IP Packet Accelerator Driver API

Interface Specification

80-NC254-45 A

March 19, 2013

Confidential and Proprietary – Qualcomm Technologies, Inc.

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Contents

Figures

Tables

Revision History

Revision	Date	Description
A	Mar 2013	Initial release.

1 Introduction

1.1 Purpose

This document captures the API exposed by the IP Packet Accelerator (IPA) driver to client peripheral drivers in kernel space and user space controller applications. This document is intended for distribution to engineering teams involved in the development and integration of software that uses IPA hardware services.

1.2 Scope

This document is an API reference. It does not capture software design of the IPA driver. The reader is expected to be familiar with the IPA system architecture and hardware capabilities.

1.3 Conventions

Function declarations, function names, type declarations, and code samples appear in a different font, e.g., `#include`.

Parameter types are indicated by arrows:

- → Designates an input parameter
- ← Designates an output parameter
- ↔ Designates a parameter used for both input and output

1.4 References

Reference documents are listed in Table 1-1. Reference documents that are no longer applicable are deleted from this table; therefore, reference numbers may not be sequential.

Table 1-1 Reference documents and standards

Ref.	Document	
Qualcomm Technologies		
Q1	*Application Note: Software Glossary for Customers*	CL93-V3077-1

1.5 Technical Assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies at https://support.cdmatech.com/.

If you do not have access to the CDMATech Support website, register for access or send email to support.cdmatech@qti.qualcomm.com.

1.6 Acronyms

For definitions of terms and abbreviations, refer to [Q1]. Table 1-2 lists terms that are specific to this document.

Table 1-2 Acronyms

Acronym	Definition
BAM	bus access manager
DMA	direct memory access
HLOS	high level operating system
IPA	IP packet accelerator
MBIM	mobile broadband interface model
NAT	network address translation
SPI	security parameter index
SPS	smart peripheral subsystem
TLP	thin layer protocol

2 IPA Driver Services

This chapter provides an overview of the different services provided by the IPA driver.

2.1 IPA Architecture Overview

IPA is a hardware block that implements data packet protocol processing functions. IPA serves to offload protocol processing for most common cases of tethering and Mobile router connectivity scenarios from the software drivers and HLOS, thereby freeing up the CPU. The offloaded processing includes:

- Aggregation/deaggregation protocols such as MBIM and TLP[1]
- IP routing to move packets between multiple endpoints
- Link layer header removal and insertion functionality to deal with differing protocols on various links
- Network address translation (NAT) for sharing public IP address with multiple LAN hosts

Figure 2-1 illustrates a conceptual diagram of IPA.

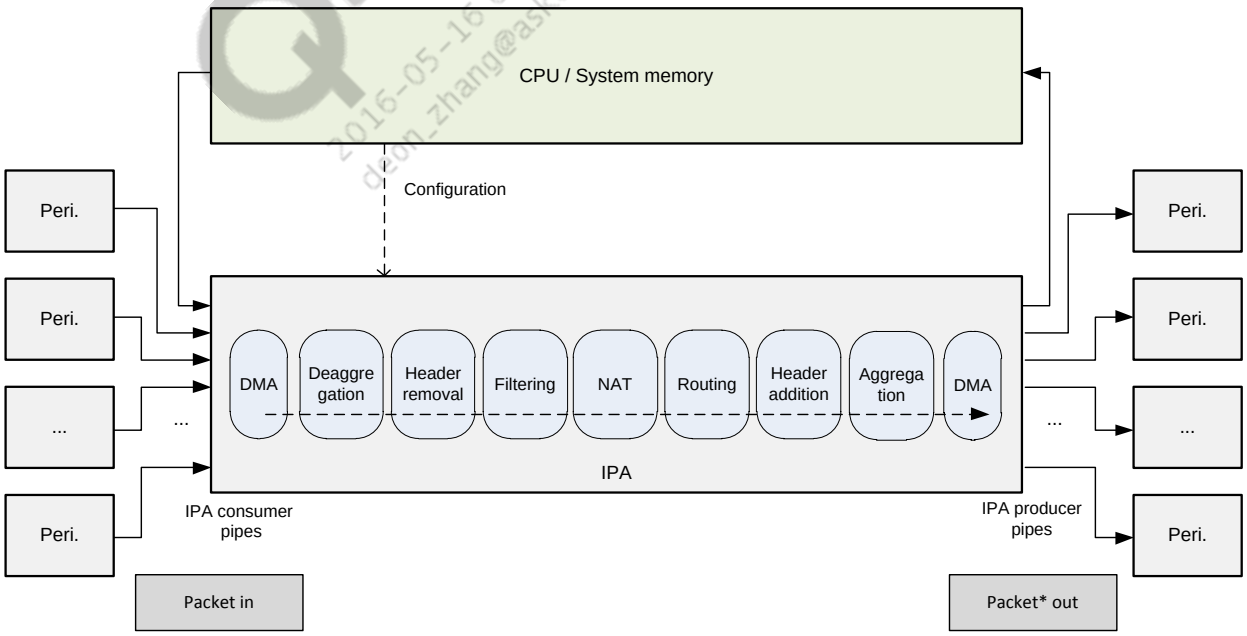

Figure 2-1 IPA conceptual diagram

In IPA-based data path architecture, each peripheral endpoint used for data plane connectivity is connected to a corresponding IPA endpoint in BAM-to-BAM mode. Packets received from a

[1] TLP is a Qualcomm defined aggregation protocol. MBIM is a Microsoft defined protocol based on NCM standard.

peripheral endpoint go through any needed protocol processing in IPA, based on rules configured by software. Routing rules configured by software are then used to determine the destination endpoint of each packet. The destination may be another peripheral endpoint.

IPA also supports data plane connectivity with the CPU through System-to-BAM mode connections. IPA can send or receive packets to and from software using these endpoints, just as with peripheral endpoints connected to it. This enables data connectivity to applications running on the CPU via peripheral endpoints connected to IPA. This software data path is also used for "exception" packets that cannot be processed by IPA and need to be handled by the driver software or HLOS.

2.2 Theory of Operation

2.2.1 Overview

IPA is configurable through a register interface and through BAM immediate commands that the software can send over the System-to-BAM endpoints. The IPA driver running on the application CPU is responsible for configuration of the IPA hardware.

The IPA driver provides an interface to configure the IPA hardware for data plane connectivity with a peripheral. Client peripheral drivers use the services provided by the IPA driver to allocate and configure an IPA endpoint in BAM-to-BAM mode with a peripheral endpoint. The client driver can then receive and transmit data packets to and from the peripheral via IPA hardware. The use of IPA hardware as a conduit for data plane connectivity for a given peripheral endpoint enables the use of aggregation/deaggregation protocol implementation in the IPA hardware; this enables offloading of this function from the host CPU. The IPA driver uses the services of the Smart Peripheral Subsystem (SPS) driver to use DMA services provided by the IPA BAM for data movement between system memory and IPA hardware. Figure 2-2 illustrates the peripheral-to-IPA connectivity.

