Virtio Message Bus over FF-A

Document number	DEN0153
Document quality	ALP0
Document version	1.0
Document confidentiality	Non-confidential

Virtio Message Bus over FF-A

Release information

Date	Version	Changes
2025/Oct/22	1.0 ALP0	• Initial public version based on Virtio Over Messages RFCv2.

Arm Non-Confidential Document Licence (“Licence”)

This Licence is a legal agreement between you and Arm Limited (“**Arm**”) for the use of Arm’s intellectual property (including, without limitation, any copyright) embodied in the document accompanying this Licence (“**Document**”). Arm licenses its intellectual property in the Document to you on condition that you agree to the terms of this Licence. By using or copying the Document you indicate that you agree to be bound by the terms of this Licence.

“**Subsidiary**” means any company the majority of whose voting shares is now or hereafter owner or controlled, directly or indirectly, by you. A company shall be a Subsidiary only for the period during which such control exists.

This Document is **NON-CONFIDENTIAL** and any use by you and your Subsidiaries (“Licensee”) is subject to the terms of this Licence between you and Arm.

Subject to the terms and conditions of this Licence, Arm hereby grants to Licensee under the intellectual property in the Document owned or controlled by Arm, a non-exclusive, non-transferable, non-sub-licensable, royalty-free, worldwide licence to:

(i) use and copy the Document for the purpose of designing and having designed products that comply with the Document;

(ii) manufacture and have manufactured products which have been created under the licence granted in (i) above; and

(iii) sell, supply and distribute products which have been created under the licence granted in (i) above.

Licensee hereby agrees that the licences granted above shall not extend to any portion or function of a product that is not itself compliant with part of the Document.

Except as expressly licensed above, Licensee acquires no right, title or interest in any Arm technology or any intellectual property embodied therein.

THE DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. Arm may make changes to the Document at any time and without notice. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

NOTWITHSTANDING ANYTHING TO THE CONTRARY CONTAINED IN THIS LICENCE, TO THE FULLEST EXTENT PERMITTED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, IN CONTRACT, TORT OR OTHERWISE, IN CONNECTION WITH THE SUBJECT MATTER OF THIS LICENCE (INCLUDING WITHOUT LIMITATION) (I) LICENSEE’S USE OF THE DOCUMENT; AND (II) THE IMPLEMENTATION OF THE DOCUMENT IN ANY PRODUCT CREATED BY LICENSEE UNDER THIS LICENCE). THE EXISTENCE OF MORE THAN ONE CLAIM OR SUIT WILL NOT ENLARGE OR EXTEND THE LIMIT. LICENSEE RELEASES ARM FROM ALL OBLIGATIONS, LIABILITY, CLAIMS OR DEMANDS IN EXCESS OF THIS LIMITATION.

This Licence shall remain in force until terminated by Licensee or by Arm. Without prejudice to any of its other rights, if Licensee is in breach of any of the terms and conditions of this Licence then Arm may terminate this Licence immediately upon giving written notice to Licensee. Licensee may terminate this Licence at any time. Upon termination of this Licence by Licensee or by Arm, Licensee shall stop using the Document and destroy all copies of the Document in its possession. Upon termination of this Licence, all terms shall survive except for the licence grants.

Any breach of this Licence by a Subsidiary shall entitle Arm to terminate this Licence as if you were the party in breach. Any termination of this Licence shall be effective in respect of all Subsidiaries. Any rights granted to any Subsidiary hereunder shall automatically terminate upon such Subsidiary ceasing to be a Subsidiary.

The Document consists solely of commercial items. Licensee shall be responsible for ensuring that any use, duplication or disclosure of the Document complies fully with any relevant export laws and regulations to assure that the Document or any portion thereof is not exported, directly or indirectly, in violation of such export laws.

This Licence may be translated into other languages for convenience, and Licensee agrees that if there is any conflict between the English version of this Licence and any translation, the terms of the English version of this Licence shall prevail.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-21585 version 4.0

Contents

Virtio Message Bus over FF-A

Preface

This specification defines a transport binding of the *Virtio Over Message Transport* (hereafter, *virtio-msg transport*), as described in the Virtio specification [1] maintained by the OASIS consortium, to the *Arm Firmware Framework for Arm A-profile (FF-A)* version 1.2 [2].

It enables Virtio drivers and devices to exchange message-based requests and responses between FF-A endpoints using FF-A message-passing and memory-sharing primitives.

About This Document

This document specifies how operations allocated to the *Virtio Over Message Bus* (hereafter, *virtio-msg bus*) such as device discovery, message delivery, and memory sharing are mapped onto FF-A message-passing and memory-sharing interfaces to specify a *Virtio Message Bus over FF-A* (hereafter, *virtio-msg FF-A bus*).

Using This Document

This document consists of informative and normative sections, as follows:

- *Informative:* Concepts, Message Transfer, and the Glossary provide background and architectural context.
- *Normative:* Discovery, Memory Sharing, Monitoring, and Message Definitions specify the protocol details of the Virtio Message Bus over FF-A.
- *Error Handling:* Defines fallback responses and endpoint recovery behavior.
- *Compliance:* Defines mandatory implementation requirements.

Scope

This document is in an *alpha* state and may change substantially before a *beta* quality release. Implementations of this version are for evaluation and integration prototyping only and may not remain compatible with later revisions.

The design is derived from the publicly shared RFCv2 draft of the virtio-msg transport and adopts several elements expected in RFCv3. This forward alignment reduces later churn. Specifically:

- Field naming uses `msg_op` (Message Operation). Earlier draft material used `msg_id`.
- A single `msg_uid` (Message Unique Identifier) together with `dev_num` defines the correlation tuple (`dev_num`, `msg_uid`). This supports explicit request/response correlation for any bus implementation adopting the format.
- The common header is 8 bytes (type, msg_op, dev_num, msg_uid, msg_size), an anticipated RFCv3 format superseding the earlier 6-byte form without `msg_uid`.
- Error reporting: The bus may report certain bus-level failures through a dedicated bus error message instead of only handling them internally. This anticipates expected relaxation in later virtio-msg drafts.

If future RFCv3 text diverges substantially from an assumption listed above, the terminology or constraints in this document may be adjusted in a subsequent revision.

Additional reading

This section lists publications by Arm and by third parties.

See Arm Developer (http://developer.arm.com) for access to Arm documentation.

[1] *Virtual I/O Device (VIRTIO) Version 1.4*. See https://docs.oasis-open.org/virtio/virtio/v1.3/virtio-v1.3.html

[2] *Arm® Firmware Framework for Arm A-Profile Architecture version 1.2*. See https://developer.arm.com/documentation/den0077/g

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

If you have comments on the content of this book, send an e-mail to errata@arm.com. Give:

- The title (Virtio Message Bus over FF-A).
- The number (DEN0153 1.0).
- The page numbers to which your comments apply.
- The rule identifiers to which your comments apply, if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests PDFs only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the appearance or behavior of any document when viewed with any other PDF reader.

Chapter 1
Concepts

The *Virtio Message Bus over FF-A* (virtio-msg FF-A bus) is a binding of the generic virtio-msg transport onto FF-A primitives to connect isolated endpoints.

The virtio-msg transport (as defined by the Virtio specification) encapsulates Virtio device operations into transport messages and is shared across all virtio-msg bus implementations. This FF-A binding adds only what is needed to realize that transport over FF-A: endpoint discovery, version negotiation, message delivery method selection, event delivery method selection, and shared memory management for virtqueue and buffer access.

This chapter summarizes foundational concepts. Detailed normative rules are defined in later chapters.

1.1 Software Stack Components

The virtio-msg FF-A bus software stack consists of the following components:

- **Virtio driver/device**: Implements Virtio device class logic and driver-side operations.
- **Virtio-msg transport driver/device**: Encodes and decodes Virtio driver operations into structured messages.
- **virtio-msg FF-A bus driver/device**: FF-A-specific virtio-msg bus implementation integrating FF-A messaging and memory-sharing primitives.
- **FF-A driver**: Provides FF-A messaging and memory-sharing services to higher components.

The virtio-msg FF-A bus is instantiated on both the driver and device endpoints.

The following diagram illustrates a representative topology:

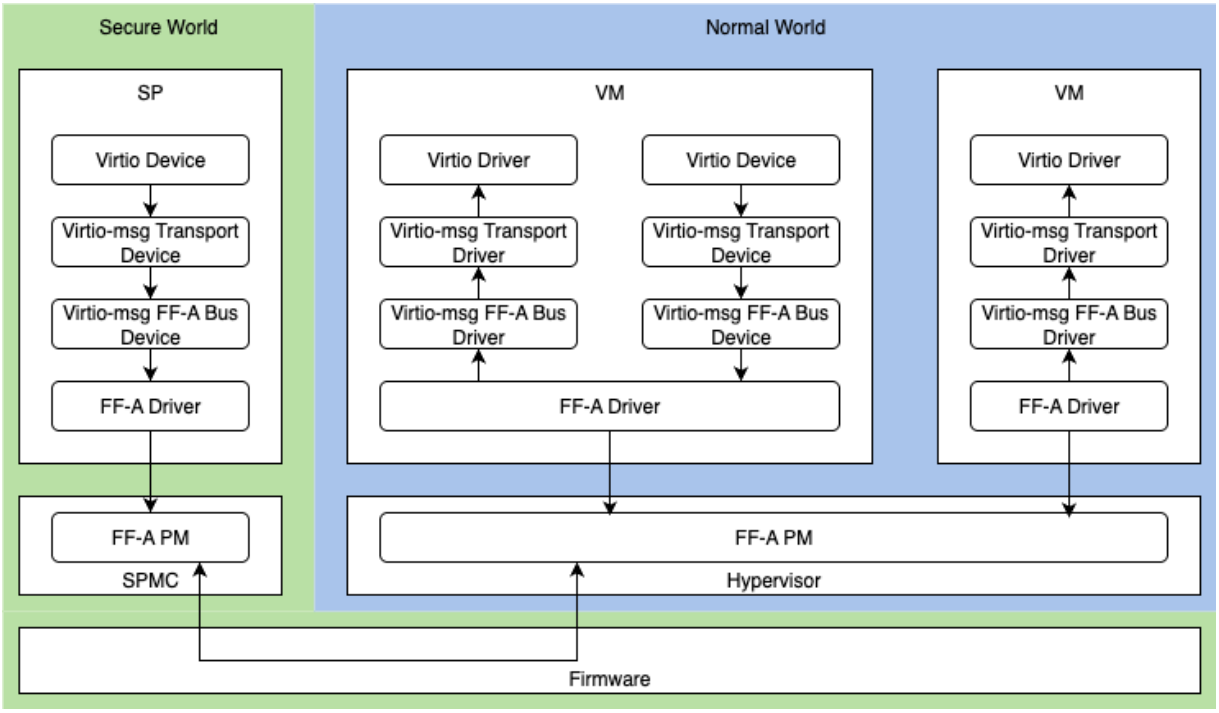

Figure 1.1: System Topology

The diagram is illustrative rather than prescriptive. It shows that endpoints may act as both providers and consumers of Virtio devices across security and virtualization boundaries.

1.2 Scope and Relationship to the Virtio Specification

The virtio specification defines transport semantics from the driver perspective and does not prescribe the software composition of the device endpoint.

This document specifies the virtio-msg FF-A bus at both endpoints to bind virtio transport to FF-A and does not redefine Virtio device semantics, feature negotiation, virtqueue layout, or transport message formats. Those aspects remain fully governed by the Virtio specification.

All transport (device / driver) messages defined by the virtio-msg transport are forwarded unchanged by the virtio-msg FF-A bus.

The virtio-msg FF-A bus performs FF-A endpoint discovery, version/method negotiation, routing and memory sharing.

Only FF-A specific bus messages are defined here.

1.3 Message-Based Communication

The virtio-msg transport replaces register access with a message-based model that encapsulates Virtio operations.

This binding distinguishes between:

- **Transport messages**: Defined by the Virtio specification. Forwarded by virtio-msg FF-A bus without modification or reinterpretation.
- **Bus messages**: Core bus messages are defined by the Virtio specification as part of the virtio-msg transport. This specification defines only the additional FF-A binding specific bus messages needed for version negotiation, memory sharing orchestration, event delivery configuration, lifecycle management and error signaling.

Virtio specification defines two messages categories:

- **Normal Message**: Expects a response.
- **Event Message**: Does not expect a response.

1.4 FF-A Primitives

The FF-A (Firmware Framework for A-profile) provides the foundational mechanisms used by the virtio-msg FF-A bus. These include:

- **Message Transfer Primitives**: Enable structured communication between isolated endpoints using direct and indirect messaging.
- **Memory Sharing Primitives**: Facilitate the sharing and revocation of memory regions, enabling secure and efficient data exchange.
- **Notification Mechanisms**: Allow endpoints to signal events or changes in state asynchronously.
- **Partition Discovery Primitives**: Support the identification and enumeration of FF-A partitions, enabling endpoint discovery.

The virtio-msg FF-A bus leverages these primitives across multiple aspects of its operation, including endpoint discovery (see Chapter 2 *Discovery*), message transfer (see Chapter 3 *Message Transfer*), and memory sharing (see Chapter 4 *Memory Sharing*).

The FF-A driver abstracts transport-specific details, presenting a simplified messaging model for higher-level services. Each message received over FF-A is associated with a service UUID that identifies the intended recipient within the endpoint. This abstraction allows higher-level components to interact with FF-A primitives without directly handling protocol details.

1.5 Message Transfer and Event Signaling

The virtio-msg FF-A bus provides three underlying transfer methods. All normal Messages and event Messages use one of these:

- **Direct messaging**: Uses FF-A direct request/response calls using `FFA_MSG_SEND_DIRECT_REQ2` / `FFA_MSG_SEND_DIRECT_RESP2` FF-A ABI; delivery is synchronous.
- **Indirect messaging**: Uses RX/TX buffers shared with Partition Manager using `FFA_MSG_SEND2` FF-A ABI; delivery is asynchronous.
- **FIFO-based Message Transfer**: Uses a shared memory FIFO pair with notifications for signaling; delivery is asynchronous.

Event messages are either delivered asynchronously or can be retrieved by the driver endpoint using a polling system.

Selection of a transfer method and any event signaling approach is performed during discovery and event delivery configuration (see 3.7 *Transfer Method Selection*) based on negotiated FF-A Bus Features and whether the device endpoint can originate direct, indirect or FIFO-based transfers.

1.6 Memory Sharing Model

The memory-sharing model enables DMA-based communication between a Virtio Driver and Virtio Device using virtqueues. This binding adds only the FF-A mechanics required to share, revoke, and reference memory. Virtqueue layout, descriptor formats, and Virtio feature negotiation remain unchanged and are not restated.

Key binding-specific aspects:

- **Shared Memory Areas**: Regions allocated by the driver endpoint and shared with the device endpoint using FF-A memory management primitives.
- **Area identifiers**: Unique 16-bit IDs for tracking shared regions.
- **Bus addresses**: 64-bit values combining an Area Identifier and offset to reference subranges inside a shared area.
- **Memory Revocation**: Mechanisms to revoke and reclaim shared memory when no longer needed.

These mechanisms ensure secure and efficient access to shared resources (see Chapter 4 *Memory Sharing*).

1.7 Endpoint Roles

Endpoints in the Virtio Message Bus over FF-A can assume one or both of the following roles:

- **Driver Endpoint**: Initiates discovery, device binding, and message exchange.
- **Device Endpoint**: Hosts one or more Virtio devices and responds to requests from driver endpoints.

Role assignment is flexible, allowing an endpoint to implement both roles concurrently, for example, to expose devices to other endpoints while also using remote devices.

1.8 Endpoint Lifecycle

The lifecycle of an endpoint includes:

1. **Initialization**: The endpoint initializes its local software stack and advertises its role.
2. **Version Negotiation and Discovery**: Driver endpoints identify compatible device endpoints and negotiate supported protocol versions and capabilities. Once compatibility is established, the driver endpoint discovers the set of exposed Virtio devices and configures how device-to-driver events are to be delivered (see Chapter 2 *Discovery*).
3. **Operation**: Endpoints exchange messages and perform memory-sharing operations to support Virtio devices (see Chapter 3 *Message Transfer* and Chapter 4 *Memory Sharing*).
4. **Dynamic Management**: Devices can be added or removed during runtime, enabling hot-plugging and seamless integration without requiring a system reboot. This relies on mechanisms such as endpoint discovery, message-based communication, and notification signaling (see Chapter 5 *Monitoring and Hotplug*).
5. **Monitoring and Reset**: Device endpoints can be monitored using a ping mechanism to ensure their availability. If a device endpoint becomes unresponsive, the driver endpoint can initiate a reset to restore functionality. These mechanisms are detailed in the hotplug section (see Chapter 5 *Monitoring and Hotplug*).
6. **Teardown**: Endpoints release resources and terminate communication when no longer needed.

1.9 Error Handling Framework

Error handling complements (not replaces) transport-defined semantics. Error classification categories (see Chapter 6 *Operational Error Handling*):

- **Transient** (e.g., `FFA_BUSY`): Retry with bounded policy.
- **Fatal**: Trigger device removal or reset.
- **Reported / Transport-visible**: Conveyed via a bus error message to the peer; then surfaced to the virtio-msg transport unchanged.

The binding never alters the meaning of an error code defined by the Virtio specification; it only defines when and how FF-A level failures are converted into bus-visible error messages or are forwarded to the transport.

Chapter 2
Discovery

Discovery is the initial coordination phase in the virtio-msg FF-A bus. It establishes endpoint compatibility, communication capabilities, and identifies available devices before operational message exchanges.

Discovery is organized into the following phases:

1. **Endpoint discovery**: The virtio-msg FF-A bus driver enumerates FF-A partitions and selects those advertising the device protocol UUID.
2. **Version negotiation**: The virtio-msg FF-A bus driver and each device endpoint agree on the FF-A Bus Version, the Transport Revision, the Feature Bits, and the FF-A Bus Features.
3. **Optional FIFO setup**: The virtio-msg FF-A bus driver configures FIFO-based transfer when both endpoints support it.
4. **Device enumeration**: The virtio-msg FF-A bus driver lists and characterises Virtio devices available on each device endpoint.
5. **Event delivery configuration**: The virtio-msg FF-A bus driver selects the device-to-driver event delivery method.

This chapter details the discovery and enumeration process.

2.1 Endpoint Discovery

Each FF-A partition that hosts a virtio-msg FF-A bus driver and/or a virtio-msg FF-A bus device advertises a protocol UUID in its FF-A partition information. Endpoint discovery enumerates FF-A partitions and selects those that advertise the virtio-msg FF-A bus device protocol UUID.

Two protocol UUIDs are defined—one identifying a partition implementing the virtio-msg FF-A bus driver role and one identifying a partition implementing the virtio-msg FF-A bus device role:

Table 2.1: Protocol UUIDs for virtio-msg FF-A bus endpoints

Endpoint Role	UUID
virtio-msg FF-A bus driver	bd7fd089-6795-472b-b47f-db0c5d9a719d
virtio-msg FF-A bus device	c66028b5-2498-4aa1-9de7-77da6122abf0

Endpoint discovery is performed cooperatively by the virtio-msg FF-A bus driver and the FF-A driver:

1. The virtio-msg FF-A bus driver requests a partition list filtered by the device protocol UUID from the FF-A driver.
2. The FF-A driver issues `FFA_PARTITION_INFO_GET` (Protocol UUID) and returns matching FF-A endpoints.
3. For each endpoint, the virtio-msg FF-A bus driver derives the initially usable message transfer methods from reported partition attributes.

