ALP LAB

Multi-Processor Mailbox Communication

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

Document Number: AN-007 **Revision:** 0.2 **Date:** June 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 (the V2N preset's `ocram_low` region by default) for the **RPMsg** (Remote Processor Messaging) OpenAMP virtio rings; the V2N hardware-mailbox doorbell provides the notify-on-write side channel. The SDK exposes this through the framed-RPC surface in `<alp/rpc.h>`, which sits on top of the mailbox + shared-memory primitives in `<alp/mproc.h>`.

This pattern is the SDK's canonical 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.

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). Familiarity with OpenAMP / RPMsg semantics is helpful but not required.
Outcome	A request/response ping-pong: the A55 issues a `ping` request carrying `seq = N` via `alp_rpc_call()`, and the M33 reflects it back on the same `ping` method with its local timestamp `t_us`. Round-trip latency <= TBD µs under load.
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 Architecture

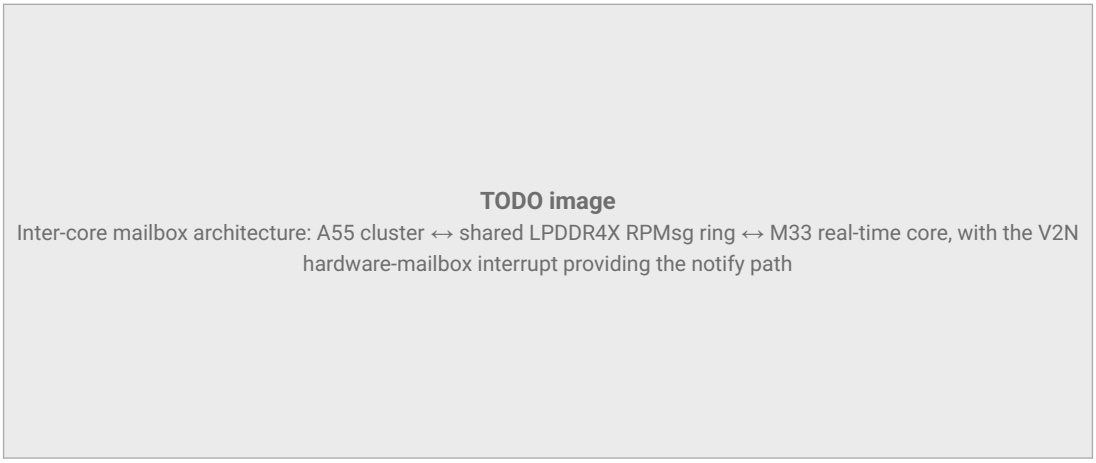

Figure 1 Inter-core mailbox architecture: A55 cluster ↔ shared LPDDR4X RPMsg ring ↔ M33 real-time core, with the V2N hardware-mailbox interrupt providing the notify path

`<alp/rpc.h>` defines no pre-baked services: method names are **application-defined**. Both core slices share one channel contract through the build-time-generated `<alp/system_ipc.h>` (endpoint IDs, mailbox channel, and carve-out address), emitted by the orchestrator from the project's `ipc:` block. This note uses the methods below:

Method	Purpose
`ping`	Request/response health check. The A55 invokes it with `alp_rpc_call()`; the M33 reflects it with `alp_rpc_send()`. Used by **AN-007** below.
`temperature`	Fire-and-forget M33 → A55 stream of samples, one per `alp_rpc_send()`. This is the method the `rpmsg-v2n` example ships.

Table 2 Methods used in this note

Customers register additional methods simply by passing fresh names to `alp_rpc_subscribe()` / `alp_rpc_send()`. 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.

Copyright © 2026 Alp Lab AB

3 Software Walkthrough

3.1 M33 side (real-time)

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

static alp_rpc_channel_t *ch;

// A55 -> M33 "ping": reflect it straight back on the same method with
// the M33's local timestamp. Runs on the SDK's RPMsg RX worker.
static void on_ping(const void *payload, size_t len, void *user)
{
    (void)user;
    uint32_t seq = 0;
    if (len == sizeof seq) memcpy(&seq, payload, sizeof seq);

    uint32_t t_us = k_cyc_to_us_floor32(k_cycle_get_32());
    alp_rpc_send(ch, "ping", &t_us, sizeof t_us);
    printf("[m33] ping[%u] reflected t_us=%u\n", seq, t_us);
}

int main(void)
{
    printf("[m33] rpmsg-v2n responder coming up\n");

    // Every id/address comes from the generated header; nothing is
    // hand-coded. The peer (A55) opens the mirror of this contract.
    ch = alp_rpc_open(&(alp_rpc_config_t){
        .name    = ALP_IPC_ALP_DEFAULT_RPMSG_NAME,
        .src_ept = ALP_IPC_ALP_DEFAULT_RPMSG_SRC_EPT,
        .dst_ept = ALP_IPC_ALP_DEFAULT_RPMSG_DST_EPT,
        .mbox_ch = ALP_IPC_ALP_DEFAULT_RPMSG_MBOX_CH,
    });
    alp_rpc_subscribe(ch, "ping", on_ping, NULL);
    for (;;) { k_msleep(1000); }   // RX is serviced on the SDK's worker
}
```

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