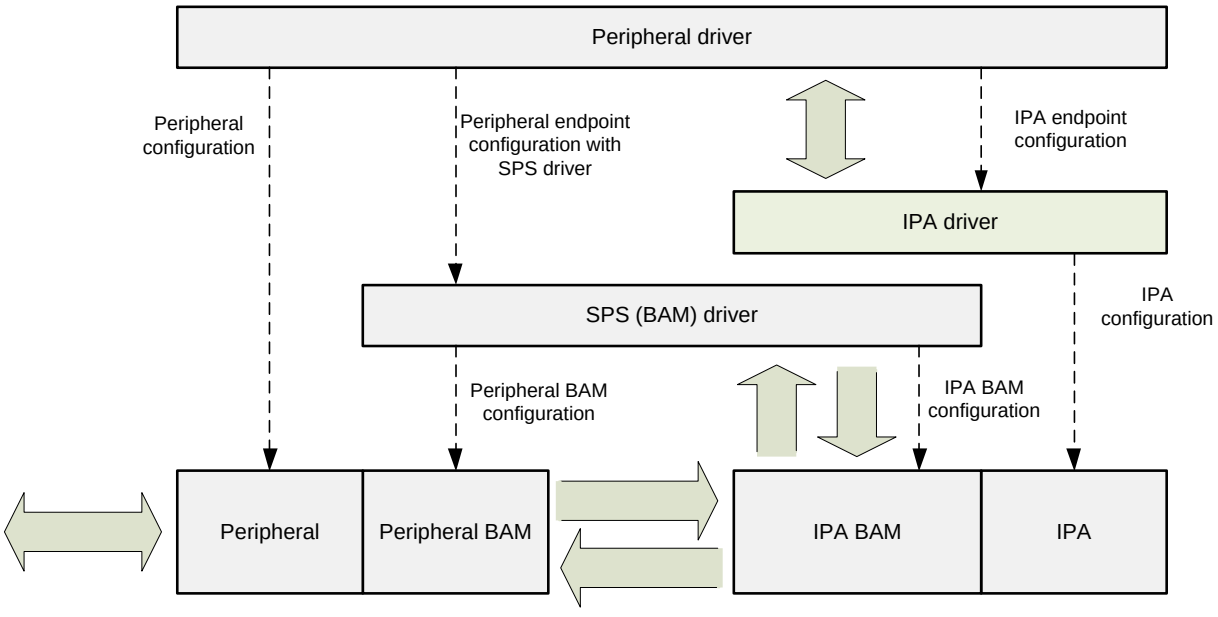

Figure 2-2 Software architecture for peripheral-to-IPA connectivity

Note that the IPA driver only provides API functions for configuring IPA hardware (including the IPA BAM hardware). A peripheral-to-IPA connection also requires the peripheral side of the pipe to be set up and this requires the peripheral driver to directly use the SPS driver for setting up that end of the connection.

Client drivers and/or control applications in the user space can also request additional services to enable direct communication between two peripherals via IPA hardware. The IPA driver enables establishing peripheral-to-peripheral data plane connectivity in BAM-to-BAM mode through the configuration of appropriate filtering and routing rules, thereby offloading the data plane processing from the host CPU. If certain "exception" packets need to be processed by software, it can be done by appropriate configuration of the filtering rules. The IPA driver provides a flexible API that enables arbitrary configuration of rules by client drivers/user space as needed for different use cases. It is up to the client drivers/user space to determine the appropriate routing and filtering configuration needed in the hardware. The IPA driver provides an abstract API that hides the hardware-specific implementation details from the clients. Figure 2-3 illustrates the peripheral-to-peripheral connectivity via IPA.

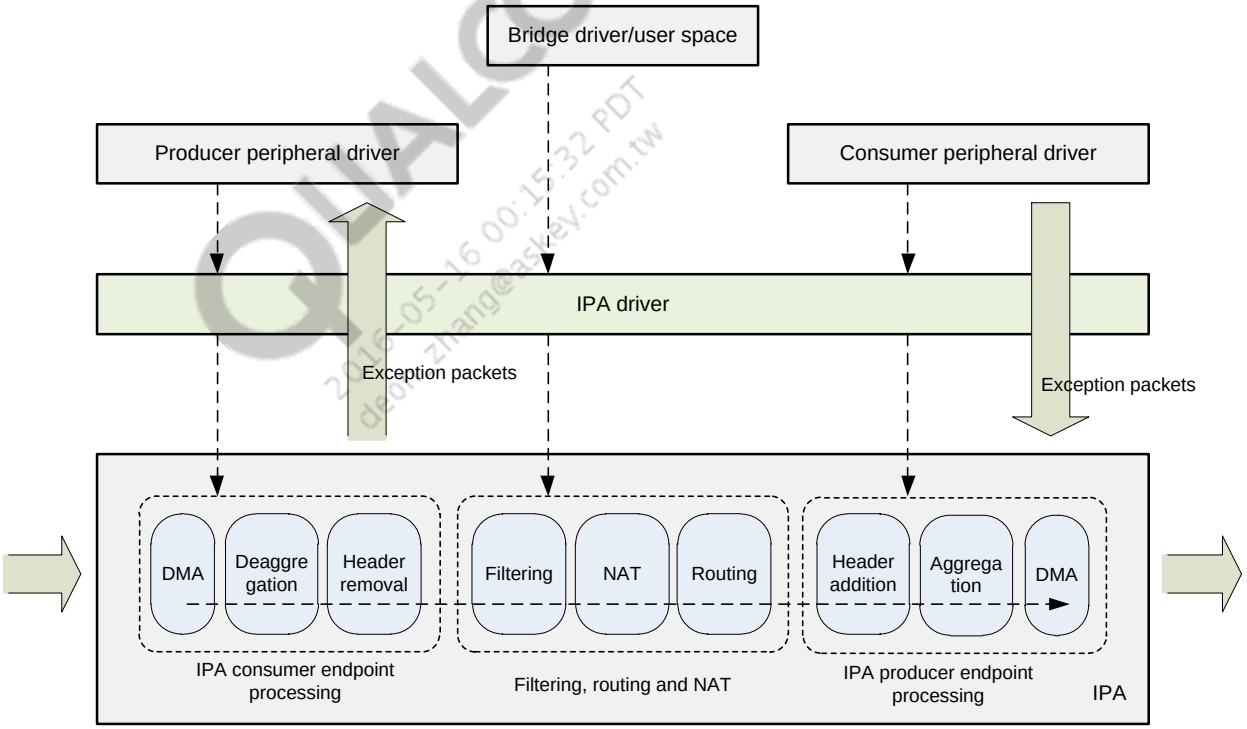

Figure 2-3 Software architecture for peripheral-to-peripheral connectivity via IPA

2.2.2 Driver Services

Figure 2-4 illustrates the different services provided by the IPA driver and the intended clients of those services. The IPA driver API provides both kernel space and user space functions to enable clients from either domain to use IPA services. There are some limitations on the services available from each domain.