Partitions advertising the virtio-msg FF-A bus device protocol UUID and supporting at least one compatible transfer method are candidates for protocol version negotiation.

2.2 Version Negotiation

After endpoint discovery the virtio-msg FF-A bus driver sends `FFA_BUS_MSG_VERSION` (see 7.3 *FFA_BUS_MSG_VERSION*) using any mutually supported direct or indirect messaging method. Negotiation establishes a common FF-A Bus Version (Major.Minor) and a common Transport Revision before any other exchange occurs.

The version negotiation uses two fields:

- `FF-A Bus Version`:
  - Major and minor components.
  - Major denotes an incompatibility boundary.
  - Minor values within a major are backward compatible unless stated otherwise.
- `Transport Revision`:
  - Integer identifying the revision defined by the virtio over messages transport specification.

2.2.1 Fast Path

The fast path follows these steps:

1. The virtio-msg FF-A bus driver sends `FFA_BUS_MSG_VERSION(0, 0)` to request the device endpoint highest supported FF-A Bus Version and Transport Revision.
2. The device endpoint returns its highest supported FF-A Bus Version, Transport Revision together with the Feature Bits, and FF-A Bus Features for it.
3. If the virtio-msg FF-A bus driver accepts those values it must send `FFA_BUS_MSG_VERSION(version, ↪revision)` echoing exactly the returned FF-A Bus Version and Transport Revision.
4. The device endpoint responds by echoing the same `(version, revision)` with the Feature Bits and FF-A Bus Features for it. Because the echoed version and revision match the driver's request, negotiation is complete for this association.

If the driver does not accept the values it enters the fallback procedure (see 2.2.2 *Fallback and Downgrade Procedure*) instead of sending the echo.

On successful completion the virtio-msg FF-A bus driver:

- Forwards the Transport Revision, Feature Bits, and the fixed maximum message size (104 bytes, see 3.2 *Message Size Constraints*) to the virtio-msg transport.
- Retains the FF-A Bus Version and FF-A Bus Features for internal selection of transfer and event delivery methods.

If any returned element violates local policy the driver proceeds with the ordered downgrade search.

At completion further per-message semantics are defined in 2.2.3 *Version Message Rules*.

2.2.2 Fallback and Downgrade Procedure

If the virtio-msg FF-A bus driver rejects the device response (for policy or capability reasons) it performs an ordered downgrade search. The ordering is:

1. Lower Transport Revision within the same FF-A Bus Version (same major and minor) if multiple revisions are supported.
2. If no acceptable Transport Revision exists, lower the minor version (same major) selecting the highest minor below the previous attempt.
3. If no acceptable minor remains, lower the major version selecting the highest lower major supported below the previous attempt.

Fallback procedure follows these steps:

1. Virtio-msg FF-A bus driver sends a proposal `FFA_BUS_MSG_VERSION(version, revision)` using the ordering rules above.
2. Device endpoint:

- If the requested FF-A bus Version and Transport Revision are supported, echoes the values back with its highest Feature Bits and FF-A Bus Features for that tuple and mark negotiation as complete for this association.
- If the requested FF-A bus Version and Transporr Revision are not supported, responds with `(0, 0)` to reject the request.

2. Virtio-msg FF-A bus driver:

- On rejection, it selects the next lower candidate per the ordering and repeats, or aborts negotiation for that device endpoint if exhausted.
- On acceptance, negotiation is complete as in the fast path.

If a device endpoint responds to the initial `(0, 0)` request with a pair that is lower than what is expected, the virtio-msg FF-A bus driver may elect to perform a bus reset to discard a previously negotiated version (see 5.3 *Bus Stop and Reset*).

After reset the driver repeats discovery for that endpoint and performs negotiation again.

2.2.3 Version Message Rules

The device endpoint applies the rules in Table 2.2 to every `FFA_BUS_MSG_VERSION` received from a given driver endpoint until a bus reset occurs (see 5.3 *Bus Stop and Reset*).

In this table, **features** denotes the *Feature Bits* and *FF-A Bus Features* bitmasks returned with the message.

Table 2.2: Device behavior for `FFA_BUS_MSG_VERSION`

Input `(version, revision)`	Negotiation complete?	Device endpoint response	Negotiated state change
`(0, 0)`	No	Highest supported pair + features	None
`(0, 0)`	Yes	Negotiated pair + features	None
Supported non-zero `(V, R)`	No	Echo `(V, R)` + features	Set to `(V, R)`
Negotiated pair `(V, R)`	Yes	Echo `(V, R)` + features	None
Any other pair	Any	`(0, 0)`	None

`FFA_BUS_MSG_VERSION` may be accepted in any internal state, including immediately after a local reboot.

Resetless resynchronization is achieved through a status query or an idempotent echo.

The driver endpoint must not attempt to negotiate a different `(version, revision)` without a bus reset (see 5.3 *Bus Stop and Reset*).

2.2.4 Per-Endpoint Negotiation State

Each device endpoint is negotiated independently. Different device endpoints in the same system may operate with different FF-A Bus Versions, Transport Revisions, selected transfer methods, and event delivery methods. The virtio-msg FF-A bus driver maintains a per-endpoint record including:

- Agreed FF-A Bus Version (major.minor)
- Transport Revision
- Feature Bits (forwarded to transport)
- FF-A Bus Features
- Selected transfer method(s)
- Selected event delivery method

Operations directed to a device endpoint must honor that endpoint's negotiated values. Uniformity across endpoints is neither required nor implied.

2.2.5 Feature Compatibility

Feature handling is as follows:

- **Feature Bits** are forwarded unchanged to the virtio-msg transport.
- **FF-A Bus Features** constrain which transfer methods and event delivery methods the bus may select for this endpoint pair.

Details on delivery selection and configuration are described in 3.7 *Transfer Method Selection* and 2.5 *Event Delivery Configuration*.

2.2.6 Pre-Negotiation Restrictions

Until negotiation completes the device processes only `FFA_BUS_MSG_VERSION`. Other messages may be silently dropped or answered with a no-operation response (`msg_op = 0`, no payload). See per-message rules in 2.2.3 *Version Message Rules*.

2.2.7 Supported Protocol Versions

The following table defines the currently supported negotiated values including FF-A Bus Version, Transport Revision, Feature Bits, and maximum message size.

Table 2.3: Supported Protocol Versions

FF-A Bus Version	Transport Revision	Feature Bits	Maximum Message Size (Bytes)	Comments
1.0	1	None	104	Current

The FF-A Bus Version may change without a Transport Revision bump (and vice versa); a compatible association requires agreement on both values.

2.3 FIFO-based Message Transfer Configuration

Support for FIFO-based transfer is indicated by bit[6] in the FF-A Bus Feature Flags field of the `FFA_BUS_MSG_VERSION` message. If both endpoints advertise support for FIFO-based transfer, the virtio-msg FF-A bus driver may initiate FIFO configuration as described in 3.6.2 *Configuration and Setup*.

For a detailed explanation of FIFO-based transfer, its rationale, and operational flow, see 3.6 *FIFO-based Message Transfer* in the Message Transfer chapter.

2.4 Device Enumeration

Device enumeration proceeds as follows (per device endpoint):

1. The virtio-msg FF-A bus driver sends `BUS_MSG_GET_DEVICES`. The device endpoint returns a bitmap in which each set bit corresponds to an available Device Number and may supply a continuation offset if more Device Numbers remain.
2. For each set bit the virtio-msg FF-A bus driver sends `VIRTIO_MSG_GET_DEVICE_INFO(dev_num)`. If the response succeeds the Virtio device can be bound; on failure that Device Number is skipped.

Message encodings for `VIRTIO_MSG_GET_DEVICE_INFO` and `BUS_MSG_GET_DEVICES` are defined by the Virtio specification [1].

The returned Virtio device ID and Virtio vendor ID identify the device class and vendor implementation. An operating system or runtime uses these identifiers to locate and probe the appropriate Virtio driver for each device.

2.5 Event Delivery Configuration

After device enumeration the virtio-msg FF-A bus driver sends `FFA_BUS_MSG_EVENT_CONFIGURE` to select the device-to-driver event delivery method and enable event messages. The chosen method (poll, notification-assisted poll, indirect message, or FIFO-based) is derived using the rules in 3.7 *Transfer Method Selection* based on the intersection of:

- Device emission capabilities (FF-A Bus Features) and
- Driver reception capabilities.

The driver must apply a deterministic preference policy and pick the highest supported method not already rejected. The message carries required parameters (e.g., Notification ID; zero if plain polling).

Device endpoint on receipt must validate the method and:

- Return Success and begin using it for subsequent asynchronous event messages, or
- Return Error (method unsupported or rejected) and not emit events using it.

On Error the driver must try the next compatible method; if none remain it must discard the device endpoint (no event delivery possible). The driver must not send another configuration unless a bus reset occurs. A duplicate of the active configuration after success is benign (device may return Success if unchanged).

Event delivery is always configured by the driver endpoint; the device endpoint only advertises supported methods.

Structure and field semantics: see 7.4 *FFA_BUS_MSG_EVENT_CONFIGURE*.

2.6 Error Handling During Discovery

Error handling uses the following rules (each applied per device endpoint):

- **Negotiation**: If the proposed FF-A Bus Version or Transport Revision is unsupported the device returns `(0, 0)`.
- **Post-negotiation failures**: Timeouts or invalid responses allow the virtio-msg FF-A bus driver to discard or reset the device endpoint; protocol-visible faults may be reported using `FFA_BUS_MSG_ERROR`.
- **Enumeration**: Failures of `BUS_MSG_GET_DEVICES` or `VIRTIO_MSG_GET_DEVICE_INFO` cause the driver to skip the affected Device Number; persistent systemic failures justify endpoint discard.
- **Configuration**: A result of `Error` from `FFA_BUS_MSG_EVENT_CONFIGURE` or `FFA_BUS_MSG_FIFO_CONFIGURE` ↪ requires fallback selection or endpoint discard if no alternative exists.

The message `FFA_BUS_MSG_ERROR` must not be used to signal pure negotiation failure.

2.7 Discovery Summary

Illustrative end-to-end sequence (numbers correspond to the diagram in Figure 2.1):

Identify endpoints

1. The virtio-msg FF-A bus driver queries the FF-A driver for partitions advertising the virtio-msg FF-A bus device protocol UUID.
2. The FF-A driver uses `FFA_PARTITION_INFO_GET` with the UUID filter to retrieve the list of endpoints providing the protocol UUID
3. The partition manager returns a list of endpoints providing the protocol UUID.
4. The FF-A driver returns the matching list.

Negotiate version (Fast-Path)

5. The virtio-msg FF-A bus driver sends `FFA_BUS_MSG_VERSION(0, 0)` to request the highest supported FF-A Bus Version and Transport Revision.
6. The device endpoint replies with `FFA_BUS_MSG_VERSION(version, revision, features)` including the Feature Bits and FF-A Bus Features for its highest supported values or the active negotiated pair.
7. The virtio-msg FF-A bus driver sends `FFA_BUS_MSG_VERSION(version, revision)` with the received values to complete negotiation.
8. The device endpoint replies with `FFA_BUS_MSG_VERSION(version, revision, features)` with the same values as the request and marks version as negotiated.

Optional FIFO setup

Enumerate devices

9. The virtio-msg FF-A bus driver sends `BUS_MSG_GET_DEVICES` to begin enumeration.
10. The device endpoint returns `BUS_MSG_GET_DEVICES(bitmap, continuation_offset)` with bits set for available Device Numbers.
11. For each set bit the virtio-msg FF-A bus driver sends `VIRTIO_MSG_GET_DEVICE_INFO(dev_num)`.
12. The device endpoint replies with `VIRTIO_MSG_GET_DEVICE_INFO(dev_id, vendor_id, config)`.

Enable event delivery and bind drivers

13. The virtio-msg FF-A bus driver sends `FFA_BUS_MSG_EVENT_CONFIGURE(method, notification_id)`.
14. The device endpoint replies with `FFA_BUS_MSG_EVENT_CONFIGURE(Success)`.
15. The virtio-msg FF-A bus driver binds class drivers to all discovered supported devices.

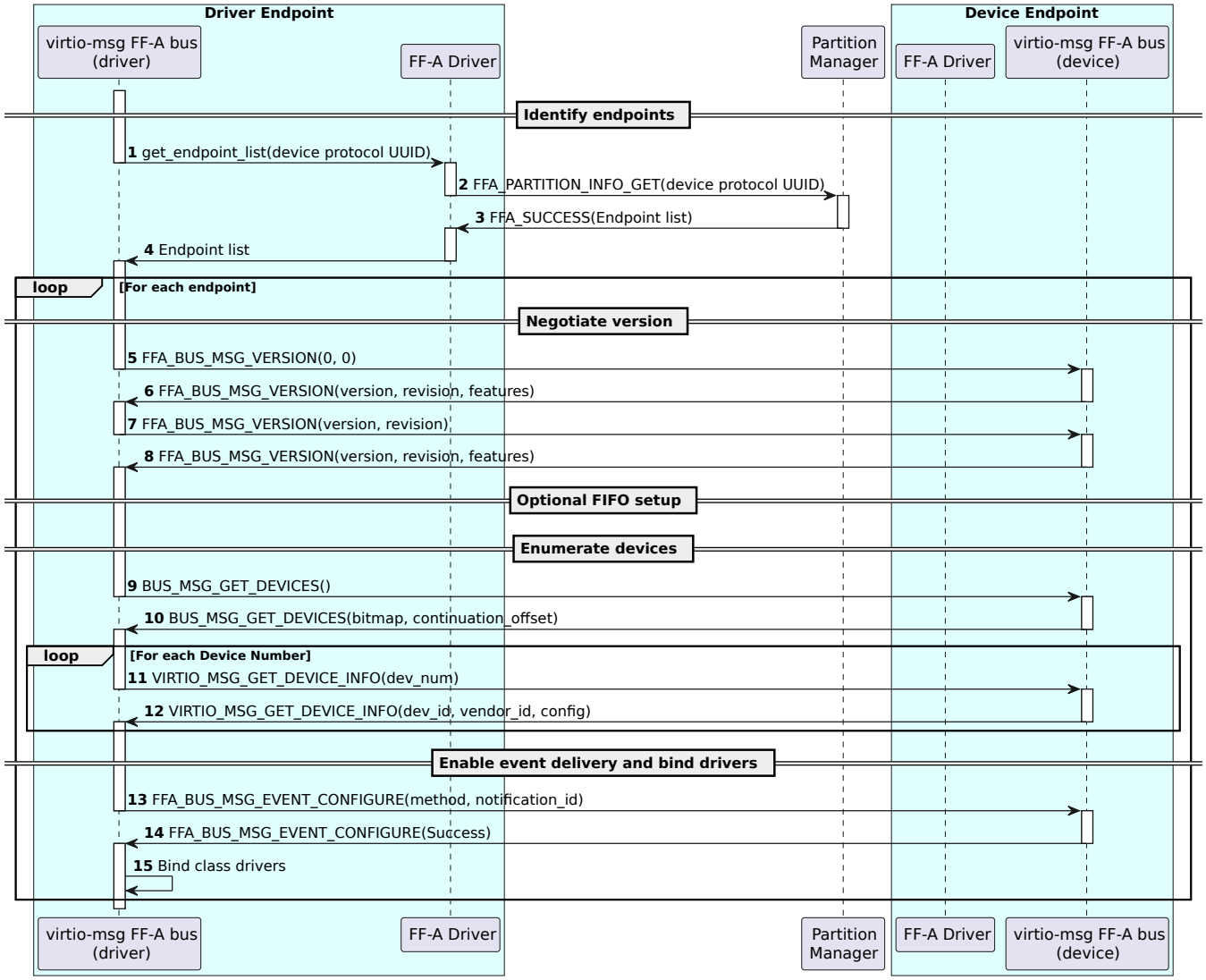

Figure 2.1: Discovery Summary Flow

Chapter 3
Message Transfer

This chapter defines how Virtio messages are exchanged between a virtio-msg FF-A bus driver and a device endpoint using FF-A message-passing mechanisms.

The virtio-msg FF-A bus supports three Message Transfer Methods:

- **Direct messaging**, using `FFA_MSG_SEND_DIRECT_REQ2` and `FFA_MSG_SEND_DIRECT_RESP2`.
- **Indirect messaging**, using `FFA_MSG_SEND2` and shared TX/RX buffers.
- **FIFO-based messaging**, using a shared memory region and FF-A notifications.

The Message Transfer Method is selected per endpoint pair based on the capabilities exchanged during discovery (see Chapter 2 *Discovery*).

Both synchronous Normal Messages (request-response) and asynchronous Event Messages are carried over these mechanisms, subject to direction, message type, and delivery constraints.

The virtio-msg FF-A bus driver must configure the event delivery method to be used by the device endpoint using `FFA_BUS_MSG_EVENT_CONFIGURE`, based on the delivery capabilities discovered during version negotiation.

See 3.7 *Transfer Method Selection* for selection rules and 2.5 *Event Delivery Configuration* for configuration semantics.

3.1 Message Transfer Architecture

The virtio-msg FF-A bus delivers transport messages by invoking FF-A message-passing primitives through an abstract interface provided by the FF-A driver.

The FF-A driver exposes access to the underlying ABI and is responsible for:

- Registering and managing shared TX/RX buffers for indirect messaging
- Issuing FF-A message calls (`FFA_MSG_SEND2`, `FFA_MSG_SEND_DIRECT_REQ2`, etc.)
- Handling FF-A return codes and delivering them to the Virtio-msg FF-A bus
- Forwarding received messages to the Virtio-msg FF-A bus for further processing

The virtio-msg FF-A bus interprets FF-A return codes and applies retry logic where required. Retry behavior for temporary failures, including handling of `FFA_BUSY`, is defined in 6.3 *Retry-Based Recovery*.

Messages are dispatched based on the protocol UUID contained in the FF-A message. The FF-A driver uses this UUID to route the message to the appropriate service instance, such as the virtio-msg FF-A bus or a class-specific handler.

The FF-A protocol UUIDs for the virtio-msg FF-A bus driver and device are defined in Table 2.1 and are used to identify the virtio-msg FF-A bus on each endpoint.

For comprehensive details on error handling—including retry logic, fallback procedures, and protocol-visible error reporting such as the use of `FFA_BUS_MSG_ERROR`, refer to Chapter 6 *Operational Error Handling*.

3.2 Message Size Constraints

All Virtio messages exchanged over FF-A must fit within a **104-byte** limit, including the 8-byte virtio-msg header (see *7.2 Message Header and Field Encoding*). This unified limit applies to direct, indirect, and FIFO-based transfers and maintains compatibility with FF-A direct messaging primitives.

The maximum payload size is 96 bytes (total minus header). The `msg_size` field must reflect the total number of bytes in the message, including the header and payload.

If the payload is smaller than the maximum, any unused bytes in the message buffer (for all transfer methods) must be zero-filled. This ensures consistent parsing and avoids leaking stale memory contents.

The 104-byte limit (8-byte header + 96-byte payload) allows all transfer methods (direct, indirect, FIFO-based) to share a single fixed buffer size while reserving headroom (8 bytes) for future extension. This limit is selected to fit entirely within the FF-A *Direct Request/Response 2* register payload, which can carry **up to 112 bytes**.

3.3 Correlation Semantics

Direct messaging is synchronous; the call boundary supplies correlation.

