ALP LAB

Multi-Processor Mailbox Communication

AN-007 · E1M-X V2N and V2N-M1 (CA55 ↔ CM33)

Document Number: AN-007 **Revision:** 0.7 **Date:** November 2026 **Status:** Preliminary

alplab.ai

© 2026 Alp Lab AB. All rights reserved.

1 Scope

Exchange messages between the **Cortex-A55 application cluster** (running Yocto Linux) and the **Cortex-M33 system-manager core** (running Zephyr) on the E1M-X V2N / V2N-M1 SoM. The two cores share a non-cacheable carve-out for the **RPMsg** (Remote Processor Messaging) OpenAMP virtio rings; the V2N hardware-mailbox doorbell provides the notify-on-write side channel.

This pattern is the Alp SDK™'s way to combine deterministic real-time control (motor loops, sensor sampling at 1 ms cadence) with rich application logic (UI, network, AI) on a single SoM.

> **Important: Read this before you plan around it.** On V2N today the two halves of this link are **not symmetric**, and this AN does not pretend otherwise. The A55 side is served by a real `<alp/rpc.h>` backend. The shipped M33 slice **deliberately bypasses `<alp/rpc.h>`** and speaks raw OpenAMP, because it was shaped to interoperate with Renesas's licence-gated `rpmsg_sample_client`. What is bench-proven on silicon is the **transport**; what does not exist yet is a portable `<alp/rpc.h>` backend on the M33. **Section 2** states exactly what you can run today.

Audience	Firmware engineers building heterogeneous-compute applications.
Prerequisites	V2N or V2N-M1 SoM, **QS-E1M-X-EVK-001** completed for **both** core targets (A55 and M33). Working knowledge of OpenAMP / RPMsg semantics — on the M33 side you are writing OpenAMP code directly, not a portable SDK call.
Outcome	An attached A55 ↔ M33 OpenAMP link with a proven byte-exact echo round-trip. The A55 drives it through the production `<alp/rpc.h>` API; the M33 runs a raw-OpenAMP echo responder. Round-trip latency is not yet characterised.
Time	45 minutes.
Source	`docs/heterogeneous-builds.md`, `examples/multicore/rpmsg-v2n/`, and `examples/multicore/mproc-mailbox/` in alp-sdk.

Table 1 Scope summary

2 What Works Today

Capability	State	Detail
Raw A55 ↔ M33 UIO / OpenAMP transport	**Bench-proven on silicon**	Proven on E1M-X V2N-M1: the A55 attaches to the already-running M33's published resource table, and attach plus a byte-exact echo round-trip (1, 4, 16, and 64 B) passes end-to-end over the real MHU doorbell and real shared-memory vrings. The concurrent-close use-after-free hardening (GHSA-xhm8-7f87-93q5) is proven on the same bench. Exercised by `tests/yocto/rpc_uio_bench_main.c`, which links the real dispatcher and drives the public `<alp/rpc.h>` API — not a test double.
`<alp/rpc.h>` on the **A55**	**Available**	`src/backends/rpc/yocto_uio_drv.c` binds the RPC dispatcher to userspace OpenAMP / libmetal over `/dev/uio*` in ATTACH mode. It is registered for V2N silicon specifically and outranks the generic `/dev/rpmsg*` backend on this SoC; every other Linux target keeps the chardev backend. So `alp_rpc_open()` / `_call()` / `_send()` / `_subscribe()` / `_close()` are the real API on the A55.
`<alp/rpc.h>` on the **M33**	**Not available**	The shipped M33 slice bypasses it by design and hand-wires OpenAMP against the board devicetree. A V2N-specific `<alp/rpc.h>` M33 backend is explicitly deferred follow-up work. **Budget for writing raw OpenAMP on the**