The client peripheral drivers use the following *core* services:

- **Endpoint configuration service** to allocate and configure IPA endpoint for data plane connectivity between IPA and peripheral
- **Header configuration service** to optionally configure link layer headers to attach to outgoing packets
- **Filtering rule configuration service** to optionally configure filter rules on endpoints owned by the client driver

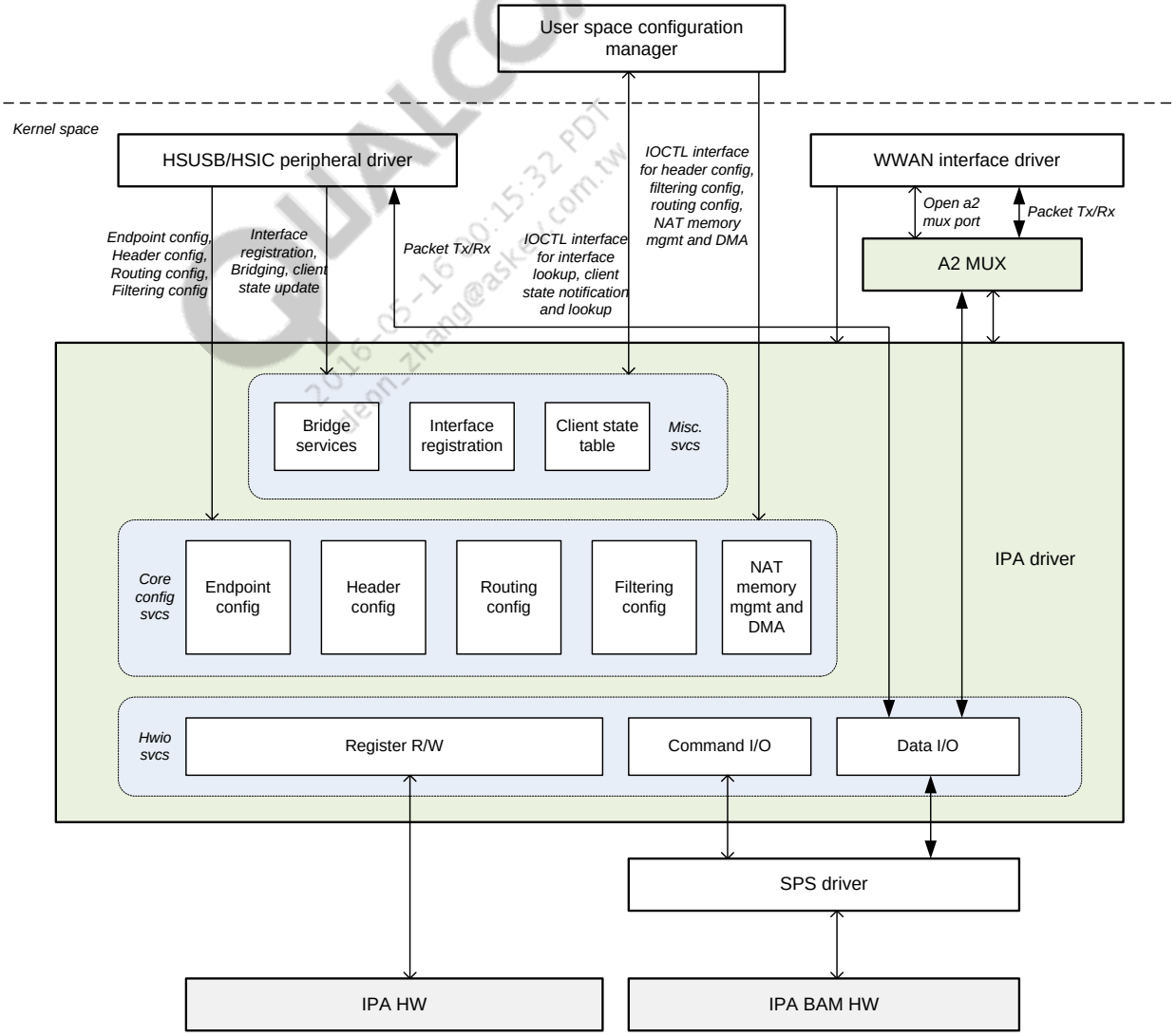

Figure 2-4 IPA driver services

In addition, the IPA driver provides certain *extended* services. Some extended services build upon the core services of the driver to provide a higher level and easier to use interface to clients. Some extended services are not related to the IPA hardware functions themselves, but are provided to facilitate information sharing among the different clients of the driver, thereby enabling the coordination needed in software to manage the complex configuration of the hardware for various use cases.

The following extended services are used by the client drivers:

- **Interface registration service** to optionally register interface (logical endpoint) to (physical) endpoint mapping information with the IPA driver
- **Bridge service** to optionally set up data plane bridging between two logical endpoints, enabling peripheral-to-peripheral data connectivity
- **Client state update service** to optionally request enabling/disabling of IPA-based acceleration for specific clients

The user space configuration manager uses the following core services:

- **Filtering rule configuration service** to configure filter rules on various endpoints, if needed, for peripheral-to-peripheral data connectivity
- **Routing rule configuration service** to configure routing tables and add/delete routing rules to those tables, if needed, for peripheral-to-peripheral data connectivity
- **Header configuration service** to optionally configure link layer headers to attach to outgoing packets
- **NAT memory management and DMA service** to manage NAT configuration in user space

The user space also has access to the following extended services:

- **Interface lookup service** to look up interface (logical endpoint) to (physical) endpoint mapping information
- **Client state lookup service** to determine whether IPA-based acceleration is enabled for specific clients

Finally, IPA software also includes an implementation of A2 MUX. **A2 MUX service** provides multiplexing and demultiplexing to use the shared A2 endpoint for multiple WWAN calls/interfaces.

2.2.3 Endpoint Configuration

For enabling data plane connectivity between a peripheral endpoint and IPA, e.g., with IPA in consumer mode, the following events must occur:

1. A corresponding IPA endpoint (pipe) must be allocated to be connected with the peripheral endpoint.
2. IPA hardware must be configured with the endpoint configuration for the IPA consumer pipe, including:
    a. NAT configuration: whether packets arriving on that endpoint go through NAT, and if so, whether source NAT or destination NAT is performed

b. Header configuration: if packets received on that endpoint carry a link layer header, then the parameters of that header need to be specified, so that IPA hardware can figure out the start of the IP packet and, if needed, strip the link layer header before delivering to another peripheral

c. Mode configuration: whether packets arriving on the endpoint go through full-blown routing or are directly dispatched to a specific producer endpoint

d. Aggregation configuration: if an aggregation protocol is used for the link, then the protocol must be identified

e. Route configuration: The default route table to use for routing packets coming in on this endpoint must be specified. This is normally “a5lan” or “a5wan”, which are tables preconfigured by the IPA driver.