Asynchronous correlation applies only to driver requests that expect a response and when using indirect or FIFO-based messaging. The device endpoint never originates a correlated request: it sends responses or event messages only.

Fields used for correlation:

- `dev_num` identifies the device for transport messages. Bus messages also carry a `dev_num` field but its value, including 0, must not be used alone to classify the message type.
- `msg_uid` is 16 bits. Value 0 is reserved for messages that do not expect a response (events and any explicitly one-way operations).

Algorithm (receiver side correlation):

1. Classify the incoming message:
   - If `msg_op` identifies an event message operation, deliver/report the event immediately (per its semantics) and stop: no correlation lookup is performed (event `msg_uid` is 0 by definition).
   - If `msg_op` identifies `FFA_BUS_MSG_ERROR`, treat it as an error and obtain the original operation from `original_msg_op`.
   - Otherwise decode `msg_op` to determine whether the message is a bus message or a transport (device) message. Do not use `dev_num` alone for classification; value 0 has no special meaning.
2. Determine device context:
   - For a transport message, use `dev_num` to select the device's in-flight request table.
   - For a bus message, ignore `dev_num` for lookup purposes.
3. Correlate using `msg_uid`:
   - Bus message: lookup key = `msg_uid`.
   - Transport message: lookup key = (`dev_num`, `msg_uid`).
   - `msg_uid` must be non-zero for any correlated response. Value 0 always denotes an event (no response expected) or an explicitly one-way operation.
4. Validate correlation inputs. If validation fails the message is ignored. See correlation error definitions in 6.2.1 *Correlation Errors*.
5. Apply ordering (transport messages only):
   - Responses must arrive in the order requests were submitted per device.
   - An out-of-order response is treated as a device error; the implementation may escalate (e.g., reset) but the specific escalation behavior is outside this correlation algorithm. The response itself can be ignored after logging.
6. Complete the original request on successful correlation and deliver the payload (or error) to the waiting logic.
7. Release the correlation key so the driver may reuse the `msg_uid` (and `dev_num` for transport) in future requests. Wrap-around is handled by not reusing a key while it is in flight.
8. Termination conditions (see 6.2.1 *Correlation Errors*) release the key and surface a failure. Escalation triggers (repeated malformed or ordering violations) are defined in Chapter 6 *Operational Error Handling*.

Supporting rules (sender side):

- Only the virtio-msg FF-A bus driver allocates non-zero `msg_uid` values.
- A driver request that expects a response uses a `msg_uid` unique among that device's in-flight requests (transport) or among bus in-flight requests (bus domain).
- Multiple in-flight requests per device are allowed (bounded by the 16-bit space; practical limit < 65k).
- Event messages always use `msg_uid` = 0 and never produce responses.

- On bus or device reset, all in-flight requests are abandoned and lookup tables are cleared (counters may be reset).

Multiple requests may be in flight for the same Virtio Device as long as each uses a distinct `msg_uid`. Responses may be received out of order and are matched by correlation only.

3.4 Direct Message Transfer

3.4.1 Purpose and Use Case

Direct messaging provides synchronous communication between the driver endpoint and the device endpoint using the FF-A direct message ABI. This method is used when the virtio-msg FF-A bus driver can send direct messages and the device endpoint can receive them (determined during discovery).

Direct messages are always initiated by the virtio-msg FF-A bus driver. The device endpoint does not initiate direct messages.

3.4.2 Responsibilities and Interfaces

The Virtio-msg FF-A bus delegates direct message transmission to the FF-A driver. For each direct message transmission, the bus provides:

- Destination endpoint ID
- Protocol UUID (identifying recipient service)
- Message payload (including the 8-byte Virtio-msg header and payload data)

The FF-A driver is responsible for:

- Formatting the message using the provided destination ID, UUID, and payload
- Invoking `FFA_MSG_SEND_DIRECT_REQ2`
- Returning the response payload or appropriate FF-A error code to the virtio-msg FF-A bus

3.4.3 Message Handling and Responses

Each direct message sent via `FFA_MSG_SEND_DIRECT_REQ2` must be answered synchronously using `FFA_MSG_SEND_DIRECT_RESP2`.

For standard request-response messages, the response is a valid Virtio message and must be parsed and dispatched by the receiving endpoint accordingly.

For driver-initiated event messages sent via direct messaging, a synthetic response is generated to satisfy FF-A protocol requirements (see synthetic response description below).

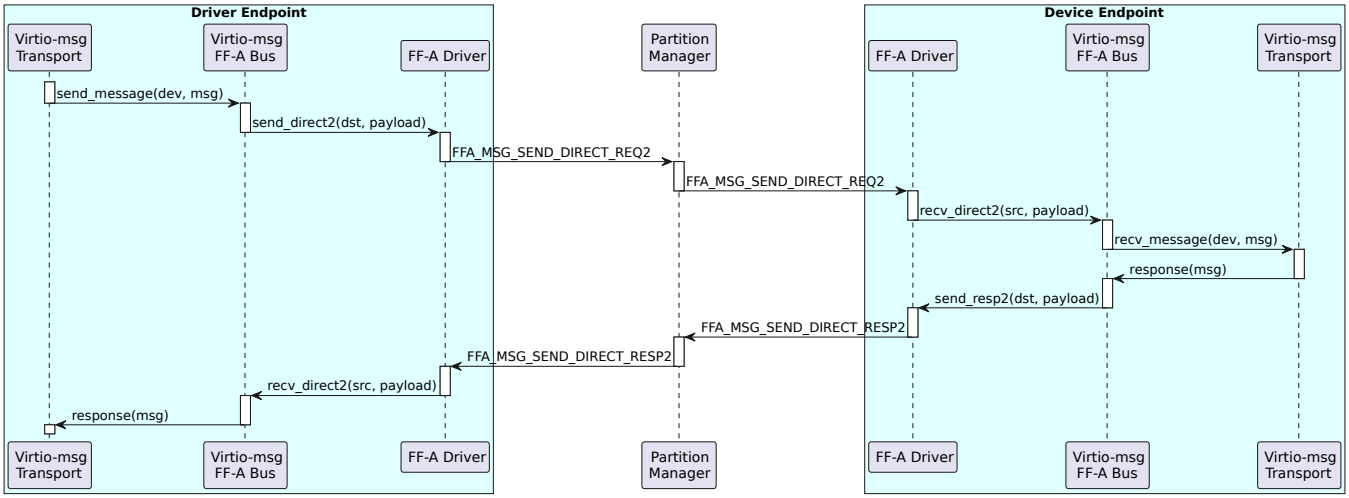

Figure 3.1: Direct Message Transfer Flow Diagram

3.4.4 Event Message Delivery

Indirect or FIFO-based messaging is preferred for delivering event messages, because they allow asynchronous transmission without requiring a synthetic response. When the sending endpoint supports an asynchronous method (FIFO- based or indirect) and the receiving endpoint supports reception, Event Messages should use that method instead of direct messaging.

The sections below describe how event delivery is handled when asynchronous messaging is not available for a given direction. A device endpoint must not expose event messages before completion of `FFA_BUS_MSG_EVENT_CONFIGURE` (see Chapter 2 *Discovery*).

3.4.4.1 Driver-initiated Event Messages (direct fallback)

Although Virtio event messages are conceptually one-way, FF-A direct messaging requires a response for each direct request.

To comply, the virtio-msg FF-A bus constructs a synthetic response with the following properties:

- `type` field is set to indicate a **response** and **bus message**
- `msg_op` and `dev_num` fields are copied from the original request
- `msg_uid` field is copied unchanged from the original request
- `msg_size` is set to 8 (i.e., header only; no payload)
- The response contains no additional payload bytes beyond the 8-byte header
- All unused bytes in the response message buffer are zero-filled

This synthetic response is handled locally by the virtio-msg FF-A bus and never propagated to the virtio-msg transport.

3.4.4.2 Device-initiated Event Messages

Device endpoints are not permitted to initiate direct messages.

The virtio-msg FF-A bus driver retrieves pending device-initiated event messages using `FFA_BUS_MSG_EVENT_POLL` ↪ over direct messaging only when the device endpoint cannot originate an asynchronous transfer (FIFO-based or indirect) in the device-to-driver direction.

When an asynchronous method (FIFO-based or indirect) is available and has been configured (see Chapter 2 *Discovery* and 3.7 *Transfer Method Selection*), the device endpoint delivers events directly over that method and direct polling is not used.

If no asynchronous device-to-driver method is available after `FFA_BUS_MSG_EVENT_CONFIGURE` completes, direct polling-based delivery uses the following signaling hierarchy:

1. Notification-assisted polling: If both endpoints support FF-A notifications and a Notification ID has been bound for event signaling, the device endpoint emits a notification when it enqueues one or more events. The virtio-msg FF-A bus driver then issues `FFA_BUS_MSG_EVENT_POLL` to retrieve queued events until the queue is empty.
2. Polling: If notifications are not supported or not bound, the driver endpoint periodically issues `FFA_BUS_MSG_EVENT_POLL` (e.g., from a scheduler tick or timer) to drain queued events.

Polling semantics:

- Each `FFA_BUS_MSG_EVENT_POLL` request elicits either:
  - one queued event message, or
  - an empty `FFA_BUS_MSG_EVENT_POLL` response indicating that no events remain.
- The device endpoint must return at most one event per poll and must return an empty `FFA_BUS_MSG_EVENT_POLL` ↪ response when the queue becomes empty. No explicit *more events* flag is defined or used by this binding.
- Upon receiving a non-empty event response, the driver endpoint must immediately issue another `FFA_BUS_MSG_EVENT_POLL` and repeat this drain loop until an empty response is received.
  - With notification-assisted polling, the driver starts the drain loop upon receipt of a notification and must stop polling when the first empty response is observed, resuming only after a subsequent notification.

- With periodic polling (no notifications), any poll that returns an event must be followed by immediate repeated polls until the first empty response is observed; the driver then returns to its periodic cadence.

Polling sequence:

1. The virtio-msg FF-A bus driver sends `FFA_BUS_MSG_EVENT_POLL` via direct messaging.
2. The device endpoint responds with either a queued event message (one per poll) or an empty `FFA_BUS_MSG_EVENT_POLL` response when no events remain.
3. If the response contained an event, the driver repeats step 1; otherwise the drain is complete.

Notification-assisted polling reduces unnecessary polls by allowing the virtio-msg FF-A bus driver to defer polling until signaled, but delivery still occurs through `FFA_BUS_MSG_EVENT_POLL` messages.

FF-A notification binding and usage for event signaling are defined in 2.5 *Event Delivery Configuration*.

3.5 Indirect Message Transfer

3.5.1 Purpose and Use Case

Indirect messaging provides asynchronous communication using shared TX/RX buffers managed by the FF-A driver. This method is used when both the driver endpoint and the device endpoint support sending and receiving indirect messages, as determined during discovery.

Both synchronous (request-response) and asynchronous (event) messages can be transmitted via indirect messaging.

3.5.2 Responsibilities and Interfaces

The Virtio-msg FF-A bus delegates indirect message transmission to the FF-A driver. For each message, the bus provides:

- Destination endpoint ID
- Protocol UUID (identifying recipient service)
- Message payload buffer (single Virtio message)

The FF-A driver is responsible for:

- Embedding the payload into the TX buffer as specified by the FF-A ABI
- Formatting and sending the message using `FFA_MSG_SEND2`
- Delivering incoming messages from the RX buffer to the virtio-msg FF-A bus

The FF-A driver does not implement retry logic. Handling of FF-A return codes is performed by the Virtio-msg FF-A bus.

3.5.3 Message Handling and Responses

Indirect messages sent via `FFA_MSG_SEND2` may return `FFA_BUSY` if the recipient's RX buffer is unavailable. The Virtio-msg FF-A bus implements retry mechanisms as described in 6.3 *Retry-Based Recovery*.

When a response is expected, the virtio-msg FF-A bus waits asynchronously for a message to arrive through the FF-A RX buffer. Responses received via indirect messaging are matched to the original request using (`dev_num`, `msg_uid`). The `msg_op` field may be used as an additional consistency check but must not be required for successful correlation. An error message whose `msg_op` differs from the request must still be correlated solely via (`dev_num`, `msg_uid`).

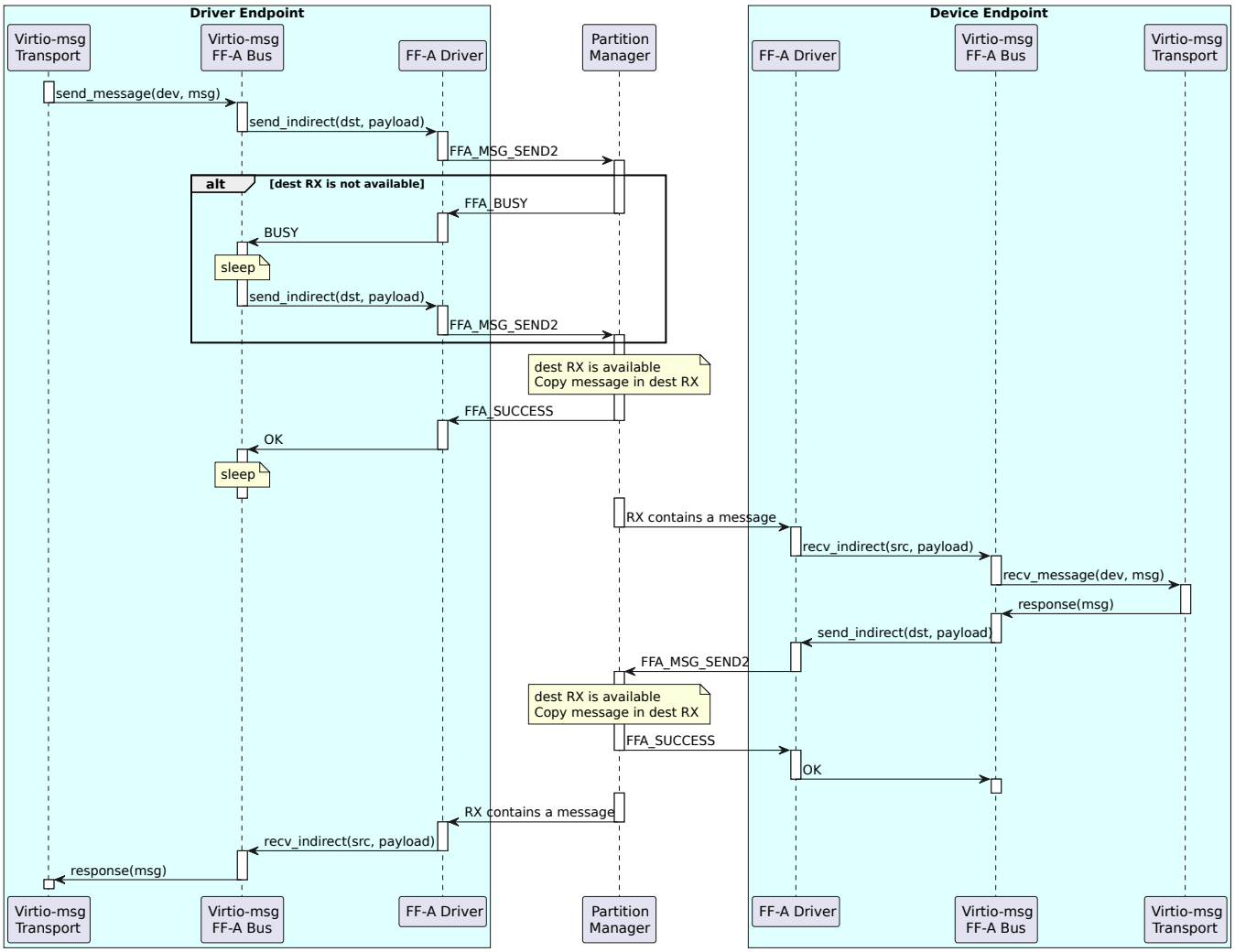

Figure 3.2: Indirect Message Transfer Flow Diagram

3.5.4 Event Message Delivery

Driver-initiated event messages can always be delivered asynchronously using indirect or FIFO-based messaging when supported.

Device-initiated event messages use the asynchronous method (FIFO-based or indirect) configured during event configuration (see Chapter 2 *Discovery*); direct polling paths are covered in 3.4 *Direct Message Transfer*.

3.6 FIFO-based Message Transfer

3.6.1 Purpose and Use Case

FIFO-based Message Transfer provides asynchronous communication between the virtio-msg FF-A bus driver and a device endpoint using a memory region shared by the driver plus FF-A notifications for signaling. The shared memory region contains two FIFO structures: one for each direction of communication.

This method supports both synchronous (request-response) and asynchronous (event) messages. It is used when both endpoints advertise support for FIFO-based transfer and complete configuration as described in 2.3 *FIFO-based Message Transfer Configuration*.

3.6.2 Configuration and Setup

FIFO-based Message Transfer requires explicit configuration before use.

Configuration is initiated by the virtio-msg FF-A bus driver sending `FFA_BUS_MSG_FIFO_CONFIGURE`. The format and encoding of this message are described in 7.10 *FFA_BUS_MSG_FIFO_CONFIGURE*.

If the device endpoint replies with an error, the virtio-msg FF-A bus driver must fall back to an alternate transfer method as described in 3.7 *Transfer Method Selection*.

FIFO configuration may be performed before or after device enumeration, but must be completed before sending `FFA_BUS_MSG_EVENT_CONFIGURE`.

3.6.2.1 FIFO setup

The virtio-msg FF-A bus driver allocates a memory region containing two FIFO structures:

- The first FIFO is used for driver-to-device messages.
- The second FIFO is used for device-to-driver messages.

The FIFO format and layout are defined in 9.1 *FIFO Message Format*. The virtio-msg FF-A bus driver must initialize both FIFO headers before sharing the memory.

The virtio-msg FF-A bus driver may choose the FIFO entry size and depth according to its own requirements. A recommended configuration is 30 FIFO entries of 128 bytes each, for a total memory size of 8 KiB. This layout ensures alignment with typical page sizes and provides sufficient message depth for common use cases.

The memory region must be shared with the device endpoint using `FFA_MEM_SHARE`.

The region must be described with the following attributes:

- **Memory Type**: Normal memory
- **Access Permissions**: Read-Write
- **Shareability**: Inner Shareable or Outer Shareable
- **Cacheability**: Write-Back cacheable
- **Security**: Non-secure memory

The FF-A memory handle returned by `FFA_MEM_SHARE` is communicated to the device endpoint via the `FFA_BUS_MSG_FIFO_CONFIGURE` request message.

3.6.2.2 Notification setup

Each endpoint must bind a Notification ID that the peer will use to signal FIFO activity.

The virtio-msg FF-A bus driver must bind a Notification ID that the device endpoint will use to notify when messages are written to the device-to-driver FIFO. This ID is included in the `FFA_BUS_MSG_FIFO_CONFIGURE` request message.

The device endpoint must bind a Notification ID that the virtio-msg FF-A bus driver will use to notify when messages are written to the driver-to-device FIFO. This ID is returned in the response to the `FFA_BUS_MSG_FIFO_CONFIGURE` message.

Both bindings must be performed using `FFA_NOTIFICATION_BIND` before the configure message is sent or acknowledged. FF-A does not permit delivery of a notification from a given sender unless the receiver has previously bound the sender ID.