Copyright © 2026 Alp Lab AB

Capability	State	Detail
		M33, or keep the M33 slice's protocol surface small enough to migrate later.
The `rpmsg-v2n` example, end to end	**Matched pair**	Its `linux/` half now drives the M33 echo responder's fixed endpoint through `<alp/rpc.h>`: four `alp_rpc_call()` round trips on the `echo_test` method, each checked byte for byte. The example README carries the bench procedure (CM33 image build, flash, run) and a HIL spec; the pass line is `[rpmsg-v2n] done (4/4 round trips verified)`. See **Section 4**.
Link-loss reporting on V2N	**Not available**	The V2N UIO backend reports `ALP_RPC_LINK_UP` only; it is not wired to report `ALP_RPC_LINK_LOST`. A CM33 reset or crash is not surfaced through `alp_rpc_link_state()` on this SoC — use the liveness beacon's heartbeat (**Section 5**) if you need to detect it.

Table 2 V2N A55 ↔ M33 link: state as of SDK v0.17.0

3 Architecture

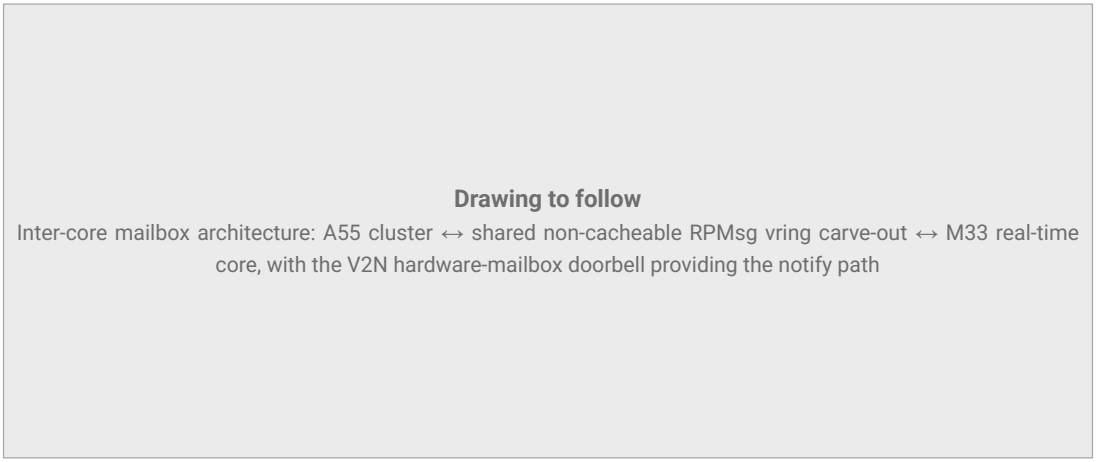


Figure 1 Inter-core mailbox architecture: A55 cluster ↔ shared non-cacheable RPMsg vring carve-out ↔ M33 real-time core, with the V2N hardware-mailbox doorbell providing the notify path

`<alp/rpc.h>` defines no pre-baked services: method names are **application-defined**. On the wire every RPMsg frame carries a single-line ASCII method header (`<method>\0`, up to 32 bytes including the NUL) followed by an opaque application-defined payload that the SDK passes through verbatim.

That framing is what makes the current asymmetric link usable at all. The M33 echo responder has **no method routing**: it reflects whatever bytes arrive, verbatim. Because an `<alp/rpc.h>` request frame is `<method>\0<payload>`, echoing the whole frame back returns a well-formed response **on the same method** — so an A55 `alp_rpc_call()` completes normally. It works, but understand what it is:

> **Warning: The M33 is not implementing your methods — it is reflecting them.** Any method name you call will "succeed" and return your own payload. That is a transport proof, not application logic. The moment your M33 slice must compute a distinct response, you write that responder against raw OpenAMP yourself; there is no `alp_rpc_subscribe()` on the M33 to hang it off.

