Multi Zone API

AMP/Linux Communication Protocol

Document Version Rev 1.0.0

August 10, 2019

Table 1: Version History

Version	Date	Changes
1.0	August 10, 2019	Initial Release

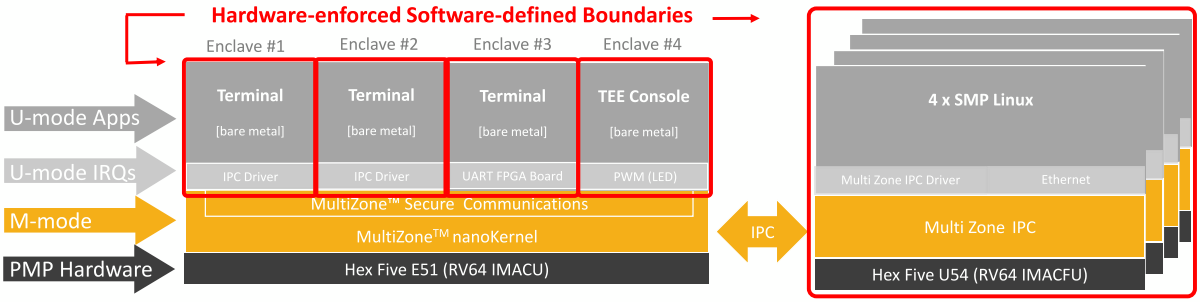
Contents

Chapter 1

Introduction

This technical note describes the specification of the multi zone API AMP/Linux communication protocol. This protocol is intended to be used on heterogeneous platforms, which typically are endowed with one microcontroller and multiple Linux-capable processors. Although the specification is generic enough to be implemented on any operating system (OS) targeting any heterogeneous platform, this document is mainly focused on the current MultiZone for Linux implementation (targeting the RISC-V-based U540 platform), whose system architecture is depicted in Figure 1.1.

Figure 1.1: MultiZone for Linux SDK: System Architecture

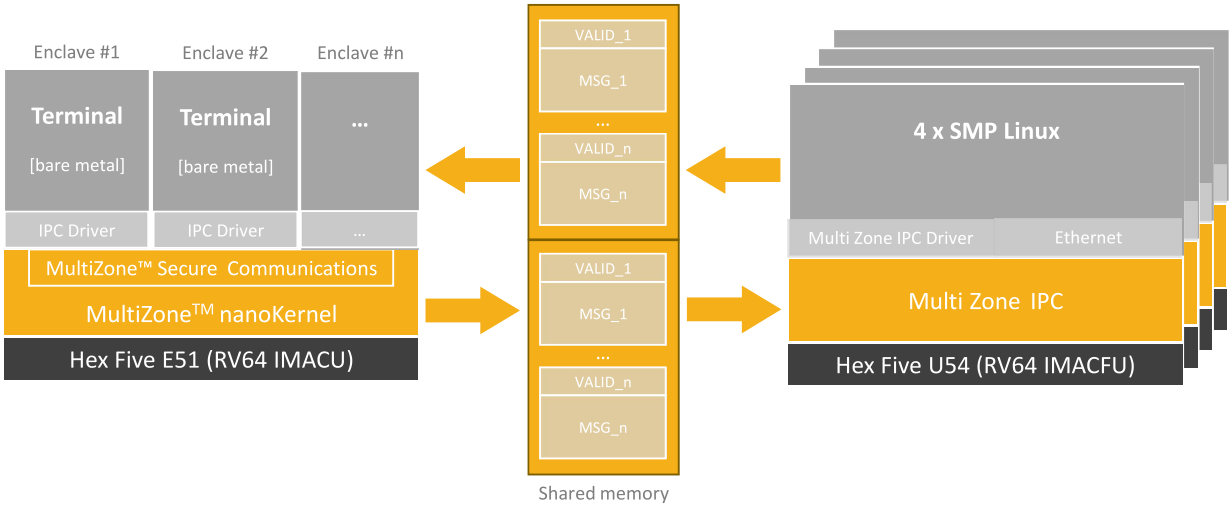


Chapter 2

Communication protocol

This chapter describes the specification of the multi zone AMP communication protocol. The protocol is based on four main invariants. These invariants specify that the communication data path is implemented trough a shared memory buffer which is split into 2 parts: MultiZone inbox and `Zone_0` inbox. These inboxes follow a specific layout, and its length depends on the number of zones running atop the MultiZone nanokernel. The communication flow specifies that access to inboxes requires synchronization to avoid race conditions for multi-thread and multi-core systems. Figure 2.1 illustrates how the protocol was designed on the MultiZone for Linux SDK.

Figure 2.1: MultiZone for Linux SDK: Communication Protocol



2.1 Invariants

The multi zone AMP communication protocol is based on the following invariants:

- The OS driver and MultiZone communicate via a shared memory buffer. In the context of

the MultiZone for Linux SDK targeting the SiFive U540, the buffer is placed in E51's DTIM (0x0100_0000);

- The shared buffer is split into 2 parts: (1) MultiZone inbox (OS driver sending messages to MultiZone zones) and (2) `Zone_0` (or OS driver) inbox (MultiZone zones sending messages to the OS driver);
- Each inbox consists of '`n`' inbox entries, where '`n`' is the number of zones running on MultiZone;
- Access to inboxes requires synchronization to avoid race conditions for multi-thread and multi-core systems. Reading `VALID_n` involves an acquire, while writing `VALID_n` involves a release.