3.6.2.3 Configuration Flow

The configuration sequence is as follows:

1. Driver prepares and shares memory using `FFA_MEM_SHARE`.
2. Driver binds its Notification ID using `FFA_NOTIFICATION_BIND`.
3. Driver sends `FFA_BUS_MSG_FIFO_CONFIGURE` including the memory handle and driver Notification ID.
4. Device endpoint binds its notification ID using FFA_NOTIFICATION_BIND.
5. Device endpoint retrieves the memory region using FFA_MEM_RETRIEVE_REQ.
6. Device endpoint replies with FFA_BUS_MSG_FIFO_CONFIGURE, including its notification ID.
7. Both endpoints may begin using FIFO-based transfer as defined in this section.

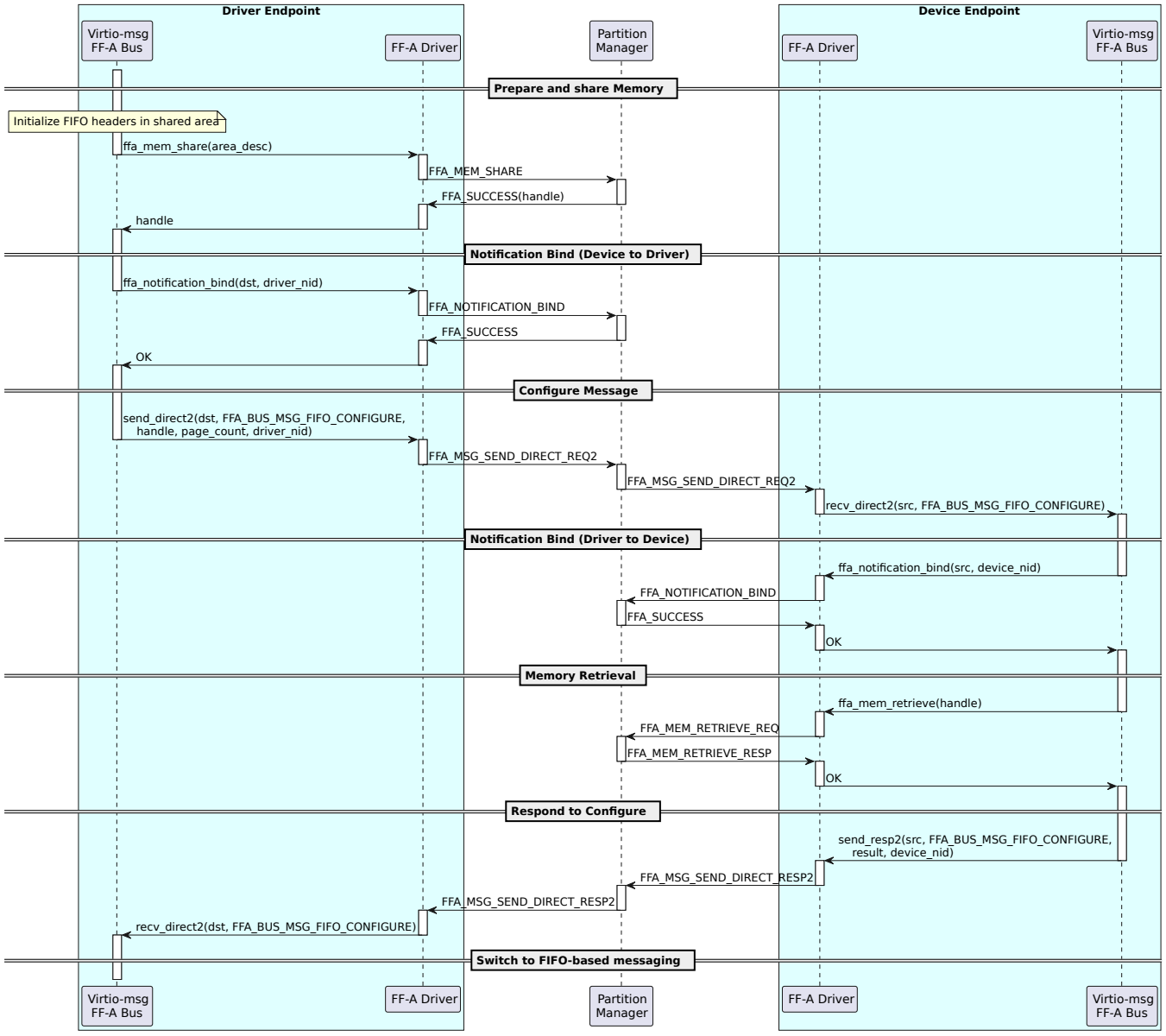

Figure 3.3: FIFO-based Message Transfer Configuration Flow

3.6.3 Responsibilities and Interfaces

The Virtio-msg FF-A bus is responsible for:

- Writing messages into the transmit FIFO
- Reading messages from the receive FIFO
- Sending notifications using `FFA_NOTIFICATION_SET` after writing
- Processing received notifications to drain the receive FIFO

The FF-A driver is responsible for:

- Emitting notifications to the peer
- Delivering incoming notifications to the virtio-msg FF-A bus
- Providing access to the shared memory region

Each endpoint owns one FIFO for transmission and one for reception. The driver is responsible for initializing both FIFOs and sharing the memory with the device endpoint.

The first FIFO is used for messages from the driver to the device endpoint.

The second FIFO is used for messages from the device endpoint to the virtio-msg FF-A bus driver.

Notification IDs must be bound by each endpoint during configuration before message delivery begins.

3.6.4 Message Handling and Responses

Each message written to the FIFO must follow the access protocol and memory ordering rules defined in 9.1 *FIFO Message Format*.

When a response is expected, the sending endpoint writes the request to its outbound FIFO and signals the peer using a notification.

The receiving endpoint reads the message, processes it, and writes the response to its own outbound FIFO. The `dev_num` and `msg_uid` fields must be copied unchanged into the response to allow correlation. The `msg_op` field should be copied unchanged for a successful response; an error response may use a different `msg_op` (e.g., `FFA_BUS_MSG_ERROR`).

Responses are delivered by reading the inbound FIFO and matched to the original request using (`dev_num`, `msg_uid`). Implementations may additionally check `msg_op` for advisory validation, but must not rely on it for correlation.

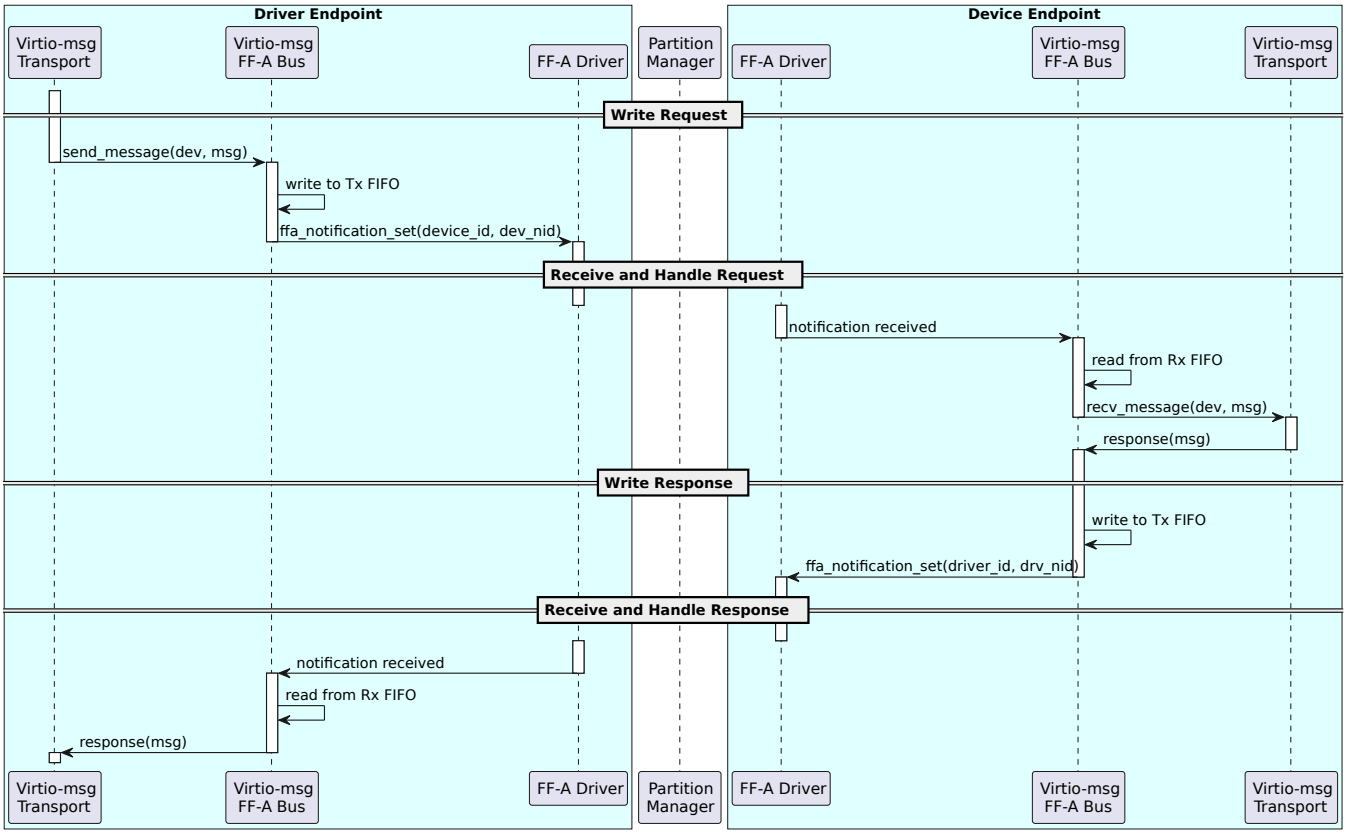

Figure 3.4: FIFO-based Message Transfer Flow Diagram

3.6.5 Event Message Delivery

Driver-initiated and device-initiated event messages are delivered using the same FIFO mechanism as request and response messages once FIFO-based transfer has been configured (see Chapter 2 *Discovery*). The sending endpoint writes the event message to its outbound FIFO and signals the peer using `FFA_NOTIFICATION_SET`.

3.7 Transfer Method Selection

Each endpoint selects the appropriate message delivery method based on the capabilities advertised by its peer during version negotiation.

The virtio-msg FF-A bus driver sends both control and event messages. Device endpoints respond to control messages and initiate event messages.

The delivery rules below are listed in order of preference. If a method is not supported in a given direction, the next available one must be used. If no valid method is available, the virtio-msg FF-A bus driver must skip the corresponding device endpoint.

- **Driver-to-Device Control Messages** (response expected):
  - FIFO-based messaging (if supported and configured).
  - Indirect messaging.
  - Direct messaging (fallback) when no asynchronous method is available.
- **Driver-to-Device Event Messages** (no real response expected):
  - FIFO-based messaging (if supported and configured).
  - Indirect messaging.
  - Direct messaging with a synthetic response (fallback only).
- **Device-to-Driver Event Messages**:
  - FIFO-based messaging (if supported and configured).
  - Indirect messaging.
  - Notification-assisted polling (direct polling triggered by notification).
  - Polling (direct polling without notification).

Event messages are unidirectional and do not require a response. Notifications are only used to signal pending device-to-driver events and are only valid from device endpoint to virtio-msg FF-A bus driver.

Support for a delivery method alone does not permit device-to-driver Event Message delivery. The virtio-msg FF-A bus driver must configure the chosen method using `FFA_BUS_MSG_EVENT_CONFIGURE` before the device endpoint may use it.

If no supported delivery method is available in a required direction, the virtio-msg FF-A bus driver must skip the corresponding device endpoint and exclude it from the active topology.

Chapter 4
Memory Sharing

This chapter defines the memory sharing model used by the virtio-msg FF-A bus to grant the Virtio Device access to virtqueues and data buffers owned by the Virtio Driver.

This model does not apply to the shared memory used for FIFO-based Message Transfer, which is shared separately during FIFO-based messaging configuration (see 2.3 *FIFO-based Message Transfer Configuration*).

Memory sharing is requested by the virtio-msg transport or directly by the Virtio Driver, depending on the usage context. It applies to the virtqueue structures and data buffers commonly exchanged between a Virtio Driver and a Virtio Device.

Shared memory areas are shared using the FF-A memory management interface, which enables transfer of access without transfer of ownership. The driver endpoint retains ownership of all shared memory areas. Access is revoked when no longer required, using unshare and reclaim transactions. This model enables the Virtio Device to perform direct memory access (DMA) operations on driver endpoint-owned memory in accordance with FF-A memory access semantics.

4.1 Shared Memory Areas and Identifiers

Each shared memory area between a driver endpoint and device endpoint is assigned a 16-bit Area Identifier by the virtio-msg FF-A bus driver. The Area ID is scoped to a specific driver/device endpoint pair and is valid only for the lifetime of the mapping.

The area identifier is used in all memory-related bus messages and forms part of the Bus Address format used by the virtio-msg FF-A bus device to locate shared buffers (see 4.2 *Bus Address Format*).

Each endpoint must internally track, for every active area identifier:

- The FF-A memory handle associated with the shared memory area
- The virtual and physical address mapping for each page in the area

This information is required to:

- Reclaim shared memory areas via `FFA_MEM_RECLAIM`
- Translate a Bus Address into a local memory pointer for device endpoint access

Area identifiers must not be reused until the associated memory has been fully reclaimed by the driver endpoint.

4.2 Bus Address Format

A Bus Address is a 64-bit value used by the device endpoint to reference a specific offset within a shared memory area.

Bus Addresses are defined by the virtio-msg FF-A bus and are opaque to the Virtio Device. They are translated by the device endpoint into a local memory pointer using internal tracking data for the corresponding area identifier.

The format of a Bus Address is:

- Area Identifier (16 bits): The identifier assigned by the driver when the shared memory area is shared
- Offset (48 bits): A byte offset from the start of the shared area

This format supports up to 65,536 shared memory areas per driver/device endpoint pair and allows addressing large areas without exposing FF-A handles or physical addresses to the device.

4.3 Sharing Shared Memory Areas

To make a shared memory area available to a Virtio Device, the driver endpoint initiates a memory share operation coordinated through the FF-A ABI and bus messages.

The sharing sequence proceeds as follows:

1. The virtio-msg FF-A bus driver allocates a unique Area Identifier and creates a shared memory handle using `FFA_MEM_SHARE`.
2. It then sends a `FFA_BUS_MSG_AREA_SHARE` message to the virtio-msg FF-A bus device, containing the area identifier, memory handle, and sharing attributes.
3. The virtio-msg FF-A bus device responds to the share message with a result of `Success` or `Error`.

If the memory cannot be shared successfully, the device endpoint must:

- Relinquish any partially mapped memory using the `FFA_MEM_RELINQUISH` ABI.
- Reply to the driver endpoint with a failure result in the share response.

Once completed successfully, the shared memory area becomes accessible to the device endpoint until it is explicitly unshared or released. The virtio-msg FF-A bus coordinates lifecycle management using the messages defined in this chapter.

The format and semantics of the `FFA_BUS_MSG_AREA_SHARE` message are defined in 7.5 *FFA_BUS_MSG_AREA_SHARE*.

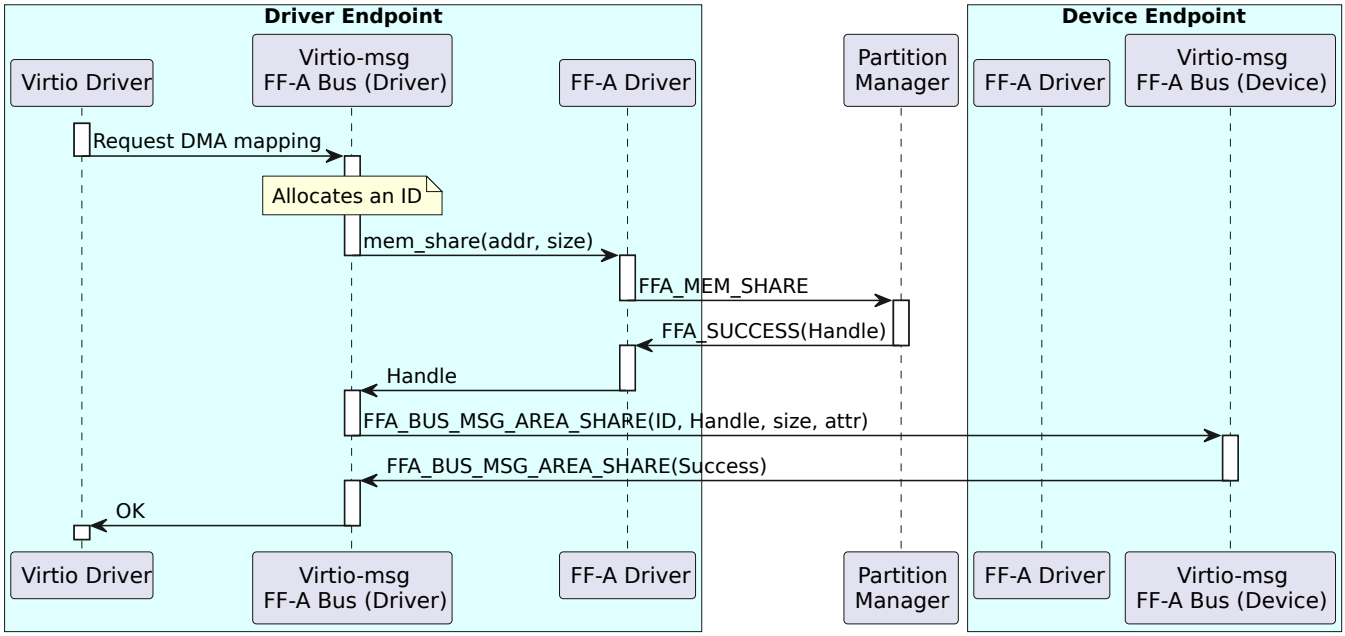

Figure 4.1: Memory Sharing Flow

4.4 Retrieving Shared Memory Areas

To retrieve and map a shared memory area, the virtio-msg FF-A bus device endpoint follows these steps:

1. The virtio-msg FF-A bus device invokes the `FFA_MEM_RETRIEVE_REQ` ABI to the Partition Manager, specifying the memory handle and attributes provided in the `FFA_BUS_MSG_AREA_SHARE` message.
2. The Partition Manager processes the request and responds with the `FFA_MEM_RETRIEVE_RESP` ABI, which includes the properties of the mapped area, such as virtual addresses and access permissions.
3. The virtio-msg FF-A bus device uses the information in the `FFA_MEM_RETRIEVE_RESP` ABI to establish local mappings for the shared memory area. Once mapped, the shared memory area is accessible to all virtio devices associated with the endpoint.

If the retrieval or mapping fails, the virtio-msg FF-A bus device must relinquish any partially mapped memory using the `FFA_MEM_RELINQUISH` ABI and handle the error as described in 4.7 *Shared Memory and Addressing Errors*. These steps ensure that the shared memory area is properly prepared for access by the Virtio Device.