```
cd alp-workspace
# Fans out the M33-SM Zephyr image (BOARD=alp_e1m_v2n101_m33_sm) and
# the A55 Yocto slice from a single board.yaml.
west alp-build alp-sdk/examples/multicore/rpmsg-v2n

# Iterate on just the M33 slice (skips the long Yocto rebuild):
west alp-build alp-sdk/examples/multicore/rpmsg-v2n --core m33_sm

# Bundle + flash, walking the preset's boot_order:
west alp-image
west alp-flash
```

3.2 A55 side (Yocto Linux)

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

Copyright © 2026 Alp Lab AB

```c
#include <stdio.h>

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

    // The A55 mirrors the M33's endpoint IDs (src/dst swapped); every
    // value comes from the generated header, so neither side hand-codes
    // an address or an endpoint id.
    alp_rpc_channel_t *ch = alp_rpc_open(&(alp_rpc_config_t){
        .name    = ALP_IPC_ALP_DEFAULT_RPMSG_NAME,
        .src_ept = ALP_IPC_ALP_DEFAULT_RPMSG_DST_EPT,   // mirror of M33
        .dst_ept = ALP_IPC_ALP_DEFAULT_RPMSG_SRC_EPT,
        .mbox_ch = ALP_IPC_ALP_DEFAULT_RPMSG_MBOX_CH,
    });

    for (uint32_t seq = 0; seq < 10; ++seq) {
        uint32_t t_us = 0;
        size_t   resp_len = sizeof t_us;
        alp_status_t rv = alp_rpc_call(ch, "ping",
                                       &seq, sizeof seq,        // request
                                       &t_us, &resp_len,        // response
                                       /*timeout_ms*/ 100);
        if (rv == ALP_OK)
            printf("seq=%u  m33_t=%uus\n", seq, t_us);
        else
            printf("seq=%u  call failed rv=%d\n", seq, (int)rv);
    }

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

On the A55 the same `<alp/rpc.h>` surface links against the Yocto SDK sysroot; the Linux backend reaches the M33 over libmetal + librpmsg user-space access to `/dev/rpmsg*` (no custom kernel module). Both slices `#include <alp/rpc.h>` and `<alp/system_ipc.h>` – the generated header is the shared contract.

4 Expected Output

A55 console:

```
$ ./rpmsg-v2n-ping
[a55] ping client coming up
seq=0  m33_t=42135us
seq=1  m33_t=142398us
seq=2  m33_t=242502us
seq=3  m33_t=342654us
...
[rpmsg-v2n] done
```

M33 console (FTDI Channel B):

```
[m33] rpmsg-v2n responder coming up
[m33] ping[0] reflected t_us=42135
[m33] ping[1] reflected t_us=142398
[m33] ping[2] reflected t_us=242502
```

Copyright © 2026 Alp Lab AB

5 Performance

Typical round-trip latency under the SDK's default Linux scheduler is **TBD µs** (request leaves the A55 client → M33 RX worker fires → M33 replies via `alp_rpc_send()` → the A55's `alp_rpc_call()` returns). Under hard-real-time conditions (M33 isolated, A55 with `chrt -f 80`) it drops to **TBD µs**.

For applications where the M33 is sampling at >= 10 kHz and only periodically reports to the A55, prefer a fire-and-forget streaming method (`alp_rpc_send()`, as with the `temperature` method above) carrying batched payloads to minimise transport overhead.

6 Troubleshooting

Symptom	Likely cause / fix
M33 builds OK but A55 `alp_rpc_open()` returns NULL	The M33 firmware never reached its `alp_rpc_subscribe()` call, so the name-service handshake never registers the peer endpoint. Check the M33 boot console for an `alp_rpc_open()` failure earlier (`alp_last_error()`).
Requests time out under load	The A55 process lacks access to `/dev/rpmsg*`. Run as root or grant the user read/write on the rpmsg char device.
Replies have wildly incorrect time-stamps	M33's `k_cycle_get_32()` wrapped during the measurement; use `k_uptime_get()` for windows > 25 s on the 200 MHz M33.
Channel comes up but no frames arrive	DMA-capable carve-out not declared non-cacheable. Open the channel with `cacheable = false` (the v0.6 default) so the backend keeps the virtio rings coherent, or confirm the orchestrator placed the carve-out in the preset's non-cacheable region (`ocram_low` on V2N).

Table 3 Common failures

7 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.
- **SDK API**: `<alp/rpc.h>` (framed RPMsg) plus the generated `<alp/system_ipc.h>`; lower-level mailbox / shmem / hwsem primitives in `<alp/mproc.h>`.
- **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).

8 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

Table 4 Revision History

Copyright © 2026 Alp Lab AB