
Largest transfer this adapter performs	1024 bytes; above it a read returns nothing rather than truncating. A property of the controller, not the target
HDR-DDR	Yes; complete measurement frames transferred and verified byte-for-byte against SDR. Not described in the datasheet or implemented in the vendor's driver
In-band interrupts	Declared in BCR; not raised by the ranging firmware
Zone grids	8x8, 16x16, 32x32, 48x32, plus long-range and high-accuracy 8x8 variants
Objects per zone	Up to 4, configurable

12 The utility

The reference utility is four files, split so that each one can be understood and tested on its own:

Table 18. Utility structure

File	Responsibility
`tof_frame.py`	Frame bytes to zones, and back. No hardware dependency
`tof_render.py`	Zone results to something a person can read
`tof_device.py`	The sensor: bootloader, firmware download, configuration, frames
`i3c_tof.py`	The command line, which only decides what to show

Nothing above `i3c_tof.py` knows about terminals, and nothing below it knows about the bus.

`tof_frame.py` encodes as well as decodes, which is what makes it testable without a sensor: a round trip through both directions exercises the real layout rather than a mock. All 32 combinations of object count and optional fields round-trip, and the independently computed frame size agrees with the length the frame reports for every one of them.

Table 19. Commands

Command	What it does
`demo`	Renders a synthetic scene; no hardware needed
`modes`	Lists the zone grids and what each one costs to read
`bringup`	Downloads the firmware and starts measuring, reporting each step
`frame`	Captures and prints a single image
`live`	Watches the sensor until interrupted

13 Practical notes

Download to both processors. `W_FIFO_BOTH`, not `W_FIFO`. The wrong one downloads cleanly, starts cleanly, and halts, and it can keep working for as long as a valid image survives in RAM from a previous session.

Re-enumerate after every reset, and after starting the application. Forcing the bootmonitor resets the part and starting the RAM application restarts it. Both drop it off the bus, and the dynamic address afterwards is not guaranteed to be the one it had.

`0xFF` from register 0x00 has two causes worth separating. The target is no longer addressed, or the target is addressed and its processor has halted. Reading `ENABLE` distinguishes them: it answers in the second case, and `cpu_ready` tells the rest of the story.

Read the layout byte before sizing anything. It is the difference between a decoded frame and a frame of plausible nonsense, and there is no error to catch if it is wrong.

Clear the interrupt before reading, not after. The sensor is free-running, and a flag set during a transfer is a frame that clearing afterwards discards.

Check the end-of-frame marker. `0xE0F7` at the end of every frame is a cheap way to catch a chunked read that lost synchronisation, and a mis-chunked frame is otherwise indistinguishable from a strange scene.

A frame that fails to read is still in the FIFO. An over-long read leaves the sensor holding the frame, so the next read returns the same frame rather than a fresh one. The utility counts dropped frames in `live` rather than stopping, because the next frame is already on its way.

14 Troubleshooting

Table 20. Troubleshooting

Symptom	Likely cause
`APP_ID` reads `0x02` and every command returns `STAT_ERR_UNKNOWN_CMD`	The ROM application is running; the RAM application has not been downloaded
Download and `START_RAM_APP` both report success, then `APP_ID` reads `0xFF` and `cpu_ready` is low	Downloaded with `W_FIFO` instead of `W_FIFO_BOTH`; the second processor has no firmware
Every register reads `0xFF`, including `ENABLE`	The target was not re-enumerated after a reset or after starting the application
Bootloader commands rejected	An application is running; force the bootmonitor through `ENABLE` and `RESET`
`short frame read: 0 of N bytes`	The read exceeded 1024 bytes; chunk it
The image shows the same scene twice	Subframes were stacked instead of interleaved
Every distance is roughly four times too far	Quarter-millimetre units read as millimetres
Distances are a quarter of what they should be	The footer's millimetre flag was set and the scale applied anyway
A decoded frame is full of plausible nonsense	The layout byte was not read, so the zone stride is wrong
Measurements arrive more slowly than the configured period	The host is not reading each frame before the next completes; see the measurement sizes in Section 11.2
`STAT_ACCEPTED` treated as an error	A cyclic measurement reports `0x01`, not `0x00`

15 Acronyms

Table 21. Acronyms

Term	Meaning
BCR	Bus characteristics register
CCC	Common command code
DAA	Dynamic address assignment
DCR	Device characteristics register
DDR	Double data rate
dToF	Direct time-of-flight
ENTDAA	Enter dynamic address assignment
FIFO	First in, first out buffer
HDR	High data rate
IBI	In-band interrupt
PID	Provisional identifier
RSTDAA	Reset dynamic address assignment
SDR	Single data rate
SPAD	Single-photon avalanche diode
SNR	Signal-to-noise ratio

16 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 RAM application image under `assets/firmware/` is not Binho's. It is © ams-OSRAM AG, redistributed here under the MIT license it was published with at `https://github.com/ams-OSRAM/tmf8829_driver_python`, and its license text travels beside it. The frame structure definitions used by `tof_frame.py` were taken from the same source.

Table 22. Contents of the assets archive

File	Description
`AN0007.md`	Markdown source of this document
`assets/i3c_tof.py`	Command line utility
`assets/tof_device.py`	Sensor bring-up and control
`assets/tof_frame.py`	Frame decoding and encoding
`assets/tof_render.py`	Terminal renderers
`assets/i3c_mems.py`	Shared I3C bus utility
`assets/firmware/tmf8829_application.hex`	TMF8829 RAM application, © ams-OSRAM AG, MIT licensed
`assets/firmware/LICENSES-MIT.TXT`	MIT license covering the RAM application
`assets/LICENSE`	MIT license covering the example code

17 Revision history

Table 23. Revision history

Rev	Date	Change
1.0	1 September 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. **I3C** and **MIPI** are trademarks or registered trademarks of MIPI Alliance, Inc. **ams**, **OSRAM** and **ams OSRAM** are trademarks or registered trademarks of ams-OSRAM AG. **mikroBUS** and **Click** are trademarks of MikroElektronika d.o.o.