On V2N the A55 slice's channel contract comes from the build-time-generated `<alp/system_ipc.h>` (endpoint IDs, mailbox channel, carve-out address), emitted by the orchestrator from the project's `ipc:` block. The M33 slice does **not** consume that header — it takes its addresses from the board devicetree and listens and replies on a **fixed** endpoint address, `1024`

Copyright © 2026 Alp Lab AB

(APP_EPT_ADDR in `m33_sm/src/main.c`), not the address the generated header would compute. The A55 must therefore set `dst_ept = 1024` explicitly and leave `src_ept = 0` (the backend then uses RPMSG_ADDR_ANY, and the M33 learns the A55's address from the first frame). The generated header is not yet the shared contract across both cores on this link; it is the A55's half of one.

4 Software Walkthrough

4.1 M33 side (real-time)

There is no <alp/rpc.h> code to show here, because the shipped M33 slice does not use it. `examples/multicore/rpmsg-v2n/m33_sm/src/main.c` is a from-scratch OpenAMP firmware written against the real RZ/V2N devicetree and the Renesas MHU mailbox driver. In outline it:

1. Publishes a **liveness beacon** into the resource-table region as its very first action, so a bench `devmem` read from Linux proves in one shot that the M33 is alive, the shared window is backed, and the M33 ↔ A55 address translation is correct.
2. Runs `metal_init()`, registers the shared-memory device with two regions (the RPMsg buffer pool and the resource table), and copies its resource table into the shared region for the A55's attach to parse.
3. Creates a virtio device in **device** (slave) role and blocks in `rproc_virtio_wait_remote_ready()` — the M33 is started by the boot chain before Linux, so it waits for the A55 to catch up. This is an **attach**-mode link: nothing loads an ELF into the M33 at runtime.
4. Creates one endpoint, `"rpmsg-service-0"`, and echoes every frame it receives back verbatim.

The doorbell is **asymmetric**, and this is the part most likely to surprise you: the forward A55 → M33 kick and the reverse M33 → A55 kick go through **different** mailbox units, because the reply path the mailbox driver would naturally use raises an interrupt that reaches no A55 interrupt line at all. The reverse kick therefore pokes a specific software-interrupt unit directly. If you write your own M33 responder, budget time for this: it is bench-measured behaviour, not something the vendor documentation hands you.

> **Note:** Two details in that firmware are load-bearing and worth carrying into your own responder. The receive endpoint is created **before** the receive loop starts, not from the responder thread — otherwise the A55's very first request lands with no matching endpoint and is dropped, and the first `alp_rpc_call()` times out. And received frames are queued rather than held in a single buffer, so two rapid frames cannot overwrite each other before the responder runs.

Build both slices from the one `board.yaml` with the orchestrator:

```
cd alp-workspace/alp-sdk/examples/multicore/rpmsg-v2n
# tan build finds board.yaml by walking up from the cwd (it takes no
# explicit app path), then fans out the M33-SM Zephyr image
# (BOARD=alp_e1m_v2n101_m33_sm) and the A55 Yocto slice from that one file.
tan build

# Bundle + flash, walking the preset's boot_order:
tan image
tan flash
```

4.2 A55 side (Yocto Linux)

This half **is** the portable surface, and this is the call the silicon proof exercises:

```
#include <alp/rpc.h>
#include <alp/system_ipc.h>   // generated by alp_orchestrate.py
#include <stdio.h>
#include <string.h>

int main(void)
{
    printf("[a55] echo client coming up\n");
```