3. The IPA consumer endpoint must be configured in BAM-to-BAM mode with the peripheral producer endpoint using the services of the SPS driver.

The steps are similar for an IPA producer-to-peripheral consumer connection. The endpoint configuration service provides the client peripheral driver the means to accomplish all configuration.

The client peripheral driver is expected to perform endpoint configuration when the peripheral is connected, and deconfigure or reset the endpoint configuration when the peripheral is disconnected. The endpoint configuration functions require the peripheral driver to first set up the IPA side of the BAM-to-BAM connection using the provided functions, for both consumer and producer endpoints. For an IPA consumer endpoint, the peripheral driver ensures that the peripheral BAM endpoint is disconnected before the IPA endpoint is disconnected using the provided functions.

After a peripheral endpoint is connected, the peripheral driver can transmit packets to that endpoint via IPA using the packet transmit functions. As part of endpoint configuration, the peripheral driver can register a notification callback for packets received from the peripheral endpoint by IPA (which need to be handled in software) and for notification of packet transmission completion.

2.2.4 Header Configuration

When configuring an IPA producer endpoint, it may be necessary to also configure a header to be attached to IP packets that are routed by IPA to that endpoint. For example, a USB ECM endpoint expects IP packets encapsulated in Ethernet headers. The ECM function driver configures an Ethernet header that is attached to packets that the IPA routes to the USB ECM endpoint.

The header configuration functions let the client configure headers in IPA hardware. Each header must be assigned a unique name by the client. To tell the IPA hardware to apply the header, a routing rule must be configured specifying the header handle to use when routing packets to that endpoint based on this rule. The same header can be applied for multiple rules. The configuration of the routing rules are typically done by user space/bridge driver. The interface registration and lookup service allows for discovery of applicable headers for network interfaces/logical endpoints. To avoid race conditions between the configuration of headers and their use by other clients, it is recommended that the client driver configure the headers before the corresponding network interface is brought up. This lets clients serialize the routing rule configuration with header configuration, using the state of the network interface as a synchronization event.

Since routing rules use headers and rules, and headers are configured and managed by different clients, the IPA driver implements a reference counting mechanism to prevent inconsistent configuration of hardware. Each header object maintains a reference count, which is incremented when a routing rule is added that refers to the header. As a result, header deletion by the client driver may not result in immediate removal of the header from the hardware. The header deletion is committed to hardware after all rules pointing to it are removed. Similarly, addition of a header may overwrite the existing header with the same name.

In certain situations, a client peripheral driver may not have complete information to build the header in its entirety when the corresponding network interface is brought up. For example, building an Ethernet/802.3 header requires the knowledge of the MAC address of the client PC, which is not known when the WLAN interface comes up. In this situation, the client driver is expected to configure a partial header with all other fields except the MAC address of the destination client PC. The user space client is then expected to use the partial header to build the complete header for the client PC.

Note that IPA hardware attaches the configured header only to those packets that are received over a peripheral endpoint and routed by IPA to the destination peripheral endpoint, based on configured routing rules in hardware. The peripheral client driver is still expected to attach the appropriate link layer header to the IP packets it submits to the IPA driver for transmission. Any packets submitted by software to the IPA driver do not go through header addition in IPA hardware. The attached header must not include the aggregation protocol header (e.g., RNDIS) as aggregation is always performed in hardware.

2.2.5 Routing Rule Configuration

The routing rule configuration service allows configuration of route tables in IPA hardware. The IPA driver supports configuration of multiple route tables, each containing one or more routing rules. Each routing rule supports packet matching based on a combination of the following parameters, which together comprise a filter rule attribute (also used in filter rule definition):

- For IPv4 packets
  - Next protocol number
  - TOS field
  - Source IP address and mask
  - Destination IP address and mask
- For IPv6 packets
  - Next header
  - Flow label
  - Traffic class
  - Source IPv6 address and mask
  - Destination IPv6 address and mask
- For TCP/UDP packets (provided next protocol/next header is set to TCP/UDP)
  - Source port or source port range
  - Destination port or destination port range

- For ICMP/ICMPv6/IGMP/MLD packets (provided next protocol/next header is set to one of these)
  - Type
  - Code
- For IPSec packets (provided next protocol/next header is set to IPSec)
  - Security Parameter Index (SPI)
- Metadata value and mask

The routing rule includes, in addition the filter rule attributes, the destination endpoint and an optional header handle. The header handle corresponds to the header attached to a packet routed to the destination endpoint on a routing rule match. Note that routing rules are executed in order and the first matching rule is applied. Separate tables need to be configured for IPv4 and IPv6. IPA hardware cannot route non-IPv4/IPv6 packets; any non-IPv4/IPv6 packets are always handled as exception packets and handled by software. The IPA driver delivers all such exception packets to the client driver.

It is possible to define a *default* routing/filtering rule by not specifying any attribute value in the filter rule attribute. Such a rule matches all packets.

Routing table objects also use a reference counting mechanism to guarantee consistency of the configuration when multiple clients add or delete rules to the same table and add filtering rules using the table. A table object is automatically created when rules are added, and cleaned up when it becomes empty. The table object reference count is also incremented when a filtering rule is added that references the routing table. The routing table object is not deleted until all filtering rules referring to the routing table are deleted.

Clients are expected to create routing tables with descriptive and well-known names. How clients assign and share names is out of the scope of this document.

2.2.6 Filtering Rule Configuration

The filtering rule configuration service allows configuration of filter tables in IPA hardware. Each filter table is associated with a specific IPA consumer endpoint. As with routing tables, there are separate filter tables for IPv4 and IPv6. Filtering rules are expected to be installed by a user space/bridge driver for enabling peripheral-to-peripheral data connectivity (optionally including NAT processing before sending packets to the destination peripheral/endpoint). Filtering allows identification of the packets that are routed directly to another peripheral, and also identifies any exception packets that need to be handled in software. If no filtering rules are installed on an endpoint, all packets are routed by IPA to the software driver.

Each filter rule specifies the following:

- Filter rule attribute (same as used for routing rules) defining the filter specification
- Action to take when the filter rule matches, which can be one of the following:
  - Bypass NAT (and proceed directly to routing)
  - Source NAT processing (followed by routing)
  - Destination NAT processing (followed by routing)
  - Exception, i.e., pass the packet to software for handling as an exception packet

- Routing table handle to use for routing the packet

As explained earlier, a reference counting mechanism is used to ensure that a routing table with valid filtering rules referring to it is not deleted until all filter rule references are deleted as well.

2.2.7 NAT Memory Management and DMA

Configuration of NAT rules in IPA hardware is managed by a user space NAT driver. The kernel driver enables implementation of NAT table management in user space by providing memory management service for NAT tables. The kernel driver provides an interface to user space for requesting allocation of device/system memory for NAT tables and configuration of an IPA hardware register with the table addresses. The building of the table data structures in memory is managed from user space. The IPA (kernel space) driver also provides a service to issue NAT DMA commands to hardware to enable or disable specific rules in the NAT tables as required by the hardware.