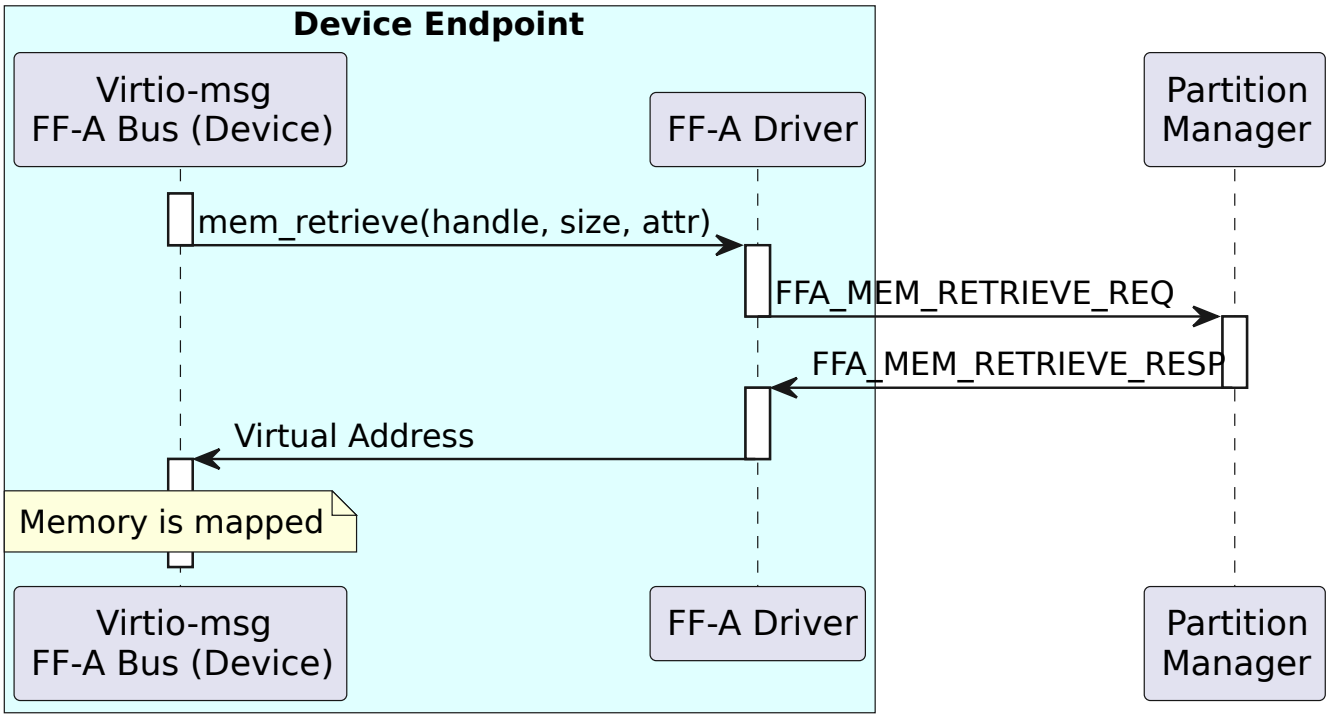

Figure 4.2: Memory Retrieval Flow

4.5 Driver Endpoint Initiated Unsharing

The driver endpoint may revoke access to a previously shared memory area by initiating an unsharing sequence.

To begin, the driver endpoint sends a `FFA_BUS_MSG_AREA_UNSHARE` message to the device endpoint. The device endpoint replies with:

- `Success`: The memory was relinquished. The driver endpoint must then reclaim the region using `FFA_MEM_RECLAIM`.
- `Busy`: The device endpoint is still using the memory. The driver endpoint must defer reclaiming the region until it receives a `FFA_BUS_EVENT_AREA_RELEASE` message from the device endpoint.

Reclaim may only proceed after a successful response or release notification.

This protocol ensures that memory is not reclaimed while still in active use. It allows the device endpoint to defer release until in-flight operations complete.

If the device endpoint fails to respond, or returns a malformed message, the driver endpoint must follow the recovery procedures defined in Chapter 6 *Operational Error Handling*.

The structure and encoding of `FFA_BUS_MSG_AREA_UNSHARE` are defined in 7.6 *FFA_BUS_MSG_AREA_UNSHARE*.

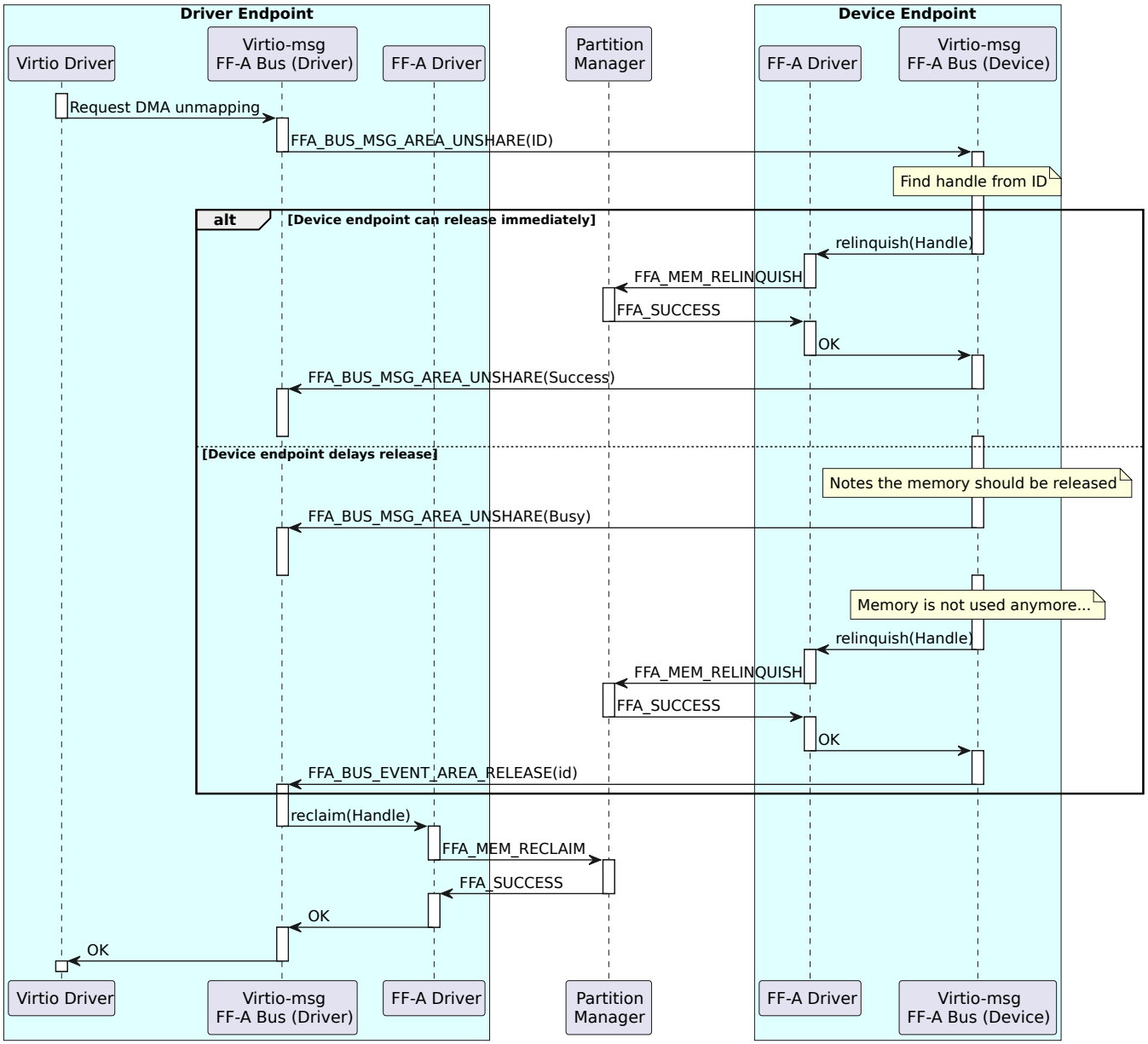

Figure 4.3: Driver-Initiated Unsharing Flow

4.6 Device Endpoint Initiated Release

A device endpoint may initiate the release of a shared memory area when it no longer requires access.

To do so, the device first relinquishes the memory using the `FFA_MEM_RELINQUISH` ABI.

It then sends a `FFA_BUS_EVENT_AREA_RELEASE` message to the driver endpoint, identifying the area identifier of the region that was released.

Upon receiving the release message, the driver endpoint must reclaim the memory region using `FFA_MEM_RECLAIM` ↪.

This mechanism allows the device endpoint to drive cleanup when the lifetime of a shared region is not directly controlled by the driver endpoint—for example, when buffers are temporary, or in the case of device-initiated teardown or fault recovery.

The device endpoint must emit this message in the following cases:

- When memory is relinquished without a preceding unshare request
- When a previous `FFA_BUS_MSG_AREA_UNSHARE` was responded to with `Busy`, and the memory has now been released

The structure and encoding of `FFA_BUS_EVENT_AREA_RELEASE` are defined in *7.7 FFA_BUS_EVENT_AREA_RELEASE*.

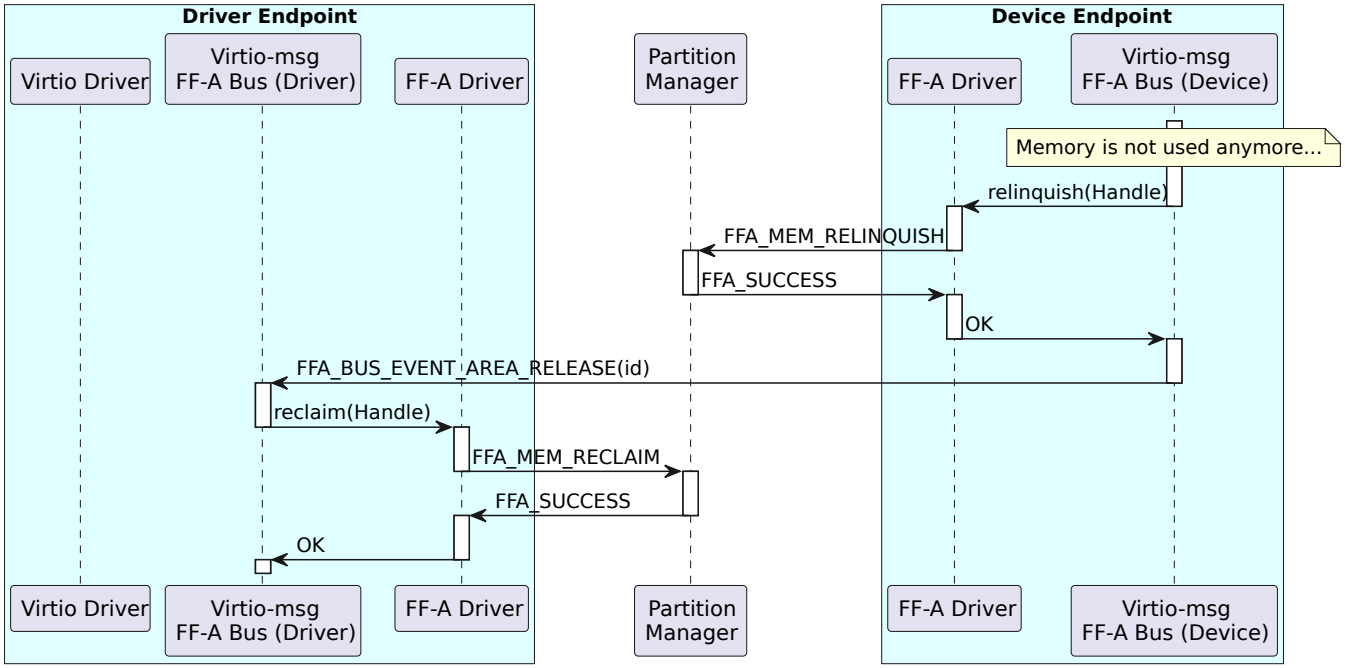

Figure 4.4: Device-Initiated Release Flow

4.7 Shared Memory and Addressing Errors

This section describes error conditions related to memory sharing, region access, and FF-A interactions. These errors are not part of the Virtio message protocol but may impact the correctness of buffer sharing and DMA operations.

4.7.1 Local Memory Operation Failures

Errors may occur during memory sharing or region access operations initiated by the driver or device. These errors are synchronous and must be reported directly to the caller. They are localized to the initiating component and are not considered protocol violations.

These include:

- `FFA_MEM_SHARE` or `FFA_MEM_RETRIEVE_REQ` returning an FF-A error.
- Failure of the device endpoint to retrieve the shared memory region.
- Platform-specific failures during memory mapping or unmapping (e.g., alignment, pinning, granule size, or resource exhaustion).

In these cases:

- The operation must return an error to the caller using local conventions.
- No fallback response is generated.
- No endpoint reset or device removal is required.
- Retry may only be attempted if the error is `FFA_BUSY`.

These errors do not affect the state of the Virtio message protocol or the liveness of the associated endpoint.

4.7.2 Transmission and Protocol Errors

The `FFA_BUS_MSG_AREA_SHARE` or `FFA_BUS_MSG_AREA_UNSHARE` messages used to coordinate memory sharing may also encounter delivery failures or protocol-level rejections.

Possible failure conditions include:

- Message delivery failure at the FF-A level (e.g., unreachable peer, dropped channel).
- Reception of a malformed or unsupported `FFA_BUS_MSG_AREA_SHARE` or `FFA_BUS_MSG_AREA_UNSHARE` message.
- `Error` result to `FFA_BUS_MSG_AREA_SHARE` or `FFA_BUS_MSG_AREA_UNSHARE` message.
- No response received for a message requiring acknowledgment.

An Error result may be used by the device endpoint to signal unsupported attributes, invalid area identifiers, or platform-specific rejection.

In such cases:

- The implementation must return an error to the requester.
- The memory region involved must be cleaned up locally if applicable, for example using `FFA_MEM_RECLAIM`.

These failures are isolated to the memory sharing operation and do not require fallback responses or transport-level recovery actions.

For comprehensive details on error handling, including protocol-visible error reporting and the use of `FFA_BUS_MSG_ERROR` for memory sharing failures, refer to Chapter 6 *Operational Error Handling*.

4.7.3 Invalid Bus Address Usage

Bus Address errors occur when the device endpoint attempts to resolve a Bus Address that is not valid within the current memory sharing context.

A device endpoint does not directly access memory through a Bus Address. Instead, it requests the virtio-msg FF-A bus device to translate a Bus Address into a local virtual address using the area identifier and offset. This

translation depends on internal metadata established during the memory retrieval phase.

Failures during this translation may occur in the following cases:

- The area identifier is unknown, no longer valid, or was never shared.
- The memory associated with the area identifier has been relinquished or reclaimed.
- The specified offset exceeds the bounds of the mapped region.
- The memory mapping was not completed successfully or is no longer active.

In these cases:

- The virtio-msg FF-A bus device must reject the translation request and return an error to the Virtio device.
- No memory access should be attempted using an invalid or unresolvable Bus Address.
- These errors are localized to the device endpoint and must not trigger a reset, fallback recovery, or FF-A level error signaling.

Such errors are typically indicative of logic bugs, stale data, or race conditions within the device endpoint or driver endpoint implementation. They are not considered violations of the virtio-msg FF-A bus protocol.

4.8 Memory Sharing Summary

This section illustrates a typical end-to-end memory sharing sequence used to expose a virtqueue from the Virtio Driver to the Virtio Device. It highlights the interaction between the virtio-msg transport, the FF-A driver, and the virtio-msg FF-A bus, and how shared memory becomes accessible to the Virtio device.

1. The Virtio Driver allocates a memory region to store a virtqueue structure.

Make memory accessible to the device endpoint

2. The Virtio Driver requests the virtio-msg FF-A bus driver to share this memory with the device endpoint.
3. The virtio-msg FF-A bus driver allocates a unique Area Identifier for the region.
4. The virtio-msg FF-A bus driver asks the FF-A driver to share the memory with the device.
5. The FF-A Driver issues a `FFA_MEM_SHARE` call to the Partition Manager and returns the resulting memory handle.
6. The FF-A Driver gets back a FF-A handle for the area.
7. The virtio-msg FF-A bus gets back the FF-A handle for the area.

Inform device endpoint of a new shared area

8. The virtio-msg FF-A bus driver sends a `FFA_BUS_MSG_AREA_SHARE` message to the virtio-msg FF-A bus device, providing the area identifier and memory handle.
9. The virtio-msg FF-A bus device responds to the share message with a `Success` result.
10. The virtio-msg FF-A bus driver constructs a Bus Address using the area identifier and offset and returns it to the Virtio Driver.

Inform device of a new virtqueue at bus address

11. The Virtio Driver sends a `VIRTIO_MSG_SET_VQUEUE` message to the Virtio Device, passing the Bus Address.
12. The Virtio Device requests translation of the Bus Address from the virtio-msg FF-A bus device.
13. The virtio-msg FF-A bus device invokes its FF-A Driver to get a virtual address for the FF-A handle corresponding to the area identifier.
14. The FF-A driver use `FFA_MEM_RETRIEVE_REQ` with the FF-A handle
15. The FF-A driver gets back an address for the area and maps it at a virtual address
16. The virtio-msg FF-A bus device gets back a virtual address for the area.
17. The virtio-msg FF-A bus device returns the virtual address corresponding to the Bus Address.

Virtqueue accessible to the Virtio Device

18. The Virtio Device accesses the virtqueue structure using the resolved virtual address.

This sequence is represented in the following flow diagram:

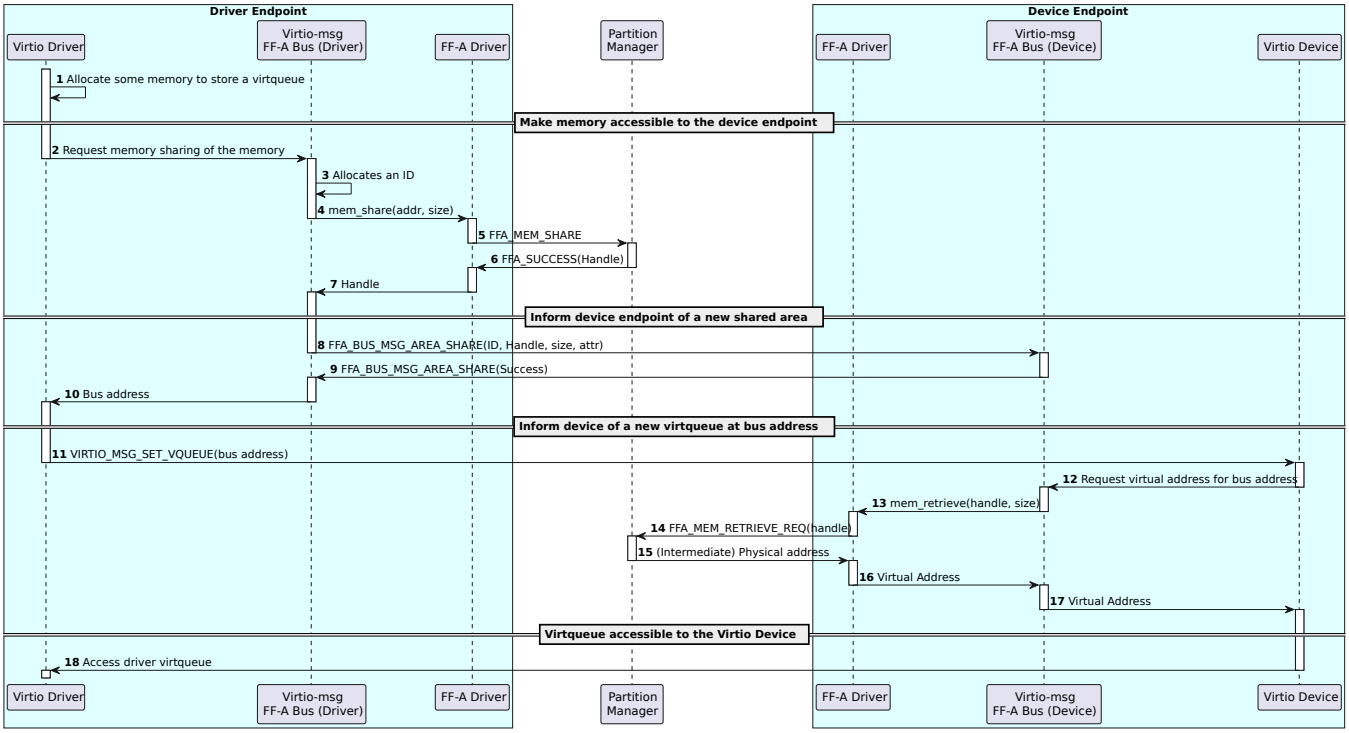

Figure 4.5: Virtqueue sharing Flow

Chapter 5
Monitoring and Hotplug

This chapter defines optional mechanisms for monitoring endpoint availability, detecting Virtio Device changes, and coordinating bus-level resets in Virtio Message Bus over FF-A systems.