Copyright © 2026 Alp Lab AB

```
    // The channel name and mailbox channel come from the generated
    // header. The M33 slice does NOT consume it: its echo responder
    // listens on the fixed address 1024 (APP_EPT_ADDR in
    // m33_sm/src/main.c), so dst_ept is set explicitly and src_ept is
    // left 0 (RPMSG_ADDR_ANY; the M33 learns it from the first frame).
    alp_rpc_channel_t *ch = alp_rpc_open(&(alp_rpc_config_t){
        .name    = ALP_IPC_ALP_DEFAULT_RPMSG_NAME,
        .src_ept = 0u,
        .dst_ept = 1024u,
        .mbox_ch = ALP_IPC_ALP_DEFAULT_RPMSG_MBOX_CH,
    });
    if (ch == NULL) {
        printf("[a55] alp_rpc_open failed: %d\n", (int)alp_last_error());
        return 1;
    }

    // The M33 echoes the frame verbatim, so the response payload comes
    // back byte-identical to the request. This proves the transport; it
    // is NOT the M33 computing a reply.
    const char req[]    = "hello";
    char       resp[64] = { 0 };
    size_t     resp_len = sizeof resp;

    alp_status_t rv = alp_rpc_call(ch, "echo_test",
                                   req, sizeof req,          // request
                                   resp, &resp_len,          // response
                                   /*timeout_ms*/ 3000);
    // After a timeout this channel refuses every later call with
    // ALP_ERR_NOT_READY: close it and open a new one before retrying.
    printf("[a55] echo rv=%d len=%zu match=%d\n",
           (int)rv, resp_len, memcmp(req, resp, sizeof req) == 0);

    alp_rpc_close(ch);
    printf("[rpmsg-v2n] done\n");
    return 0;
}
```

On V2N the Linux backend reaches the M33 over libmetal + OpenAMP user-space access to `/dev/uio*` in **attach** mode — not `/dev/rpmsg*`. V2N's Yocto BSP does not carry the mainline `rpmsg_char` remoteproc glue, only the raw generic-UIO regions a userspace OpenAMP master can attach to directly. The SDK selects this backend automatically on V2N silicon; every other Linux target keeps the `/dev/rpmsg*` backend unchanged. Your application code is the same `<alp/rpc.h>` calls either way.

5 Expected Output

A55 console for the snippet above — what the bench proves is a byte-exact round-trip (validated at 1, 4, 16, and 64 B):

```
[a55] echo rv=0 len=6 match=1
[rpmsg-v2n] done
```

The shipped `examples/multicore/rpmsg-v2n/linux` consumer prints one `[a55] recv echo_test[N]="ping-N" (verified)` line per round trip and passes with:

```
[rpmsg-v2n] done (4/4 round trips verified)
```

M33 console:

```
rpmsg_v2n_m33_sm: rpmsg-v2n/m33_sm: starting (alp-sdk #683, Path B Phase 1)
rpmsg_v2n_m33_sm: rpmsg-v2n/m33_sm: bringing up the rpmsg virtio device
rpmsg_v2n_m33_sm: rpmsg-v2n/m33_sm: Linux responder started
```

Note: Before debugging the link itself, confirm the M33 is alive: read the **liveness beacon** from Linux with `devmem` against the resource-table region's A55 alias. The beacon is the last 16 bytes of the `rsctbl` page, laid out in `<alp/protocol/amp_beacon.h>`: magic `0xA10D0683`, version (1 = RPC firmware without attach reset, 2 = with it, `0x100` and above = an image that serves no RPC), a 1 Hz heartbeat, and the attach epoch (odd = bound to a session, even = waiting for an attach). The magic proves the M33 booted, the shared window is backed and the address translation is correct; a **changing** heartbeat on a re-read distinguishes "still running" from "wrote once, then faulted". `alp_rpc_open()` reads the same words before it touches anything the CM33 reads.

6 Performance

Round-trip latency on this link is **not yet characterised**, and this note will not print a figure it cannot stand behind. The silicon work to date proves **correctness** — attach, byte-exact echo at 1/4/16/64 B, and safe concurrent close — not timing. Budget your own measurement against your own payload sizes and scheduler configuration.

Two structural points hold regardless of the measured number. The mailbox only ever carries a **wakeup**, never data: the payload is already in shared memory before the doorbell rings, so per-message cost is dominated by the vring and interrupt path rather than by payload size. And where the M33 samples at >= 10 kHz but only periodically reports, prefer fire-and-forget sends of **batched** payloads over per-sample request/response — a blocking `alp_rpc_call()` per sample pays the full round trip every time.