2.2.8 Data Transfer

As part of endpoint configuration, a peripheral driver can register a notification callback for packets received from the peripheral endpoint by the IPA (which need to be handled in software) and for notification of packet transmission completion. After a peripheral endpoint is connected, the peripheral driver can transmit packets to that endpoint via IPA using the packet transmit functions. After packet transmission is complete, notification is provided to the client using a registered completion callback. The client is expected to free the buffers on receiving the notification of completion. Any received packets are forwarded to the client using the registered Rx callback.

The same Tx/Rx mechanism is used for all packets, including packets that need to be consumed by software on the ARM® Cortex™-A5 processor as well as exception packets that need to be handled in software and then resubmitted to IPA. In most scenarios, packets that need to be transmitted from the Cortex-A5 processor software to a peripheral/modem are directed by software to be sent to a specific destination IPA producer endpoint. The transmit functions require the client to specify the destination endpoint explicitly for each packet. The only known exception to this rule is the WLAN A-MPDU use case, which requires the WLAN driver to submit reordered packets back to the IPA for full processing including routing. The same transmit functions support this special use case; in this case, the client specifies the IPA consumer endpoint as the destination in the function, and the IPA driver submits the packet to IPA for full processing on that consumer endpoint.

2.2.9 Interface Registration and Lookup

Client peripheral drivers are expected to be aware of the link-layer header formats and configure IPA driver with the headers. The user space/bridge driver clients need to know the applicable header format to use for a given endpoint so that they can configure the routing/filtering rules appropriately. In certain cases, some additional information must be known about the mapping of the logical data stream to the physical endpoints, since a given physical endpoint can carry multiplexed traffic belonging to multiple logical data streams. This multiplexing information is typically known to the client peripheral driver. The interface registration and lookup service provides a way for user space/bridge drivers to dynamically discover the logical-to-physical endpoint mapping information and the applicable header formats to use for a destination physical endpoint.

The IPA driver provides functions to client peripheral drivers to register logical data streams with the IPA driver. A logical data stream is identified through a *interface name*. The driver must use a well-known name for the logical data stream, ideally using the name of the corresponding network interface. The use of this well-known name allows the user space to locate the mapping information (*properties*) identifying the logical data stream. Each logical data stream consists of multiple physical streams of data, possibly received over multiple physical endpoints. The client driver provides the mapping of the logical data stream to the various physical data substreams comprising the logical stream. The mapping properties consist of:

- Interface name identifying the logical data stream
- One or more of the following sets of information, one per physical data substream
  - Source/destination (physical) endpoint
  - Filter rule attribute that can be used to filter out the substream from the traffic received over the source physical endpoint, or filter out the substream intended for transmission over the destination physical endpoint.
  - Header handle of the header to be applied to outgoing packets, for a destination endpoint

Figure 2-5 illustrates the mapping. Note that in the degenerate case where a single pair of source and destination endpoints exclusively carries traffic for a given network interface, the filter rule attribute can be a default rule. Also, the header handle can refer to a partial header.

As a concrete example, WLAN uses a single Rx endpoint to receive traffic for multiple logical access points, each represented through a separate network interface. Also, in AP+STA mode, traffic received from the AP may also be received over the same Rx endpoint. The WLAN driver provides the metadata-based filter rule during interface registration to identify the subset of the traffic received over the endpoint that corresponds to that interface.

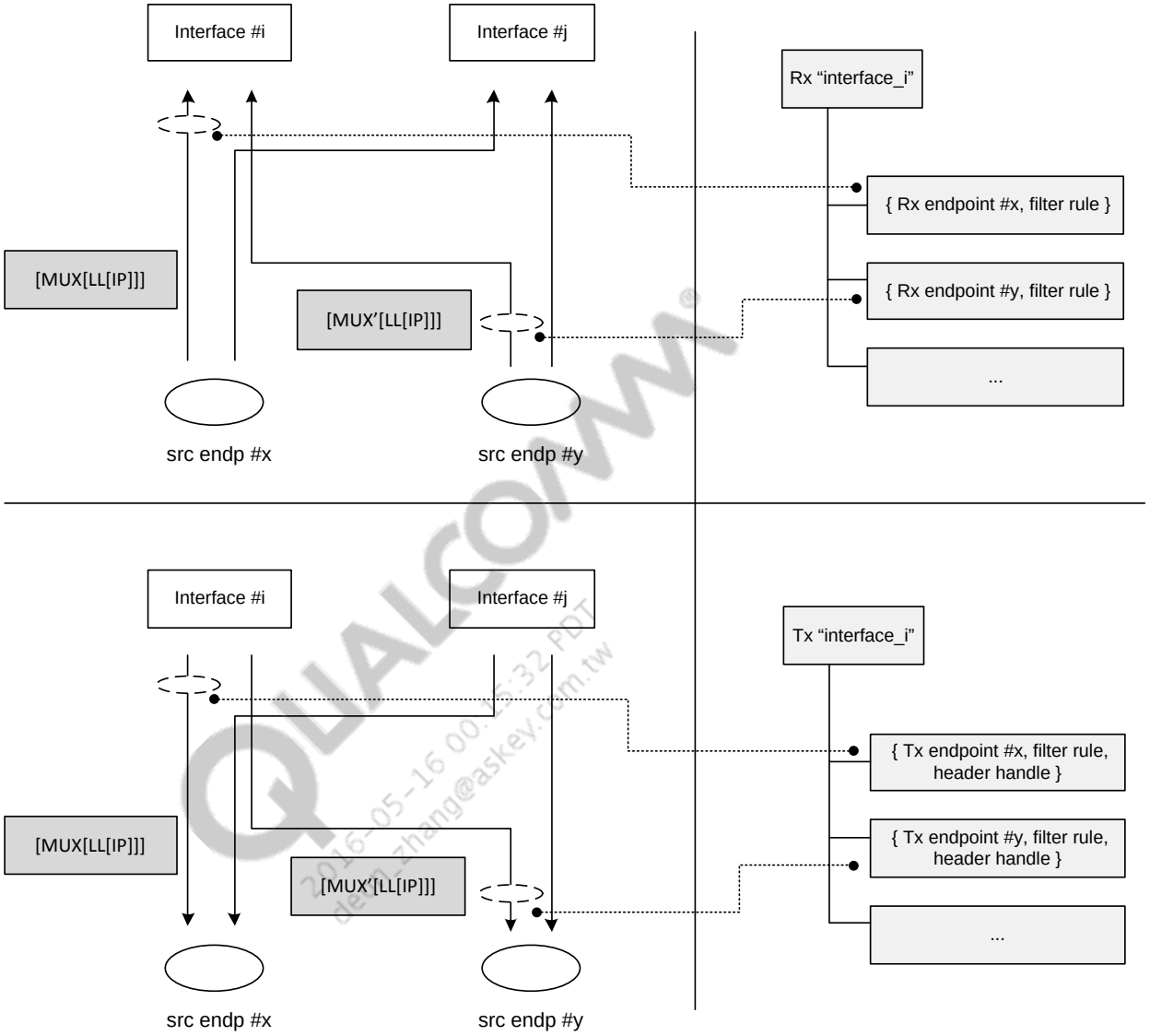

Figure 2-5 Mapping logical to physical data streams using interface properties

Note that regardless of whether multiplexing is used, the client driver must perform interface registration to allow the user space/bridge driver to discover the parameters needed to enable peripheral-to-peripheral routing of data in IPA hardware. This registration must be performed before the network interface is brought up to avoid race conditions between interface registration by the peripheral driver and lookup by other clients.