These mechanisms allow systems to implement either static topologies with fixed endpoints, or dynamic configurations that support fault recovery, liveness tracking, and Virtio Device hotplug.

The following bus-level operations are supported:

- Liveness monitoring using `BUS_MSG_PING`
- Notification of Virtio Device addition or removal using `BUS_MSG_EVENT_DEVICE`
- Coordinated shutdown and reset of the bus using `FFA_BUS_MSG_RESET`

All endpoints must respond to ping messages. Support for sending ping or hotplug notifications is optional and determined by system capabilities and implementation policy.

5.1 Ping and Liveness Monitoring

The `BUS_MSG_PING` message allows an endpoint to verify that its peer is responsive. It is used for liveness tracking and recovery coordination, but has no functional effect on the state of the bus or any Virtio Device.

An `BUS_MSG_PING` message may be sent by either endpoint. The message contains a 32-bit opaque value that must be echoed back unmodified in the response payload. The response echoes `dev_num` and `msg_uid` (see Chapter 3 *Message Transfer*) and should echo `msg_op` unchanged.

Device endpoints may only initiate ping messages if they support an outbound message delivery mechanism. Whether ping is used for liveness monitoring is IMPLEMENTATION DEFINED and not negotiated at runtime.

If a ping does not receive a response within an IMPLEMENTATION DEFINED timeout, the sender may consider the remote endpoint unresponsive and trigger fault handling procedures internal to the bus implementation (see 5.3 *Bus Stop and Reset*).

The structure and encoding of `BUS_MSG_PING` are defined in the Virtio Specification [1].

5.2 Device Hotplug Support

The set of Virtio Devices exposed by a device endpoint is not required to be static. The `BUS_MSG_EVENT_DEVICE` message allows a device endpoint to notify a driver endpoint when the list of active Virtio Devices changes at runtime.

The message includes a device change **state**, which may be one of the following:

- `Device Ready`: A new Virtio Device has been added and is available for initialization.
- `Device Not Present`: A previously exposed Virtio Device has been removed.
- `No Data`: The device set has changed, but the device endpoint cannot identify which Virtio Devices were added or removed.

If the state is `Device Ready` or `Device Not Present`, the message includes the Device Number of the affected Virtio Device. If the state is `No Data`, the Device Number is set to 0. In this case, the driver endpoint must re-enumerate the Virtio Device list using `BUS_MSG_GET_DEVICES`.

This mechanism enables dynamic Virtio Device attach/detach and fallback notification when per-device tracking is not supported. It complements but does not replace the standard discovery and enumeration procedure.

No response must be sent to a `BUS_MSG_EVENT_DEVICE` message; it is an event message (see 3.7 *Transfer Method Selection*).

The device endpoint must emit this message if a Virtio Device is removed due to an unrecoverable internal fault.

The structure and encoding of `BUS_MSG_EVENT_DEVICE` are defined in the Virtio Specification [1].

5.3 Bus Stop and Reset

The FFA_BUS_MSG_RESET message allows a driver endpoint to halt all activity associated with a device endpoint and reset the state of the message bus connection.

This message is used to:

- Force a clean shutdown of the device endpoint during driver termination or suspend
- Recover from a fatal error or an unrecoverable protocol state detected by the virtio-msg FF-A bus
- Trigger complete reinitialization and device rediscovery

The message applies to all Virtio Devices exposed by the target device endpoint. It does not replace per-device reset procedures defined by the Virtio Specification (e.g., via SET_DEVICE_STATUS).

Upon receiving a FFA_BUS_MSG_RESET message, the device endpoint must perform the following actions:

- Terminate any active Virtio Device operation.
- Release all active memory mappings.
- Prepare to respond to subsequent version negotiation and device enumeration requests.

The device endpoint must reply with a response indicating Success or Error.

FFA_BUS_MSG_RESET is version-independent: the device endpoint accepts it regardless of the currently negotiated FF-A Bus Version or Transport Revision. On successful completion, the endpoint returns to the unnegotiated state and is ready to process FFA_BUS_MSG_VERSION as the next operation.

After sending FFA_BUS_MSG_RESET, the sender should not assume any outstanding operations are retained; all message correlation state is cleared.

The structure and encoding of FFA_BUS_MSG_RESET are defined in 7.8 *FFA_BUS_MSG_RESET*.

The bus implementation may initiate a reset in response to internal faults or persistent delivery failures, even if such failures are reported back to the virtio-msg transport.

Chapter 6
Operational Error Handling

This chapter defines how implementations must detect, classify, and report errors that occur during operation of the virtio-msg FF-A bus.

Errors may be:

- **Transient**, requiring retry-based recovery.
- **Fatal**, requiring removal of the affected Virtio Device or reset of the entire endpoint.
- **Transport-visible**, reported explicitly to the virtio-msg transport through error return codes or protocol messages.

All recovery mechanisms are defined in terms of FF-A message delivery, message semantics, and the interface between the virtio-msg transport and virtio-msg FF-A bus. Driver and device endpoints follow the same principles.

6.1 Error Taxonomy and Surfaces

Two error surfaces exist:

- Local interface errors returned immediately to the virtio-msg transport or Virtio Driver (e.g., allocation failure, exhausted in-flight capacity);
- Protocol-visible errors conveyed over FF-A as `FFA_BUS_MSG_ERROR` or as operation-specific result codes in a response payload.

Operation chapters define operation-specific result codes; this chapter centralizes shared correlation, classification, and escalation semantics.

6.2 Correlation and Protocol Error Responses

6.2.1 Correlation Errors

See the full correlation algorithm in Chapter 3 *Message Transfer*. This summary defines ignored vs terminating vs escalation triggers.

Ignored (optionally logged; no protocol-visible response):

- `msg_uid` is zero where a response was required.
- No in-flight entry matches the correlation key (unknown / already completed / duplicate / late / uncorrelatable `FFA_BUS_MSG_ERROR`).
- Invalid `dev_num` for a transport message (unknown / inactive device).

Termination (releases key and reports failure locally):

- Timeout waiting for a response.
- Receipt of `FFA_BUS_MSG_ERROR` for the in-flight request.
- FF-A transmission failure not resolved by retry (terminal return code or retry budget exhausted) when sending the original request or its mandatory response.
- Persistent FIFO enqueue failure after bounded retries.

Escalation candidates (feed into fatal classification):

- Repeated out-of-order responses (per-device ordering violation).
- Repeated malformed responses after a bounded IMPLEMENTATION DEFINED threshold.
- Message type bit (bus/transport) mismatch in a correlated response.

Exhausted in-flight capacity (no available non-zero `msg_uid`): apply local back-pressure (wait or fail locally). Not protocol-visible and not fatal by itself.

Event messages bypass correlation.

6.2.2 Protocol errors: delivery-method mismatch

When a request/response operation is received over a delivery method that is not permitted by the receiver for that direction, the device endpoint reports the failure using `FFA_BUS_MSG_ERROR` correlated to the original `msg_uid` ↪. The condition is non-fatal; the driver should retry using any receiver-permitted method discovered during negotiation.

A method-mismatch error is considered non-fatal; the sender retries the same operation using a permitted delivery method without resetting version or state.

6.2.3 Device-Side Error Response: FFA_BUS_MSG_ERROR

The `FFA_BUS_MSG_ERROR` message:

- Must only be sent as a response to a message that required a reply.
- Must not be used for event messages or bus control messages.
- Is emitted only by the virtio-msg FF-A bus at the device endpoint that processed (and failed) the original request.
- Must include the original `dev_num` and `msg_uid` copied unchanged from the failing request.
- Must include `original_msg_op`, echoing the failing request's `msg_op` for provenance (not part of the correlation key).
- Has a header `msg_op` value identifying the message as `FFA_BUS_MSG_ERROR`.
- Must not be followed by any additional response.

Correlation follows the algorithm in Chapter 3 *Message Transfer*. The driver first classifies the message (for an error this may require consulting `original_msg_op` to determine whether the failing operation was a bus or transport operation), then applies the appropriate key: `msg_uid` for bus messages or (`dev_num`, `msg_uid`) for

transport messages. The `original_msg_op` field participates only in classification; it is not part of the correlation key itself.

Transport message case: the bus reports a failure to the virtio-msg transport layer for the corresponding in-flight request.

Bus message case: the bus completes the in-flight bus request with failure locally.

Receipt of `FFA_BUS_MSG_ERROR` releases the correlation key immediately.

6.3 Retry-Based Recovery

Some FF-A operations may fail transiently with `FFA_BUSY`, indicating that the receiver or transport is temporarily unable to process the request. These conditions are not fatal and must be handled using retry-based recovery.

Retry-based recovery applies when an FF-A interface returns `FFA_BUSY` in response to the following operations:

- `FFA_MSG_SEND2` (indirect message)
- `FFA_MSG_SEND_DIRECT_REQ2` (direct message)
- `FFA_MEM_SHARE` (memory sharing request)

It also applies when FIFO-based delivery is blocked due to a full outbound FIFO and the sender is awaiting a space-available notification.

In all of these cases, the sender must:

- Wait for an IMPLEMENTATION DEFINED delay before retrying.
- Retry the operation for an IMPLEMENTATION DEFINED number of attempts.

If the operation does not complete after all retry attempts, the failure must be escalated. See 6.4 *Fatal Error Classification* and 6.5 *Transport-Level Error Reporting* for required behavior in that case.

6.4 Fatal Error Classification

Fatal errors occur when a virtio-msg operation fails in a way that prevents continued progress and cannot be resolved through retry. These conditions require removal of the affected Virtio Device or reset of the entire endpoint, depending on scope.

The following sections classify fatal error types and indicate the appropriate recovery action.

6.4.1 Device-Level Failures

These errors affect a specific Virtio Device. The Virtio Device must be removed as described in 6.8.1 *Device Removal*.

Fatal device-level failures include:

- A `BUS_MSG_EVENT_DEVICE` message is received indicating device removal.
- Persistent delivery failure or timeout for a request targeting one Virtio Device.
- Malformed or missing response to a message that expected a reply.
- Repeated `FFA_BUSY` on FIFO writes or FF-A messaging for a specific Virtio Device after retry attempts are exhausted.
- Protocol violations limited to a single Virtio Device (e.g., invalid response format, unexpected state).

If the failed request originated from the transport, the bus must return an error code. If the failed request was received over FF-A and required a reply, a `FFA_BUS_MSG_ERROR` must be sent. If the failed request was not a transport message (e.g., event or bus control), the error must be handled internally by the bus. If recovery is not possible, the Virtio Device must be removed.

6.4.2 Endpoint-Level Failures

These errors affect the entire remote FF-A endpoint. The endpoint must be reset as described in 6.8.2 *Endpoint Reset*.

Fatal endpoint-level failures include:

- All Virtio Devices on the remote endpoint have failed or been removed.
- FF-A calls return permanent error codes:
  - `FFA_DENIED`
  - `FFA_ABORT`
  - `FFA_INVALID_PARAMETERS`
  - `FFA_NOT_SUPPORTED`
- Multiple messages result in malformed or protocol-invalid responses.
- Persistent FIFO synchronization loss or setup failure (e.g., memory retrieval fails).
- The remote endpoint becomes unresponsive to all FF-A messaging.
- Retry-based recovery fails for operations that apply at endpoint scope (e.g., FIFO configuration).

As with device-level failures:

- If the request originated from the transport, the bus must return an error code.
- If the failed request was received over FF-A and required a reply, a `FFA_BUS_MSG_ERROR` must be sent.
- If the failed request was not a transport message, the bus must handle the fault internally and initiate a full endpoint reset if recovery is not possible.

6.5 Transport-Level Error Reporting

This section defines how transport-visible errors are reported between the virtio-msg transport and virtio-msg FF-A bus, and when the `FFA_BUS_MSG_ERROR` message must be used.

6.5.1 Bus-to-Transport Error Reporting

When the virtio-msg transport submits a request through the bus, and the bus cannot deliver it or obtain a valid response, it must return an error code to the transport through the local interface.

These conditions include:

- FF-A message delivery failure.
- No response received within an IMPLEMENTATION DEFINED timeout.
- A `FFA_BUS_MSG_ERROR` was received from the peer.
- The response was malformed or invalid.

The transport is responsible for interpreting the error and applying recovery policies.

6.5.2 Transport-to-Bus Error Feedback

When the bus receives a request on the device endpoint side that requires a reply, and forwards it to the transport, the transport may fail to process the message or generate a response.

In this case, the transport must return an error to the bus. The bus must then generate a `FFA_BUS_MSG_ERROR` message to report that the message cannot be answered.

6.6 Memory Sharing Error Mapping

Memory sharing failures (see 4.7 *Shared Memory and Addressing Errors*) classify as follows:

- Area share or retrieve permanent FF-A failure: classify as device-level; if the same failure occurs across all devices classify as endpoint-level.
- `AREA_UNSHARE` remains Busy and times out: classify as device-level.
- Malformed `AREA_SHARE` / `AREA_UNSHARE` / `AREA_RELEASE`: classify as device-level; if repeated across devices classify as endpoint-level.
- Invalid Bus Address translation attempts: local only (never escalate by themselves).

6.7 Event Configuration Failure Mapping

Event configuration failures map to recovery actions as follows:

- If `FFA_BUS_MSG_EVENT_CONFIGURE` returns an error, mark the device as not event-capable and continue with non-event operations.
- If all devices fail and events are required by policy, the implementation may escalate per an IMPLEMENTATION DEFINED policy but this is not automatically endpoint-level.

6.8 Reset Procedures

This section defines the required actions when the virtio-msg FF-A bus must remove a faulty Virtio Device or reset an entire endpoint. These recovery actions are triggered when a fatal error has been classified as requiring device-level or endpoint-level recovery (see 6.4 *Fatal Error Classification*).

6.8.1 Device Removal

When a Virtio Device is determined to be faulty or unreachable, it must be removed from the system. Device removal is scoped to a single Virtio Device and does not affect other Virtio Devices hosted by the same FF-A endpoint.

The driver-side bus must:

- De-register the Virtio Device from its internal device list.
- Reclaim all shared memory associated with the Virtio Device using FF-A mechanisms (e.g., `FFA_MEM_RECLAIM`).
- Release all protocol state associated with the Virtio Device.

The Virtio Driver must:

- Remove or unmap all shared buffers and memory regions associated with the Virtio Device.
- Follow platform-specific procedures for Virtio Device deregistration.

The device-side bus must:

- Stop accepting or responding to messages targeting the removed Virtio Device.
- Release any transport state related to the removed Virtio Device.

Device removal is final for the current session. The Virtio Device may be rediscovered in a later session based on platform policy.

6.8.2 Endpoint Reset

When an entire endpoint is no longer usable, the driver-side bus must reset the endpoint. This removes all Virtio Devices and protocol state associated with the remote partition.

The driver-side bus must:

- Remove all Virtio Devices associated with the endpoint.
- Reclaim all shared memory regions, including FIFO and Virtio Device buffers.
- Release all local protocol state.

If the remote endpoint is still responsive:

- Send a `FFA_BUS_MSG_RESET` message.
- Await a reply and optionally re-initiate version negotiation and device enumeration.

If the endpoint is no longer reachable:

- Perform local cleanup as described in 6.8.1 *Device Removal*.
- Do not attempt to send a reset message.

After reset, the bus must resume normal operation by waiting for rediscovery or initiating a new discovery cycle.

Chapter 7
Message Definitions

This chapter defines the message types and field layouts used by the virtio-msg FF-A bus. It specifies only the message formats that are specific to the FF-A transport binding.

Message semantics, usage rules, and delivery constraints are defined in the relevant chapters for discovery, memory sharing, transfer, hotplug, and error handling. This chapter does not restate those behaviors.

Generic virtio-msg transport messages are defined by the Virtio specification [1] and are not repeated here.

7.1 Message Operations

Each virtio message encodes an 8-bit `msg_op` (Message Operation) field in the standard message header (see 7.2 *Message Header and Field Encoding*). The Message Operation value selects the semantic meaning of the message. Values are defined in this section.

- Transport Message Operations are defined by the Virtio specification [1].
- Bus Message Operations are used by the virtio-msg FF-A bus to manage discovery, memory, and lifecycle.

The tables below list:

- Generic bus messages defined by the Virtio specification and used unmodified in this binding.
- FF-A-specific bus messages defined by the virtio-msg FF-A bus protocol.

Table 7.1: Generic bus Message Operations defined by the Virtio specification

Name	ID	Sender	Description
`BUS_MSG_GET_DEVICES`	`0x02`	Driver	Retrieve Device Numbers available on a device endpoint
`BUS_MSG_PING`	`0x03`	Any	Check that the other side is still alive
`BUS_MSG_EVENT_DEVICE`	`0x40`	Device	Notify a driver of a device state change or request re-enumeration

Table 7.2: FF-A-specific bus Message Operations defined by this binding

Name	ID	Sender	Description
`FFA_BUS_MSG_VERSION`	`0x80`	Driver	Version and capability negotiation
`FFA_BUS_MSG_AREA_SHARE`	`0x81`	Driver	Notify device endpoint of a newly shared memory area
`FFA_BUS_MSG_AREA_UNSHARE`	`0x82`	Driver	Request that device endpoint relinquish a shared memory area
`FFA_BUS_MSG_RESET`	`0x83`	Driver	Request to stop or reset the entire bus and all associated devices
`FFA_BUS_MSG_EVENT_POLL`	`0x84`	Driver	Request pending event messages
`FFA_BUS_MSG_EVENT_CONFIGURE`	`0x85`	Driver	Configure and enable events
`FFA_BUS_MSG_FIFO_CONFIGURE`	`0x86`	Driver	Configure FIFO-based Message Transfer
`FFA_BUS_MSG_ERROR`	`0x87`	Device	Indicate that a reply cannot be provided due to error
`FFA_BUS_EVENT_AREA_RELEASE`	`0xC0`	Device	Notify driver endpoint that an area has been released

7.2 Message Header and Field Encoding

All Virtio messages exchanged over FF-A use a common 8-byte header defined by the virtio-msg transport. This format is used consistently for both transport messages and bus messages.

The header fields are described below. Message-specific payload formats are defined in the corresponding sections for each message type.

All messages (transport and bus) are subject to the common constraints in 8.4 *Common Message Constraints*.