7 Troubleshooting

Symptom	Likely cause / fix
A55 `alp_rpc_open()` returns NULL	Read `alp_last_error()` — on the V2N UIO backend it names the cause: `ALP_ERR_NOT_READY` = no CM33 beacon (the CM33 is not running, or runs an image without the heartbeat beacon); `ALP_ERR_NOSUPPORT` = the CM33 image serves no RPC (beacon version `0x100` or above, the value reserved for the idle stock shim; a shim that publishes no beacon fails with `ALP_ERR_NOT_READY` instead); `ALP_ERR_BUSY` = the CM33 is still attached to an earlier session and cannot reset (version-1 firmware, so a second open in the same CM33 boot fails), or another process holds the link; `ALP_ERR_TIMEOUT` = the CM33 did not acknowledge the attach reset. Otherwise check the process can open `/dev/uio*` — run as root or grant read/write on those devices.
Second `alp_rpc_open()` in the same CM33 boot fails	With beacon-version-2 firmware every open after the first runs an **attach reset** (the A55 waits up to 500 ms for the CM33 to acknowledge). That path is not yet bench-verified: until it is, cold-cycle the board between runs. Version-1 firmware cannot reset at all and returns `ALP_ERR_BUSY`.
One call timed out, now every call returns `ALP_ERR_NOT_READY`	**Expected.** Replies are matched to calls by method name only (no sequence id), so after any `alp_rpc_call()` timeout the channel refuses every later call until it is closed and reopened — otherwise the late reply would be handed to the next call. `alp_rpc_close()` and `alp_rpc_open()` again; do not retry on the same handle.
`alp_rpc_open()` succeeds but every call times out	The M33 is blocked in its wait-for-remote-ready and never saw the attach, or the reverse doorbell is not reaching the A55. The reverse M33 → A55 kick uses a **different** mailbox unit from the forward path (see **Section 3**); a responder that rings the "natural" reply channel raises an interrupt no A55 line receives, so replies are silently never delivered.
The first call after a cold attach times out, later ones work	The responder endpoint is being created **after** the receive loop starts, so the A55's first frame arrives with no matching endpoint and is dropped. Create the endpoint before the loop.
Two rapid requests, only the second answered	A single receive buffer is being overwritten before the responder thread runs. Queue received frames instead.

Copyright © 2026 Alp Lab AB

Symptom	Likely cause / fix
Every call times out against the M33 echo responder	`dst_ept` is not the responder's fixed address. Set `dst_ept = 1024` and `src_ept = 0`; the generated `<alp/system_ipc.h>` endpoint IDs do not match the shipped M33 slice (see **Section 3**).
Channel comes up but no frames arrive, or data is stale	The RPMsg carve-out is non-cacheable by design and the SDK performs no cache maintenance (there is no per-channel cache setting). For `board.yaml ipc:` endpoints the generator emits `CONFIG_DCACHE=n` for every endpoint core; hand-written M33 firmware must map the carve-out non-cacheable in the MPU itself.

Table 3 Common failures

8 References

- **Canonical walkthrough**: `docs/heterogeneous-builds.md` in alp-sdk (end-to-end dual-OS A55 ↔ M33 build + RPMsg flow).
- **Examples**: `examples/multicore/rpmsg-v2n/` (V2N A55 ↔ M33) and `examples/multicore/mproc-mailbox/` (single-SoC AEN M55-HP ↔ M55-HE) in alp-sdk. `rpmsg-v2n`'s README carries the bench procedure for one RPC round trip over the UIO backend, and its two slices are a matched pair: the `linux/` half round-trips `echo_test` against the M33 echo responder's fixed endpoint. Its `m33_sm/testcase.yaml` is `build_only: true` (the firmware needs real RZ/V2N devicetree plus the vendor MHU / FSP / OpenAMP modules), so CI proves it **compiles**, not that it runs.
- **Beacon layout**: `<alp/protocol/amp_beacon.h>` in alp-sdk.
- **Silicon proof**: `tests/yocto/rpc_uio_bench_main.c` — the A55 attach + echo bench binary. It links the real dispatcher and drives the public `<alp/rpc.h>` API against the live M33, so it is the closest thing to a working end-to-end demo of this link.
- **SDK API**: `<alp/rpc.h>` (framed RPMsg) plus the generated `<alp/system_ipc.h>`; lower-level mailbox / shmem / hwsem primitives in `<alp/mproc.h>`. V2N A55 backend: `src/backends/rpc/yocto_uio_drv.c`.
- **Companion tutorial**: `docs/tutorials/15-mproc-mailbox.md` in alp-sdk – the `<alp/mproc.h>` mailbox primitives on AEN.
- **V2N HW Design Guide** §6.7 (RZ/V2N internal interconnect / DMA / mailbox).