2.2.10 Generic Notification Mechanism

The IPA driver provides a generic notification/messaging mechanism to enable communication between kernel space clients of the driver and user space clients. This enables user space clients to receive notifications from kernel space. The following use cases use this mechanism.

2.2.10.1 WLAN Client Management

The WLAN use case requires knowledge of the active WLAN clients (MAC addresses) so that the WLAN header configuration and routing rule configuration for each client can be performed by software. Using messages defined for this purpose, WLAN client driver can notify user space of active WLAN clients. User space can then pull the message and perform the required routing/header configuration for active clients.

The WLAN use case also introduces the requirement to buffer packets in the WLAN driver while a WLAN station is in power save. This requires disabling direct hardware routing of packets to WLAN transmit endpoints for the clients in power save. The WLAN driver can use the notification mechanism to notify the user space when there is a change in the power save state of a client. The user space is then expected to reconfigure routing/filtering rules to avoid hardware packet routing for clients in power save.

2.2.10.2 IPA Hardware Routing Bypass

In certain use cases it may be necessary to disable the hardware data path completely. As an example, when operating in concurrent AP and STA mode with multi-channel concurrency, all outgoing WLAN traffic must be buffered by the host WLAN driver. The WLAN driver can use the notification mechanism to notify the user space when this is necessary and the user space is then expected to change routing/filtering configuration accordingly.

3 API Reference

3.1 Common Types and Definitions

This section describes the definitions of common data types used in the API functions.

3.1.1 Data Types

3.1.1.1 enum ipa_client_type

Enumeration of client endpoints that can connect with IPA.

Definition

```
enum ipa_client_type {
    IPA_CLIENT_PROD,
    IPA_CLIENT_HSIC1_PROD = IPA_CLIENT_PROD,
    IPA_CLIENT_HSIC2_PROD,
    IPA_CLIENT_HSIC3_PROD,
    IPA_CLIENT_HSIC4_PROD,
    IPA_CLIENT_HSIC5_PROD,
    IPA_CLIENT_USB_PROD,
    IPA_CLIENT_A5_WLAN_AMPDU_PROD,
    IPA_CLIENT_A2_EMBEDDED_PROD,
    IPA_CLIENT_A2_TETHERED_PROD,
    IPA_CLIENT_A5_LAN_WAN_PROD,
    IPA_CLIENT_A5_CMD_PROD,
    IPA_CLIENT_Q6_LAN_PROD,

    IPA_CLIENT_CONS,
    IPA_CLIENT_HSIC1_CONS = IPA_CLIENT_CONS,
    IPA_CLIENT_HSIC2_CONS,
    IPA_CLIENT_HSIC3_CONS,
    IPA_CLIENT_HSIC4_CONS,
    IPA_CLIENT_HSIC5_CONS,
    IPA_CLIENT_USB_CONS,
    IPA_CLIENT_A2_EMBEDDED_CONS,
    IPA_CLIENT_A2_TETHERED_CONS,
```

```
        IPA_CLIENT_A5_LAN_WAN_CONS,
        IPA_CLIENT_Q6_LAN_CONS,

        IPA_CLIENT_MAX,
    };
```

3.1.1.2 enum ipa_ip_type

Enumeration of IP protocol types.

Definition

```
enum ipa_ip_type {
   IPA_IP_v4,
   IPA_IP_v6,
   IPA_IP_MAX
};
```

3.1.1.3 struct ipa_rule_attrib

Struct type defining attributes of a routing/filtering rule. All parameters are little-endian.

Definition

```
/**
 * struct ipa_rule_attrib - attributes of a routing/filtering
 * rule, all in LE
 * @attrib_mask: what attributes are valid
 * @src_port_lo: low port of src port range
 * @src_port_hi: high port of src port range
 * @dst_port_lo: low port of dst port range
 * @dst_port_hi: high port of dst port range
 * @type: ICMP/IGMP type
 * @code: ICMP/IGMP code
 * @spi: IPSec SPI
 * @src_port: exact src port
 * @dst_port: exact dst port
 * @meta_data: metadata val
 * @meta_data_mask: metadata mask
 * @u.v4.tos: type of service
 * @u.v4.protocol: protocol
 * @u.v4.src_addr: src address value
 * @u.v4.src_addr_mask: src address mask
 * @u.v4.dst_addr: dst address value
 * @u.v4.dst_addr_mask: dst address mask
 * @u.v6.tc: traffic class
 * @u.v6.flow_label: flow label
 * @u.v6.next_hdr: next header
```

```
         * @u.v6.src_addr: src address val
         * @u.v6.src_addr_mask: src address mask
         * @u.v6.dst_addr: dst address val
         * @u.v6.dst_addr_mask: dst address mask
         */
        struct ipa_rule_attrib {
            uint32_t attrib_mask;
            uint16_t src_port_lo;
            uint16_t src_port_hi;
            uint16_t dst_port_lo;
            uint16_t dst_port_hi;
            uint8_t type;
            uint8_t code;
            uint32_t spi;
            uint16_t src_port;
            uint16_t dst_port;
            uint32_t meta_data;
            uint32_t meta_data_mask;
            union {
                struct {
                    uint8_t tos;
                    uint8_t protocol;
                    uint32_t src_addr;
                    uint32_t src_addr_mask;
                    uint32_t dst_addr;
                    uint32_t dst_addr_mask;
                } v4;
                struct {
                    uint8_t tc;
                    uint32_t flow_label;
                    uint8_t next_hdr;
                    uint32_t src_addr[4];
                    uint32_t src_addr_mask[4];
                    uint32_t dst_addr[4];
                    uint32_t dst_addr_mask[4];
                } v6;
            } u;
        };
```

3.2 Endpoint Configuration

This section describes endpoint configuration data types and functions.

3.2.1 Data Types

3.2.1.1 enum ipa_nat_en_type

Enumeration of supported NAT configurations.

Definition

```
/**
 * enum ipa_nat_en_type - NAT setting type in IPA end-point
 */
enum ipa_nat_en_type {
    IPA_BYPASS_NAT,
    IPA_SRC_NAT,
    IPA_DST_NAT,
};
```

3.2.1.2 enum ipa_mode_type

Enumeration of supported modes for an endpoint.