Table 7.3: Virtio message header format

Field	Size	Description
`type`	1 byte	• Message Type: – Bit[0]: ∗ b'0: Request. ∗ b'1: Response. – Bit[1]: ∗ b'0: Transport Message. ∗ b'1: Bus Message. – Bit[7:2]: Reserved (MBZ).
`msg_op`	1 byte	• Message Operation (see 7.1 *Message Operations*). Selects the operation semantics.
`dev_num`	2 bytes	• Identifies the Virtio Device or is 0 for bus messages.
`msg_uid`	2 bytes	• Message Unique Identifier chosen by the sender. – Echoed unchanged in any response. – MBZ if not used (e.g. event messages without reply). – Together with `dev_num` forms the correlation tuple.
`msg_size`	2 bytes	• Total length of the message in bytes, including the 8-byte header. Must be between 8 and 104 (see 3.2 *Message Size Constraints*).
Payload	Variable	• Message-type specific content, defined in the corresponding message definition table.

7.2.1 Field Encoding and Endianness

Unless otherwise stated:

- All multi-byte integer fields (header and payload) are encoded in **Little Endian** byte order.
- Boolean or bitfield values are packed into the least significant bits of the containing field as documented in their respective tables.
- Reserved (SBZ) bits and fields must be transmitted as zero and ignored on receipt.
- Future extensions must preserve the semantics of existing bits and fields when introducing new values.

7.3 FFA_BUS_MSG_VERSION

This message is used by a driver endpoint to initiate version negotiation with a device endpoint or to retrieve the currently negotiated version. It is also used by the device endpoint to reply.

Each endpoint independently declares the FF-A Bus Version (Major and Minor) and the Transport Revision it supports. The device endpoint includes its feature sets in the response. Negotiation is complete when both endpoints agree on a common Major version and select a compatible Minor version. See 2.2 *Version Negotiation* for FF-A Bus Version negotiation rules, backward compatibility, and fallback behavior.

Table 7.4: FF-A Bus version request message

Offset in bytes	**Size** in bytes	**Content**
0	8	• Message Header – type: b'10 (Bus Message Request) – msg_op: 0x80 (FFA_BUS_MSG_VERSION) – dev_num: Reserved (MBZ) – msg_size: 16
8	4	• FF-A Bus Version – Bit[31:16]: Major Version (Current 1) – Bit[15:0]: Minor Version (Current 0)
12	4	• Transport Revision (Current 1)

Table 7.5: FF-A Bus version response message

Offset in bytes	**Size** in bytes	**Content**
0	8	• Message Header – type: b'11 (Bus Message Response) – msg_op: 0x80 (FFA_BUS_MSG_VERSION) – dev_num: Reserved (MBZ) – msg_size: 26
8	4	• FF-A Bus Version – Bit[31:16]: Major Version (Current 1) – Bit[15:0]: Minor Version (Current 0)
12	4	• Transport Revision (Current 1)
16	4	• Feature Bits (Current 0)
20	4	• FF-A Bus Features (b'1 if supported sender, b'0 otherwise) – Bit[0]: Direct Message Reception – Bit[1]: Direct Message Transmission – Bit[2]: Indirect Message Reception – Bit[3]: Indirect Message Transmission – Bit[4]: FF-A Notifications Reception – Bit[5]: FF-A Notifications Transmission – Bit[6]: FIFO-based Message Transfer – Bit[31:7]: Reserved (MBZ)
24	2	• Maximum Number of Shared Memory Areas

7.4 FFA_BUS_MSG_EVENT_CONFIGURE

This message is sent by the driver endpoint to configure the event delivery mechanism used by the device endpoint to send asynchronous event messages to the driver endpoint.

The message must be sent after device enumeration and before any event messages may be emitted by the device endpoint. No device-to-driver event message may be sent prior to receiving a valid `FFA_BUS_MSG_EVENT_CONFIGURE`. If this message is not received, all device-generated events must remain queued and must not be emitted using any delivery method.

This message acts as both an instruction to activate event delivery and a configuration of the permitted delivery method on the device endpoint.

See 2.5 *Event Delivery Configuration* for the behavior associated with this message and event delivery selection rules. Delivery method availability is determined based on FF-A Bus Feature Flags exchanged in `FFA_BUS_MSG_VERSION` (see Chapter 2 *Discovery*).

The device endpoint must respond to confirm whether the event configuration was successful.

Table 7.6: FF-A Bus Event Configure request message

Offset in bytes	**Size** in bytes	**Content**
0	8	• Message Header – type: b'10 (Bus Message Request) – msg_op: 0x85 (FFA_BUS_MSG_EVENT_CONFIGURE) – dev_num: Reserved (MBZ) – msg_size: 12
8	1	• Event Delivery Selection: – 0: Polling – 1: Notification-assisted polling – 2: Indirect message transfer – 3: FIFO-based message transfer – Other values: Reserved
9	1	• Reserved (MBZ)
10	2	• Notification ID – Must be zero unless selection = 1 – Otherwise: Valid FF-A Notification ID

Table 7.7: FF-A Bus Event Configure response message

Offset in bytes	**Size** in bytes	**Content**
0	8	• Message Header – type: b'11 (Bus Message Response) – msg_op: 0x85 (FFA_BUS_MSG_EVENT_CONFIGURE) – dev_num: Reserved (MBZ) – msg_size: 10
8	2	• Result – 0x0: Success – 0x1: Error – Others: Reserved.

7.5 FFA_BUS_MSG_AREA_SHARE

This message is sent by the driver endpoint to notify the device endpoint that a Shared Memory Area has been shared using `FFA_MEM_SHARE`. It includes the Area Identifier, memory handle, size, and sharing attributes.

The device endpoint must respond to confirm whether the region was successfully retrieved.

See Chapter 4 *Memory Sharing* for memory sharing behavior and associated error conditions.

Table 7.8: FF-A Area share request message

Offset in bytes	**Size** in bytes	**Content**
0	8	• Message Header – type: b'10 (Bus Message Request) – msg_op: 0x81 (FFA_BUS_MSG_AREA_SHARE) – dev_num: Reserved (MBZ) – msg_size: 34
8	2	• Area Identifier
10	8	• FF-A Memory Handle of the Area
18	8	• FF-A Memory Tag of the Area
26	4	• Total Number of Pages in the Area
30	4	• Sharing Attributes (See Table 7.10 for details)

Table 7.9: FF-A Area share response message

Offset in bytes	**Size** in bytes	**Content**
0	8	• Message Header – type: b'11 (Bus Message Response) – msg_op: 0x81 (FFA_BUS_MSG_AREA_SHARE) – dev_num: Reserved (MBZ) – msg_size: 12
8	2	• Area Identifier
10	2	• Result – 0x0: Success – 0x1: Error – Others: Reserved.

Table 7.10: Sharing Attributes of an Area

Bits	Description
1:0	• Sharing Type – b'00: Share – b'01: Lend – b'10: Donate – b'11: Reserved
2	• Writeable – b'0: Read-Only memory – b'1: Read-Write memory
3	• Executable – b'0: Not executable memory – b'1: Executable memory
5:4	• Shareability – b'00: Non-shareable – b'01: Reserved – b'10: Outer shareable – b'11: Inner shareable
7:6	• Cacheability Attributes if Normal memory (Bit[9:8] = b'10) – b'00: Reserved – b'01: Non-cacheable – b'10: Reserved – b'11: Write-Back • Device Memory Attributes if device memory (Bit[9:8] = b'01) – b'00: Device nGnRnE – b'01: Device-nGnRE – b'10: Device-nGRE – b'11: Device-GRE • Reserved if Bit[9:8] = b'00 or b'11
9:8	• Memory Type – b'00: Not specified – b'01: Device memory – b'10: Normal memory – b'11: Reserved
10	• NS-bit – b'0: Secure memory – b'1: Non-secure memory
31:11	• Reserved (MBZ)

7.6 FFA_BUS_MSG_AREA_UNSHARE

This message is sent by the driver endpoint to request that the device endpoint relinquish access to a previously shared Area.

The response indicates whether the region was successfully released (`Success`) or is still in use (`Busy`), in which case the device will send a `FFA_BUS_EVENT_AREA_RELEASE` when it is ready to complete the operation.

See Chapter 4 *Memory Sharing* for unsharing behavior and associated error conditions.

Table 7.11: FF-A Area unshare request message

Offset in bytes	**Size** in bytes	**Content**
0	8	• Message Header – type: b'10 (Bus Message Request) – msg_op: 0x82 (FFA_BUS_MSG_AREA_UNSHARE) – dev_num: Reserved (MBZ) – msg_size: 10
8	2	• Area Identifier

Table 7.12: FF-A Area unshare response message

Offset in bytes	**Size** in bytes	**Content**
0	8	• Message Header – type: b'11 (Bus Message Response) – msg_op: 0x82 (FFA_BUS_MSG_AREA_UNSHARE) – dev_num: Reserved (MBZ) – msg_size: 12
8	2	• Area Identifier
10	2	• Result – 0x0: Success – 0x1: Error – 0x2: Busy – Others: Reserved.

7.7 FFA_BUS_EVENT_AREA_RELEASE

This event message is sent by the device endpoint to inform the driver endpoint that a previously shared Area has been relinquished and can now be reclaimed. It does not expect and must not receive a response.

See Chapter 4 *Memory Sharing* for release sequencing and memory region lifecycle.

Table 7.13: FF-A Area release message

Offset in bytes	**Size** in bytes	**Content**
0	8	• Message Header – type: b'10 (Bus Message Request) – msg_op: 0xC0 (FFA_BUS_EVENT_AREA_RELEASE) – dev_num: Reserved (MBZ) – msg_size: 10
8	2	• Area Identifier

7.8 FFA_BUS_MSG_RESET

This message is sent by the driver endpoint to instruct the device endpoint to stop all activity, release all resources, and prepare for reinitialization.

The device must respond with a result of `Success` or `Error`.

See 5.3 *Bus Stop and Reset* for reset behavior and conditions under which a reset may be issued.

Table 7.14: FF-A bus reset request message

Offset in bytes	**Size** in bytes	**Content**
0	8	• Message Header – type: b'10 (Bus Message Request) – msg_op: 0x83 (FFA_BUS_MSG_RESET) – dev_num: Reserved (MBZ) – msg_size: 8

Table 7.15: FF-A bus reset response message

Offset in bytes	**Size** in bytes	**Content**
0	8	• Message Header – type: b'11 (Bus Message Response) – msg_op: 0x83 (FFA_BUS_MSG_RESET) – dev_num: Reserved (MBZ) – msg_size: 10
8	2	• Result – 0x0: Success – 0x1: Error – Others: Reserved.

7.9 FFA_BUS_MSG_EVENT_POLL

This message is sent by the driver endpoint to retrieve a pending event message from a device endpoint that cannot deliver events asynchronously.

The response contains either a full event message or the `FFA_BUS_MSG_EVENT_POLL` response message if no event is pending.

See 3.4.4.2 *Device-initiated Event Messages* for polling conditions and delivery rules.

Table 7.16: FF-A Virtio Event poll request message

Offset in bytes	**Size** in bytes	**Content**
0	8	• Message Header – type: b'10 (Bus Message Request) – msg_op: 0x84 (FFA_BUS_MSG_EVENT_POLL) – dev_num: Reserved (MBZ) – msg_size: 8

Table 7.17: FF-A Virtio Event poll response message

Offset in bytes	**Size** in bytes	**Content**
0	8	• Message Header – type: b'11 (Bus Message Response) – msg_op: 0x84 (FFA_BUS_MSG_EVENT_POLL) – dev_num: Reserved (MBZ) – msg_size: 8

7.10 FFA_BUS_MSG_FIFO_CONFIGURE

This message is sent by the driver endpoint to configure FIFO-based Message Transfer with a device endpoint. It specifies a memory area previously shared using `FFA_MEM_SHARE`, the number of pages composing the region, and a Notification ID to be used by the device.

The device endpoint must respond with a result and its own Notification ID to be used by the driver.

Both endpoints must provide a valid FF-A Notification ID. See 3.6 *FIFO-based Message Transfer* for detailed behavior.

Table 7.18: FF-A FIFO-based Message Transfer configuration request

Offset in bytes	**Size** in bytes	**Content**
0	8	• Message Header – type: b'10 (Bus Message Request) – msg_op: 0x86 (FFA_BUS_MSG_FIFO_CONFIGURE) – dev_num: Reserved (MBZ) – msg_size: 22
8	8	• FF-A Memory Handle
16	2	• Page Count
18	2	• Driver Notification ID

Table 7.19: FF-A FIFO-based Message Transfer configuration response

Offset in bytes	**Size** in bytes	**Content**
0	8	• Message Header – type: b'11 (Bus Message Response) – msg_op: 0x86 (FFA_BUS_MSG_FIFO_CONFIGURE) – dev_num: Reserved (MBZ) – msg_size: 12
8	2	• Result – 0x0: Success – 0x1: Error – Others: Reserved.
10	2	• Device Notification ID

7.11 FFA_BUS_MSG_ERROR

This message is sent by the device endpoint when it cannot produce a normal response for a request that required one. It replaces the normal response and is only valid in that context. Only the device-side virtio-msg FF-A bus emits this message.

Header fields `dev_num` and `msg_uid` are copied from the failing request for correlation (see Chapter 3 *Message Transfer*). The `original_msg_op` field carries the operation code of the request that triggered the error; correlation relies only on the header tuple.

This message is never generated for event messages or any operation that does not expect a reply. See Chapter 6 *Operational Error Handling* for usage rules and classification.

Table 7.20: FF-A Bus Error response message

Offset in bytes	**Size** in bytes	**Content**
0	8	• Message Header – type: b'11 (Bus Message Response) – msg_op: 0x87 (FFA_BUS_MSG_ERROR) – msg_size: 10
8	2	• original_msg_op – Operation code of the request that caused the error

Chapter 8
Compliance

This chapter defines the compliance requirements for implementations of the virtio-msg FF-A bus. Each requirement corresponds to an externally observable behavior that ensures interoperability between drivers and devices.

Compliance is defined separately for:

- FF-A drivers, which expose message-passing and memory-sharing primitives
- Driver endpoints, which discover, bind, and manage Virtio devices
- Device endpoints, which expose Virtio devices and handle transport interactions

Internal implementation details, such as how Virtio drivers or devices are integrated into a software stack, are out of scope.

8.1 FF-A Driver Compliance Requirements

A compliant FF-A driver must implement the following interfaces defined in FF-A v1.2 [2]. Each operation must conform to its specified behavior, including return values, retry handling, and resource management.

- **Discovery operations**
  - Must implement at least one of `FFA_PARTITION_INFO_GET` or `FFA_PARTITION_INFO_GET_REGS` to expose endpoint UUIDs during discovery.
- **Direct messaging**
  - Must support `FFA_MSG_SEND_DIRECT_REQ2`.
  - Must support `FFA_MSG_SEND_DIRECT_RESP2`.
- **Indirect messaging and buffer management**
  - Must support `FFA_RXTX_MAP`.
  - May support `FFA_RXTX_UNMAP`.
  - Must support `FFA_MSG_SEND2`.
  - Must support `FFA_RX_RELEASE`.
  - Must support `FFA_NOTIFICATION_INFO_GET`.
  - Must support `FFA_NOTIFICATION_GET`.
- **Memory sharing and reclaiming**
  - Must support `FFA_MEM_SHARE`.
  - Must support `FFA_MEM_RETRIEVE_REQ`.
  - Must support `FFA_MEM_RETRIEVE_RESP`.
  - Must support `FFA_MEM_RELINQUISH`.
  - Must support `FFA_MEM_RECLAIM`.
- **Notification delivery**
  - Must support `FFA_NOTIFICATION_BIND`.
  - Must support `FFA_NOTIFICATION_SET`.
  - Must support `FFA_NOTIFICATION_GET`.
- **Operational guarantees**
  - Must allow a single RX buffer to be registered for the calling partition.
  - Must allow concurrent use of direct and indirect messaging interfaces.
  - Must expose error information sufficient for the virtio-msg FF-A bus driver to distinguish `FFA_BUSY` (retryable) from other errors.

8.2 virtio-msg FF-A Bus Device Compliance

A compliant Virtio Message Bus over FF-A device endpoint must implement the following observable behaviors:

- **Version negotiation**
  - Must accept and respond to `FFA_BUS_MSG_VERSION` as the first protocol message.
  - Must include Feature Bits and FF-A Bus Features in every version response.
  - Must declare supported message transfer methods in each version response.
  - Must declare supported event delivery capabilities in each version response.
  - Must consider negotiation complete only when echoing the driver proposal `(version, revision)` with Feature Bits and FF-A Bus Features.
  - Must treat `(0, 0)` after completion purely as a status query.
  - Must reject any different `(version, revision)` pair after completion by returning `(0, 0)` without altering negotiated state.
  - Must not change negotiated state except after a bus reset.
  - Must ignore or reject any non-version message received before negotiation completes.
  - May reply to a pre-negotiation non-version message with a bus response using `msg_op = 0` and no payload.
  - May silently drop a pre-negotiation non-version message, including an event message.
  - Must not send `FFA_BUS_MSG_ERROR` solely for a failed version proposal.
- **Device discovery and enumeration**
  - Must respond to `BUS_MSG_GET_DEVICES`.
  - Must respond to `VIRTIO_MSG_GET_DEVICE_INFO` for every advertised Device Number.
- **Transport message forwarding**
  - Must forward all transport messages without modification.
  - Must preserve the full message payload including all field values and encodings.
- **FIFO-based messaging**
  - Must respond to `FFA_BUS_MSG_FIFO_CONFIGURE` with `Success` only after the FIFO region is retrieved and notifications are configured.
  - Must respond to `FFA_BUS_MSG_FIFO_CONFIGURE` with `Error` if the memory cannot be retrieved or notifications cannot be bound.
  - Must retrieve the FIFO region using `FFA_MEM_RETRIEVE_REQ` before responding.
  - Must bind a notification ID using `FFA_NOTIFICATION_BIND` for incoming messages.
  - Must signal the driver with `FFA_NOTIFICATION_SET` when new FIFO messages are available.
  - Must not read from the FIFO unless a complete message is available.
  - Must use the configured notification mechanism to detect newly available FIFO messages.
  - Must align FIFO entries to `uint64_t` boundaries.
  - Must reject any misaligned FIFO message.
  - Must invalidate previously used FIFO memory after a reset or rebind.
  - Must accept only a new FIFO configuration established via `FFA_BUS_MSG_FIFO_CONFIGURE` after reset or rebind.
- **Memory sharing**
  - Must retrieve shared regions using `FFA_MEM_RETRIEVE_REQ`.
  - Must respond to `FFA_BUS_MSG_AREA_SHARE` with `Success` or `Error`.
  - Must respond to `FFA_BUS_MSG_AREA_UNSHARE` with `Success` or `Busy`.
  - Must send `FFA_BUS_EVENT_AREA_RELEASE` after deferred unsharing.
- **Event delivery**
  - Must respond to `FFA_BUS_MSG_EVENT_POLL`.
  - Must return at most one event per `FFA_BUS_MSG_EVENT_POLL`

  - Must return a FFA_BUS_MSG_EVENT_POLL‘ response when no events remain.
  - Must not make event messages visible before a valid `FFA_BUS_MSG_EVENT_CONFIGURE` is received.
  - Must, after configuration, write event messages to the outbound FIFO and emit a notification when FIFO-based transfer is selected.
  - Must, after configuration, send event messages asynchronously with `FFA_MSG_SEND2` when indirect transfer is selected.
  - Must, after configuration, queue event messages and emit a notification when notification-assisted polling is selected.
  - Must, after configuration, queue event messages without emitting notifications when pure polling is selected.
  - Must constrain only device-originated event messages by the selection.
  - May allow the driver to use any supported transfer method for outbound driver messages.