9 Revision History

Revision	Changes	Date
0.1	• Initial draft.	May 2026
0.2	• Re-based on the current alp-sdk: ▸ high-level IPC moved to `<alp/rpc.h>` (`alp_rpc_open` / `_call` / `_send` / `_subscribe`) with `<alp/mproc.h>` as the mailbox/shmem/hwsem primitive layer ▸ corrected example paths to `examples/multicore/rpmsg-v2n/` and `.../mproc-mailbox/` ▸ board target `alp_e1m_v2n101_m33_sm` ▸ `west alp-build` orchestrator flow ▸ replaced the fictional `alp.mproc.*` endpoints / CBOR framing and the Python `/dev/alp_mproc0` path with the real method-framed RPMsg surface over `/dev/rpmsg*`.	June 2026
0.3	• Corrected the portable-RPC story: the shipped `rpmsg-v2n` M33 slice deliberately bypasses `<alp/rpc.h>` and speaks raw OpenAMP (its peer is a licence-gated Renesas sample), so the two slices are not a matched pair. • The A55 side of `<alp/rpc.h>` is real and bench-proven over UIO; the gap is M33-only. • Replaced a fabricated M33 walkthrough that called `alp_rpc_open`/`alp_rpc_subscribe` on a slice that does not implement them, removed fabricated console output and latency figures, and explained why the echo appears to answer any method. • Corrected `/dev/rpmsg*` to `/dev/uio*` on V2N. • Fixed the Alp SDK repository links (`alpDevs` → `alplabai`).	July 2026

Copyright © 2026 Alp Lab AB

Revision	Changes	Date
0.4	• Retired the `west alp-*` build commands: alp-sdk v0.12.0 removed the SDK build executor (ADR-0020 Phase 4), so `west alp-build` → `tan build`, `west alp-image` → `tan image`, `west alp-flash` → `tan flash`. • `tan build` finds `board.yaml` by walking up from the cwd and takes no explicit app path, so the sequence now `cd`s into the example first; it has no per-core `--core` flag, so the separate M33-slice iterate command was dropped.	July 2026
0.5	• Re-verified against alp-sdk v0.17.0. • Added the new V2N UIO `alp_rpc_open()` failures (`ALP_ERR_BUSY`, `ALP_ERR_TIMEOUT`, `ALP_ERR_NOT_READY` for a missing CM33 beacon, `ALP_ERR_NOSUPPORT` for a CM33 image that serves no RPC), the attach reset (not yet bench-verified) and the rule that one `alp_rpc_call()` timeout leaves the channel returning `ALP_ERR_NOT_READY` until it is reopened. • Documented the beacon layout from `<alp/protocol/amp_beacon.h>`. • The `rpmsg-v2n` example is now a matched pair (its `linux/` half round-trips `echo_test`), so the "not runnable as shipped" row, troubleshooting entry and README caveat were replaced. • Corrected the A55 snippet to address the M33 responder's fixed endpoint `1024` (`src_ept = 0`). • Replaced the cacheability troubleshooting advice (the carve-out is non-cacheable by design; `alp_rpc_config_t.cacheable` was removed) and noted that the V2N UIO backend does not report link loss.	October 2026
0.6	• Release history: every entry formatted as a bullet list.	October 2026
0.7	• Product name on page 1 links to the product page on alplab.ai.	November 2026

Table 4 Revision History

Copyright © 2026 Alp Lab AB