Definition

```
/**
 * enum ipa_mode_type - mode setting type in IPA end-point
 * @BASIC: basic mode
 * @ENABLE_FRAMING_HDLC: not currently supported
 * @ENABLE_DEFRAMING_HDLC: not currently supported
 */
enum ipa_mode_type {
    IPA_BASIC,
    IPA_ENABLE_FRAMING_HDLC,
    IPA_ENABLE_DEFRAMING_HDLC,
    IPA_DMA,
};
```

3.2.1.3 enum ipa_aggr_en_type

Enumeration of supported aggregation modes for an endpoint.

Definition

```
enum ipa_aggr_en_type {
  IPA_BYPASS_AGGR,
  IPA_ENABLE_AGGR,
  IPA_ENABLE_DEAGGR,
};
```

3.2.1.4 enum ipa_aggr_type

Enumeration of supported aggregation protocols. Note that only MBIM_16 and TLP are supported.

Definition

```
/**
 *  enum ipa_aggr_type - type of aggregation in IPA end-point
 */
enum ipa_aggr_type {
    IPA_MBIM_16,
    IPA_MBIM_32,
    IPA_TLP,
};
```

3.2.1.5 struct ipa_ep_cfg_nat

Struct type for definition of NAT configuration for an endpoint.

Definition

```
/**
 * struct ipa_ep_cfg_nat - NAT configuration in IPA end-point
 * @nat_en: This defines the default NAT mode for the pipe: in case of
 *      filter miss - the default NAT mode defines the NATing operation
 *      on the packet. Valid for Input Pipes only (IPA consumer)
 */
struct ipa_ep_cfg_nat {
    enum ipa_nat_en_type nat_en;
};
```

3.2.1.6 struct ipa_ep_cfg_hdr

Struct type for definition of header configuration for an endpoint.

Definition

```
/**
 * struct ipa_ep_cfg_hdr - header configuration in IPA end-point
 * @hdr_len:    Header length in bytes to be added/removed. Assuming header len
 *      is constant per endpoint. Valid for both Input and Output Pipes
 * @hdr_ofst_metadata_valid:    0: Metadata_Ofst  value is invalid, i.e., no
 *              metadata within header.
 *              1: Metadata_Ofst  value is valid, i.e., metadata
 *              within header is in offset Metadata_Ofst Valid
 *              for Input Pipes only (IPA Consumer) (for output
 *              pipes, metadata already set within the header)
 * @hdr_ofst_metadata:  Offset within header in which metadata resides
 *          Size of metadata - 4bytes
 *          Example -  Stream ID/SSID/mux ID.
 *          Valid for  Input Pipes only (IPA Consumer) (for output
 *          pipes, metadata already set within the header)
 * @hdr_additional_const_len:   Defines the constant length that should be added
 *              to the payload length in order for IPA to update
 *              correctly the length field within the header
 *              (valid only in case Hdr_Ofst_Pkt_Size_Valid=1)
 *              Valid for Output Pipes (IPA Producer)
 * @hdr_ofst_pkt_size_valid:    0: Hdr_Ofst_Pkt_Size  value is invalid, i.e., no
 *              length field within the inserted header
 *              1: Hdr_Ofst_Pkt_Size  value is valid, i.e., a
 *              packet length field resides within the header
 *              Valid for Output Pipes (IPA Producer)
 * @hdr_ofst_pkt_size:  Offset within header in which packet size resides. Upon
 *          Header Insertion, IPA will update this field within the
 *          header with the packet length. Assumption is that
 *          header length field size is constant and is 2Bytes
 *          Valid for Output Pipes (IPA Producer)
 * @hdr_a5_mux: Determines whether A5 Mux header should be added to the packet.
 *      This bit is valid only when Hdr_En=01(Header Insertion)
 *      software should set this bit for IPA-to-A5 pipes.
 *      0: Do not insert A5 Mux Header
 *      1: Insert A5 Mux Header
 *      Valid for Output Pipes (IPA Producer)
 */
```

```
struct ipa_ep_cfg_hdr {
    u32 hdr_len;
    u32 hdr_ofst_metadata_valid;
    u32 hdr_ofst_metadata;
    u32 hdr_additional_const_len;
    u32 hdr_ofst_pkt_size_valid;
    u32 hdr_ofst_pkt_size;
    u32 hdr_a5_mux;
};
```

3.2.1.7 struct ipa_ep_cfg_mode

Struct type for definition of mode configuration for an endpoint.

Definition

```
/**
 * struct ipa_ep_cfg_mode - mode configuration in IPA end-point
 * @mode:   Valid for Input Pipes only (IPA Consumer)
 * @dst:    This parameter specifies the output pipe to which the packets
 *      will be routed.
 *      This parameter is valid for Mode=DMA and not valid for
 *      Mode=Basic
 *      Valid for Input Pipes only (IPA Consumer)
 */
struct ipa_ep_cfg_mode {
    enum ipa_mode_type mode;
    enum ipa_client_type dst;
};
```

3.2.1.8 struct ipa_ep_cfg_aggr

Struct type for definition of aggregation configuration for an endpoint.

Definition

```
/**
 * struct ipa_ep_cfg_aggr - aggregation configuration in IPA end-point
 * @aggr_en:    Valid for both Input and Output Pipes
 * @aggr:   Valid for both Input and Output Pipes
 * @aggr_byte_limit: Limit of aggregated packet size in KB (<=32KB) When
 *          set to 0, there is no size limitation on the aggregation.
 *          When both Aggr_Byte_Limit and Aggr_Time_Limit are set
 *          to 0, there is no aggregation, every packet is sent
 *          independently according to the aggregation structure
 *          Valid for Output Pipes only (IPA Producer )
 * @aggr_time_limit:    Timer to close aggregated packet (<=32ms) When set
 *          to 0, there is no time limitation on the aggregation.  When
```

```
 *          both Aggr_Byte_Limit and Aggr_Time_Limit are set to 0,
 *          there is no aggregation, every packet is sent
 *          independently according to the aggregation structure
 *          Valid for Output Pipes only (IPA Producer)
 */
struct ipa_ep_cfg_aggr {
    enum ipa_aggr_en_type aggr_en;
    enum ipa_aggr_type aggr;
    u32 aggr_byte_limit;
    u32 aggr_time_limit;
};
```

3.2.1.9 struct ipa_ep_cfg_route

Struct type for definition of route configuration for an endpoint.

Definition

```
/**
 * struct ipa_ep_cfg_route - route configuration in IPA end-point
 * @rt_tbl_hdl: Defines the default routing table index to be used in case
 *      there is no filter rule matching, valid for Input Pipes only (IPA
 *      Consumer). Clients should set this to 0 which will cause default
 *      v4 and v6 routes setup internally by IPA driver to be used for
 *      this end-point
 */
struct ipa_ep_cfg_route {
    u32 rt_tbl_hdl;
};
```

3.2.1.10 struct ipa_ep_cfg

Struct type for definition of endpoint configuration.