2.2 Inboxes Layout

The OS driver pseudo zone is `Zone_0`. Communication to and from the MultiZone system is mediated through `Zone_0` using the multi zone free API: `ECALL_SEND(0,...)` and `ECALL_RECV(0,...)`.

2.2.1 MultiZone inbox

The MultiZone inbox has the layout specified in Table 2.1. Each `VALID_n`, `MSG_n` corresponds to `Zone_n` mailbox entry for messages from `Zone_0`.

Table 2.1: MultiZone inbox layout

Entry 1 (messages for `Zone_1`)	`VALID_1`	`MSB` (1 if message valid, 0 otherwise) *	8 Bytes
	`MSG_1`	`msg[0]`	4 Bytes
		`msg[1]`	4 Bytes
		`msg[2]`	4 Bytes
		`msg[3]`	4 Bytes
...	...	...	...
Entry n (messages for `Zone_n`)	`VALID_n`	`MSB` (1 if message valid, 0 otherwise)	8 Bytes
	`MSG_n`	`msg[0]`	4 Bytes
		`msg[1]`	4 Bytes
		`msg[2]`	4 Bytes
		`msg[3]`	4 Bytes

(*) Taking into account the need of synchronization while accessing the shared buffer's inboxes, `VALID_n` is also used to implement a mutex. Acquiring a mutex implies a writing of the value '1' while releasing the mutex implies a writing of the value '0'. Listing 2.1 shows an example assembly code for a critical section guarded by a test-and-set spinlock. Note the first AMO is marked `aq` to order the lock acquisition before the critical section, and the second AMO is marked `rl` to order the critical section before the lock relinquishment.

Listing 2.1: Example of synchronization using an atomic swap to implement a mutex

```
# Example from "Volume I: RISC-V User-Level ISA V2.2"
# 7.3 Atomic Memory Operations
      li            t0, 1         # Initialize lock value
again:
      amoswap.d.aq  t0, t0, (a0)  # Attempt to acquire lock
      bnez          t0, again     # Retry if unsuccessful
      # ...
      # critical section
      # ...
      amoswap.d.rl  x0, x0, (a0)  # Release lock by storing 0
```

2.2.2 Zone 0 inbox

The `Zone_0` inbox has the layout specified in Table 2.2. Each `VALID_n`, `MSG_n` corresponds to `Zone_0` mailbox entry for messages from `Zone_n`.

Table 2.2: `Zone_0` inbox layout

Entry 1 (messages from `Zone_1`)	`VALID_1`	`MSB` (1 if message valid, 0 otherwise) *	8 Bytes
	`MSG_1`	`msg[0]`	4 Bytes
		`msg[1]`	4 Bytes
		`msg[2]`	4 Bytes
		`msg[3]`	4 Bytes
...	...	...	...
Entry n (messages from `Zone_n`)	`VALID_n`	`MSB` (1 if message valid, 0 otherwise)	8 Bytes
	`MSG_n`	`msg[0]`	4 Bytes
		`msg[1]`	4 Bytes
		`msg[2]`	4 Bytes
		`msg[3]`	4 Bytes

(*) Taking into account the need of synchronization while accessing the shared buffer's inboxes, `VALID_n` is also used to implement a mutex. Acquiring a mutex implies a writing of the value '1' while releasing the mutex implies a writing of the value '0'.

2.3 Communication Flow

This section describes the communication flow between the OS (e.g., Linux) and MultiZone. For the remaining of this section, the description is based on the current implementation of the MultiZone for Linux SDK targeting the SiFive U540; however, the same rational can/shall be applied for other OSes (e.g, FreeRTOS, Zephyr) which might run concurrently to Linux following an AMP architecture.

Linux driver sending to MultiZone Zone (e.g. to Zone 1, entry 1 of `Zone_0` inbox):

1. Linux driver tries to acquire the lock;
2. Linux driver proceeds if LSB == 0 (lock acquired successfully);
3. Linux driver tests `VALID_1` MSB and proceeds if MSB == 0 (inbox empty) - if MSB == 1, the lock shall be released;
4. Linux driver writes 4x4 bytes to `MSG_1`;
5. Linux driver sets `VALID_1` MSB == 1;
6. Linux driver releases the lock;
7. MultiZone tries to acquire the lock;
8. MultiZone proceeds if LSB == 0 (lock acquired successfully);
9. MultiZone reads `VALID_1` and proceeds if MSB == 1 (inbox full) - if MSB == 0, the lock shall be released;
10. MultiZone reads the `MSG_1`;
11. MultiZone sets `VALID_1` MSB == 0 to signal it has read the message;
12. MultiZone releases the lock;

MultiZone sending messages to the Linux driver (e.g. from Zone 1, entry 1 of the MultiZone inbox):

1. MultiZone tries to acquire the lock;
2. MultiZone proceeds if LSB == 0 (lock acquired successfully);
3. MultiZone tests `VALID_1` MSB and proceeds if MSB == 0 (inbox empty) - if MSB == 1, the lock shall be released;
4. MultiZone writes 4x4 bytes to `MSG_1`;
5. MultiZone sets `VALID_1` MSB == 1;
6. MultiZone releases the lock;
7. Linux driver tries to acquire the lock;
8. Linux driver proceeds if LSB == 0 (lock acquired successfully);
9. Linux driver reads `VALID_1` and proceeds if MSB == 1 (inbox full) - if MSB == 0, the lock shall be released;
10. Linux driver reads `MSG_1`;
11. Linux driver sets `VALID_1` MSB == 0 to signal it has read the message;
12. Linux driver releases the lock;