- **Ping and reset**
  - Must respond to `BUS_MSG_PING`.
  - Must respond to `FFA_BUS_MSG_RESET` with `Success` or `Error`.
  - Must release all Virtio device state and memory regions on reset.

- **Error reporting**
  - Must return an error to the bus if the device transport cannot process a request or provide a valid reply.
  - Must send `FFA_BUS_MSG_ERROR` only as a response to a message requiring a reply.
  - Must not send `FFA_BUS_MSG_ERROR` for event or control one-way messages.

- **Fault reporting**
  - Must emit `BUS_MSG_EVENT_DEVICE` with `Device Not Present` when a device is removed due to failure.
  - May send `VIRTIO_MSG_EVENT_CONFIG` with `NEED_RESET` to indicate device failure.

- **Correlation & ordering**:
  - Must treat any message with `msg_uid = 0` as non-correlatable (event / one-way).
  - Must never issue a response with `msg_uid = 0` if the request carried a non-zero `msg_uid`.
  - Must emit responses in request submission order per Device Number for transport messages.
  - Must ignore (and optionally log) a response whose correlation tuple does not match any in-flight request.
  - Must classify an out-of-order response as a device error and make it available for escalation logic and must not attribute it to any request.

- **Retry & back-pressure**:
  - Must not internally spin indefinitely on `FFA_BUSY`; back-off policy is IMPLEMENTATION DEFINED but bounded.
  - Must not allocate or retain correlation entries after reporting a terminal failure or timeout.
  - Must not drop an in-flight request silently; terminating conditions must be surfaced (local error path or `FFA_BUS_MSG_ERROR`).

- **Error handling & escalation (device)**:
  - Must send `FFA_BUS_MSG_ERROR` only for failed requests that required a reply and include unchanged `dev_num`, `msg_uid`, and `original_msg_op`.
  - Must not send `FFA_BUS_MSG_ERROR` for event or control messages without mandatory reply.
  - Must classify correlation errors: ignored (unknown tuple / duplicate / `msg_uid=0` where reply expected), terminating (timeout, received `FFA_BUS_MSG_ERROR`, terminal FF-A error, exhausted retry, persistent FIFO enqueue failure), escalation candidates (repeated malformed responses, ordering violations, type-bit mismatches).
  - Must record per-device ordering violations for escalation logic.
  - Must apply local back-pressure (not protocol-visible error) when in-flight `msg_uid` capacity is exhausted.

  - Must trigger device-level removal or endpoint reset per centralized fatal classification rules when termination conditions persist systematically.

- **Optional features**:
  - May send `BUS_MSG_PING` if indirect messaging is available and negotiated.
  - May send `BUS_MSG_EVENT_DEVICE` for hotplug or device removal.
  - May support both direct and indirect messaging if permitted by FF-A configuration.

8.3 virtio-msg FF-A Bus Driver Compliance

A compliant Virtio Message Bus over FF-A driver endpoint must implement the following observable behaviors:

- **Version negotiation**
  - Must begin each association by sending `FFA_BUS_MSG_VERSION(0, 0)` or a targeted proposal to discover or propose supported values.
  - Must apply downgrade ordering when proposals are rejected (see 2.2 *Version Negotiation*) until an acceptable pair is found or exhausted.
  - Must treat `(0, 0)` after completion as a status query only.
  - Must not attempt to change the negotiated pair without a bus reset.
  - Must not send any non-version message before negotiation completes.
  - Must not attempt further negotiation after any other protocol message is exchanged post-completion.
  - Must apply per-message semantics as specified in 2.2.3 *Version Message Rules*.
- **Endpoint discovery and device binding**
  - Must discover endpoints via an FF-A discovery mechanism.
  - Must identify candidate endpoints by matching the protocol UUID.
  - Must skip endpoints that respond with `version = 0` or fail negotiation.
- **Device enumeration and metadata retrieval**
  - Must send `BUS_MSG_GET_DEVICES` to enumerate Device Numbers.
  - Must send `VIRTIO_MSG_GET_DEVICE_INFO` for each discovered device.
- **Transport message forwarding**
  - Must forward all transport messages without modification.
  - Must preserve the full message payload including all field values and encodings.
- **FIFO-based messaging**
  - Must allocate and initialize inbound and outbound FIFO structures.
  - Must share the FIFO region using `FFA_MEM_SHARE`.
  - Must bind a notification ID using `FFA_NOTIFICATION_BIND` for device-to-driver signaling.
  - Must send `FFA_BUS_MSG_FIFO_CONFIGURE` with memory handle, layout, and notification ID.
  - Must write request messages into the FIFO only after successful configuration.
  - Must wait for device responses signaled via `FFA_NOTIFICATION_SET`.
  - Must not overwrite unread FIFO data.
  - Must defer transmission and retry later if the FIFO is full.
  - Must use the configured notification mechanism to detect available space.
  - Must align FIFO entries to `uint64_t` boundaries.
  - Must respect alignment constraints imposed by the message format.
  - Must invalidate any previously configured FIFO after device reset or rebind.
  - Must reconfigure a FIFO after reset or rebind using `FFA_BUS_MSG_FIFO_CONFIGURE`.
- **Memory sharing and revocation**
  - Must share memory regions using `FFA_MEM_SHARE` and `FFA_BUS_MSG_AREA_SHARE`.
  - Must revoke memory using `FFA_BUS_MSG_AREA_UNSHARE`.
  - Must reclaim memory with `FFA_MEM_RECLAIM` after receiving `FFA_BUS_EVENT_AREA_RELEASE`.
- **Event delivery and polling**
  - Must use `FFA_BUS_MSG_EVENT_POLL` to retrieve events when polling or notification-assisted polling was selected.
  - Must repeat `FFA_BUS_MSG_EVENT_POLL` immediately after any non-empty event response and continue until an empty response is received.

  - Must stop polling after the first empty response and only resume upon a subsequent notification (if notifications are used) or at the next scheduled polling interval.
  - Must, after device enumeration completes, send `FFA_BUS_MSG_EVENT_CONFIGURE` selecting exactly one delivery method.
  - Must provide a valid notification ID when notification-assisted polling is selected.
  - Must set the notification ID to zero when notification-assisted polling is not selected.
  - Must complete FIFO configuration before sending the configure message if FIFO-based transfer is selected.
  - Must constrain only device-originated event messages by the selection.
  - May use any supported transfer method for outbound driver messages.
  - Must respond to `BUS_MSG_EVENT_DEVICE` by updating device state.
  - May re-enumerate devices after processing `BUS_MSG_EVENT_DEVICE`.

- **Ping and reset**
  - Must respond to `BUS_MSG_PING`.
  - Must send `FFA_BUS_MSG_RESET` to reset a device endpoint.
  - Must rediscover and rebind devices after a successful reset.

- **Error reporting**
  - Must return an error code to the transport if a transport-initiated operation fails to complete.
  - Must treat a received `FFA_BUS_MSG_ERROR` as a terminating outcome for the correlated request.
  - Must report an error to the transport for a failed transport-originated request.
  - Must apply centralized recovery rules for failed bus-originated requests.

- **Error recovery and fallback**
  - Must apply bounded retry for `FFA_BUSY` on permitted FF-A primitives.
  - Must apply bounded retry for full FIFO enqueue.
  - Must classify and surface a terminal failure after retry exhaustion.
  - Must release a correlation entry upon timeout.
  - Must release a correlation entry upon receiving an error response.
  - Must release a correlation entry upon a terminal FF-A return code.
  - Must not reuse a non-zero `msg_uid` while its request is in flight.
  - Must treat an unexpected `FFA_BUS_MSG_ERROR` with unknown correlation data as ignored unless repeated.

- **Correlation and ordering**:
  - Must allocate `msg_uid` values (non-zero) uniquely among in-flight requests per domain (bus vs per-device transport table).
  - Must classify incoming messages per operation (`msg_op`) before correlation; must use `original_msg_op` ↪ only for classifying `FFA_BUS_MSG_ERROR`.
  - Must ignore a response with `msg_uid = 0` that claims to answer a correlated request.
  - Must detect and record ordering violations (out-of-order responses) per Device Number.
  - Must not deliver payload of an out-of-order response to higher layers prior to classification decision.

- **Reset and cleanup**:
  - Must abandon all in-flight requests on device removal or endpoint reset and release their correlation keys.
  - Must re-run version negotiation after any completed endpoint reset.
  - Retry indirect messages after `FFA_BUSY`, subject to a bounded policy
  - If retries fail, the virtio-msg FF-A bus may initiate reset or mark the endpoint as unavailable

- **Error handling & escalation (driver)**:
  - Must return a local error to the transport for terminated transport-originated requests.
  - Must treat `FFA_BUS_MSG_ERROR` as a terminating outcome for the correlated request.

  - Must apply bounded retry only to permitted FF-A operations (`FFA_MSG_SEND2`, `FFA_MSG_SEND_DIRECT_REQ2` ↪, `FFA_MEM_SHARE`) and full FIFO enqueue; on exhaustion classify failure.
  - Must classify correlation outcomes identically to device rules (ignored / terminating / escalation candidates) and feed escalation candidates fatal classification.
  - Must not fabricate protocol-visible errors for `msg_uid` exhaustion; apply local back-pressure instead.
  - Must initiate device removal or endpoint reset per fatal classification taxonomy when escalation thresholds are met.

- **Optional features**:

  - May send `BUS_MSG_PING` to monitor device endpoint liveness.
  - May use `FFA_BUS_MSG_EVENT_POLL` as a fallback delivery model.
  - May support both direct and indirect messaging if platform permits.

8.4 Common Message Constraints

This section defines normative constraints that apply uniformly to EVERY Virtio message (transport or bus) exchanged over FF-A, independent of the delivery mechanism (direct, indirect, FIFO, notification-assisted, or polling). All compliant endpoints (driver and device) must implement and honor these rules.

- **General size and layout**:
  - Must use the common 8-byte header defined in Chapter 7 *Message Definitions*.
  - Must set `msg_size` between 8 and 104 bytes inclusive.
  - Must ensure `msg_size` equals the actual number of bytes transmitted.
  - Must zero-fill any unused payload bytes up to 104 bytes.
  - Must ignore zero-filled unused payload bytes on receipt.
  - Must treat as an error any message with `msg_size < 8`, `msg_size > 104`, or a declared/actual length mismatch.
- **Correlation and identifiers**:
  - Must use `msg_uid` as the bus-domain correlation key.
  - Must use (`dev_num`, `msg_uid`) as the transport-domain correlation key.
  - Must treat `original_msg_op` in `FFA_BUS_MSG_ERROR` as classification-only.
  - Must not include `original_msg_op` in any correlation key.
  - Must echo `dev_num` and `msg_uid` unchanged in every correlated response.
  - Must set `msg_uid = 0` for event or one-way messages.
  - Must not use `msg_uid = 0` in a correlated response.
  - Must not reuse a correlation key while its request is in flight.
  - Must ignore a response whose tuple does not match any in-flight request.
- **Message operation (`msg_op`) handling**:
  - Must use the same `msg_op` in a successful response as in the request.
  - May send `FFA_BUS_MSG_ERROR` instead of an operation-specific response.
  - Must not treat the differing `msg_op` of `FFA_BUS_MSG_ERROR` as a correlation failure.
  - Must not change `msg_op` in a response except when sending `FFA_BUS_MSG_ERROR`.
- **Zero-fill and robustness**:
  - Must accept trailing zero padding up to the 104-byte limit.
  - Must not treat trailing zero padding as an error.
  - Must not transmit non-zero padding in unused payload space.
- **Reserved bits and forward compatibility**:
  - Must transmit reserved / MBZ header or payload bits as zero.
  - Must ignore reserved / MBZ bits on receipt.
  - Must not redefine existing header field semantics in future extensions.

Chapter 9
Appendix

9.1 FIFO Message Format

This appendix defines the in-memory format of a lockless FIFO used for message exchange in FIFO-based Message Transfer (see 3.6 *FIFO-based Message Transfer*). It specifies the memory layout of a single FIFO instance, including its header and message area. The FIFO format enables asynchronous communication between a single writer and a single reader, without requiring locks or atomic operations.

This format is versioned and self-describing. It may be superseded by a future standard defined by the Virtio specification.

9.1.1 Layout and Structure

Each FIFO instance occupies a contiguous memory region containing:

- A header structure describing the format version, message size, depth, and queue state.
- A circular array of fixed-size message entries.

All multi-byte fields are encoded in little-endian order. Padding and alignment are used to separate fields onto distinct cache lines and reduce contention between the writer and the reader.

Table 9.1: FIFO memory layout

Offset	Size	Content
0x0000	8 bytes	`magic` — ASCII `"VFFAFIFO"`
0x0008	2 bytes	`version` — FIFO format version (currently 0)
0x000A	6 bytes	Reserved. Must be zero.
0x0010	2 bytes	`message_size` — size of each message (in bytes)
0x0012	2 bytes	`depth` — number of message entries in the FIFO
0x0014	4 bytes	Reserved. Must be zero.
0x0018	4 bytes	`next_offset` — offset to next FIFO or zero
0x001C	36 bytes	Reserved — padding to next cache line
0x0040	2 bytes	`read_index` — updated by reader only
0x0042	2 bytes	Reserved. Must be zero.
0x0044	60 bytes	Reserved — padding to next cache line
0x0080	2 bytes	`write_index` — updated by writer only
0x0082	2 bytes	Reserved. Must be zero.
0x0084	60 bytes	Reserved — padding to message array
0x00C0	...	Message entries

The total size of the FIFO header is 192 bytes. All padding and reserved areas must be zero-initialized and ignored by both the writer and the reader.

If multiple FIFOs are defined within the same memory region, the `next_offset` field may indicate the byte offset (from the base of the current FIFO) to the next FIFO header. A value of zero indicates that no additional FIFO is present.

9.1.2 Message Entry Format

Each message entry contains a fixed-size opaque payload. The structure and semantics of each message are defined by the main body of this specification.

The total size of the message array is `message_size * depth` bytes.

The message array begins at offset `0x00C0`.

9.1.3 Index Management and Access Rules

Each FIFO instance supports exactly one writer and one reader. The following rules apply:

- The **writer** must:
  - Write message content to `fifo_msg[write_index]`.
  - Issue a store memory barrier to ensure payload visibility.
  - Update `write_index` after the message is fully committed.
- The **reader** must:
  - Load `read_index` and access `fifo_msg[read_index]`.
  - Issue a load memory barrier before reading the message.
  - Update `read_index` after the message is consumed.

Each endpoint must modify only its respective index. Index updates must be performed as 16-bit stores and wrap around modulo `depth`.

9.1.4 FIFO State and Capacity

The FIFO is treated as a circular buffer. Its state is interpreted as:

- **Empty** if `read_index == write_index`
- **Full** if `(write_index + 1)% depth == read_index`

In the full condition, one slot is reserved to disambiguate the empty and full states. At most `depth - 1` messages may be present at any time.

All index arithmetic must be performed using unsigned 16-bit values.

9.1.5 Memory Ordering Requirements

The following ordering guarantees must be upheld:

- The writer must ensure all message payload writes complete **before** updating `write_index`.
- The reader must observe a valid `write_index` **before** reading a message.
- The reader must update `read_index` **only after** the message has been fully consumed.
- The writer may read `read_index` to compute free space but must not modify it.

Store-release and load-acquire semantics must be used to enforce these rules. The exact memory barrier instructions depend on the target platform and programming environment.

9.1.6 Initialization and Validation

The writer must initialize the FIFO before use:

- Set `magic` to the ASCII string `"VFFAFIFO"` (`0x4F46414646465656` in little-endian).
- Set `version` to 0.
- Set `message_size` and `depth` to appropriate values.
- Set `read_index` and `write_index` to zero.
- Zero all reserved fields and message entries.

The reader must validate the header before use. If any of the following checks fail, the FIFO must be rejected:

- `magic` does not match the expected signature.

- `version` is not supported.
- `message_size` or `depth` are zero or exceed implementation-defined limits.

Glossary

Area Identifier

A 16-bit value assigned by the driver endpoint that identifies a shared memory region; used as part of the Bus Address format.

Bus Address

A 64-bit value composed of a 16-bit Area Identifier and a 48-bit offset that locates a specific memory position within a shared memory area.

Correlation Tuple

The pair (dev_num, msg_uid) that uniquely identifies a request/response association.

Device Endpoint

An FF-A endpoint that implements one or more Virtio devices, responds to driver-initiated messages, and manages access to shared memory regions.

Device ID

A class identifier defined by the Virtio specification indicating the type of device (e.g., network, block, console).

Device Number

A 16-bit identifier (dev_num field) in the message header addressing a Virtio device instance; 0 denotes a bus-level message.

Direct Message

An FF-A synchronous register-based transfer mechanism between two endpoints.

Driver Endpoint

An FF-A endpoint that implements one or more Virtio drivers, initiates discovery, and manages device lifecycle operations.

Event Delivery Method

A device-to-driver event message delivery pattern.

Event Message

A transport or bus message that does not expect a response.

FF-A

Firmware Framework for Arm A-Profile Architecture defining interfaces for message-based communication between isolated execution environments.

FF-A Bus Version

The negotiated Major.Minor version of this binding determining available feature set and behaviors.

FF-A Driver

A software component implementing FF-A messaging and memory sharing ABIs and offering services to virtio-msg FF-A bus.

FF-A Endpoint

A communication entity identified by a 16-bit endpoint ID that participates in FF-A messaging and memory sharing.

FF-A Partition

An isolated software context corresponding to an FF-A endpoint, assigned a 16-bit endpoint ID.

FF-A Partition Manager

A system component (e.g., hypervisor or SPMC) that enforces isolation between FF-A endpoints and provides communication services.

Indirect Message

An FF-A transfer mechanism using shared RX/TX buffers for asynchronous communication between endpoints.

Message Operation

The msg_op field (8 bits) identifying the operation or message type.

Message Transfer Method

A transport mechanism that carries any transport or bus message (Normal or Event).

Message Unique Identifier

The msg_uid field (16 bits) assigned by a requester for each solicited response.

Normal Message

A Virtio or bus message that expects a response.

Notification ID

A 16-bit identifier bound via FF-A notification interfaces and used to signal message or event availability between endpoints.

Shared Memory Area

A memory region shared between two FF-A endpoints via FF-A memory operations, identified by a 16-bit Area Identifier for DMA-style exchange.

Transport Revision

The virtio-msg transport revision negotiated alongside the FF-A Bus Version.

UUID

A standardized 128-bit identifier used for FF-A protocol services including Virtio-msg Bus Driver and Device endpoint protocols.

Virtio

A standard interface for virtualized devices defining device classes, drivers, and multiple transport mechanisms (PCI, MMIO, virtio-msg, etc.).

Virtio Device

A software component implementing a Virtio device class and processing virtqueue requests from a Virtio driver.

Virtio Driver

A software component implementing driver-side logic for a Virtio device class, managing configuration, virtqueues, and I/O.

Virtio Message Bus over FF-A

The FF-A binding of the Virtio Message Bus mapping message delivery, discovery, memory sharing, and lifecycle operations onto FF-A primitives.

Virtio Over Message Transport

The transport defined by the Virtio specification that replaces register access with structured messages.

Virtio over Messages

A Virtio transport using structured messages instead of MMIO or PCI accesses for driver–device communication.

VM

Virtual Machine.