Definition

```
/**
 * struct ipa_ep_cfg - configuration of IPA end-point
 * @nat:    NAT parameters
 * @hdr:    Header parameters
 * @mode:   Mode parameters
 * @aggr:   Aggregation parameters
 * @route:  Routing parameters
 */
```

```
struct ipa_ep_cfg {
    struct ipa_ep_cfg_nat nat;
    struct ipa_ep_cfg_hdr hdr;
    struct ipa_ep_cfg_mode mode;
    struct ipa_ep_cfg_aggr aggr;
    struct ipa_ep_cfg_route route;
};
```

3.2.1.11 struct ipa_connect_params

Struct type for definition of input parameters to ipa_connect.

Definition

```
/**
 * struct ipa_connect_params - low-level client connect input parameters.
 * Either client allocates the data and desc FIFO and specifies that in
 * data+desc OR specifies sizes and pipe_mem pref and IPA does the
 * allocation.
 *
 * @ipa_ep_cfg: IPA EP configuration
 * @client: type of "client"
 * @client_bam_hdl:  client SPS handle
 * @client_ep_idx:   client PER EP index
 * @priv:   callback cookie
 * @notify: callback
 *      priv - callback cookie evt - type of event data - data relevant
 *      to event.  May not be valid. See event_type enum for valid
 *      cases.
 * @desc_fifo_sz:   size of desc FIFO
 * @data_fifo_sz:   size of data FIFO
 * @pipe_mem_preferred: if true, try to alloc the FIFOs in pipe mem,
 *          fallback to sys mem if pipe mem alloc fails
 * @desc:   desc FIFO metadata when client has allocated it
 * @data:   data FIFO metadata when client has allocated it
 */
struct ipa_connect_params {
    struct ipa_ep_cfg ipa_ep_cfg;
    enum ipa_client_type client;
    u32 client_bam_hdl;
    u32 client_ep_idx;
    void *priv;
    ipa_notify_cb notify;
    u32 desc_fifo_sz;
    u32 data_fifo_sz;
    bool pipe_mem_preferred;
```

```
    struct sps_mem_buffer desc;
    struct sps_mem_buffer data;
};
```

3.2.1.12 struct ipa_sps_params

Struct type for definition of output parameters for ipa_connect. These are used by the client as input for sps_connect to connect the peripheral's endpoint in BAM-to-BAM mode with IPA.

Definition

```
/**
 *  struct ipa_sps_params - SPS related output parameters resulting from
 *  low/high level client connect
 *  @ipa_bam_hdl:   IPA SPS handle
 *  @ipa_ep_idx:    IPA PER EP index
 *  @desc:  desc FIFO metadata
 *  @data:  data FIFO metadata
 */
struct ipa_sps_params {
    u32 ipa_bam_hdl;
    u32 ipa_ep_idx;
    struct sps_mem_buffer desc;
    struct sps_mem_buffer data;
};
```

3.2.2 Functions

3.2.2.1 ipa_connect

Configures an IPA endpoint for BAM-to-BAM communication with a peripheral endpoint.

This function is called by the driver of the peripheral that wants to connect to IPA in BAM-to-BAM mode. These peripherals are A2, USB, and HSIC.

Prototype

```
int ipa_connect
(
const struct ipa_connect_params *in,
struct ipa_sps_params *sps,
u32 *clnt_hdl
);
```

Parameters

→	in	Input parameters from client
←	sps	SPS output from IPA needed by client for sps_connect
←	clnt_hdl	Opaque client handle assigned by IPA to client

Return value

`(int)`

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context.

3.2.2.2 ipa_disconnect

Disconnects an IPA endpoint from a peripheral endpoint previously configured using ipa_connect for BAM-to-BAM communication.

This function is called by the driver of the peripheral that wants to disconnect from IPA in BAM-to-BAM mode. This API expects the caller to free any needed headers, routing and filtering tables, and rules, as needed.

Prototype

```
int ipa_disconnect(u32 clnt_hdl);
```

Parameters

→	`clnt_hdl`	Opaque handle returned to client in ipa_connect

Return value

`(int)`

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context. Client handle is deallocated and must not be reused after the function returns.

3.2.2.3 ipa_cfg_ep

Configures an IPA endpoint. This function includes NAT, header, mode, aggregation, and route settings and is a one-shot function to fully configure the IPA endpoint. It can be used to reconfigure an IPA endpoint after initial configuration.

Prototype

```
int ipa_cfg_ep
(
u32 clnt_hdl,
const struct ipa_ep_cfg *ipa_ep_cfg
);
```

Parameters

→	clnt_hdl	Opaque handle returned to client in ipa_connect
→	ipa_ep_cfg	Endpoint configuration parameters

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context.

3.2.2.4 ipa_cfg_ep_nat

Performs NAT configuration of an IPA endpoint. This function can be used to reconfigure the IPA endpoint's NAT configuration after initial configuration.

Prototype

```
int ipa_cfg_ep_nat
(
u32 clnt_hdl,
const struct ipa_ep_cfg_nat *ipa_ep_cfg
);
```

Parameters

→	clnt_hdl	Opaque handle returned to client in ipa_connect
→	ipa_ep_cfg	Endpoint NAT configuration parameters

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context.

3.2.2.5 ipa_cfg_ep_hdr

Configures the header of an IPA endpoint. This function can be used to reconfigure an IPA endpoint's header after initial configuration.

Prototype

```
int ipa_cfg_ep_hdr
(
u32 clnt_hdl,
const struct ipa_ep_cfg_hdr *ipa_ep_cfg
);
```

Parameters

→	clnt_hdl	Opaque handle returned to client in ipa_connect
→	ipa_ep_cfg	Endpoint header configuration parameters

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context.

3.2.2.6 ipa_cfg_ep_mode

Configures the mode of an IPA endpoint. This function can be used to reconfigure an IPA endpoint's mode after initial configuration.

Prototype

```
int ipa_cfg_ep_mode
(
u32 clnt_hdl,
const struct ipa_ep_cfg_mode *ipa_ep_cfg
);
```

Parameters

→	clnt_hdl	Opaque handle returned to client in ipa_connect
→	ipa_ep_cfg	Endpoint mode configuration parameters

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context.

3.2.2.7 ipa_cfg_ep_aggr

Configures aggregation of an IPA endpoint. This function can be used to reconfigure an IPA endpoint's aggregation after initial configuration.

Prototype

```
int ipa_cfg_ep_aggr
(
u32 clnt_hdl,
const struct ipa_ep_cfg_aggr *ipa_ep_cfg
);
```

Parameters

→	clnt_hdl	Opaque handle returned to client in ipa_connect
→	ipa_ep_cfg	Endpoint aggregation configuration parameters

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context.

3.2.2.8 ipa_cfg_ep_route

Configures the route of an IPA endpoint. This function can be used to reconfigure an IPA endpoint's route after initial configuration.

Prototype

```
int ipa_cfg_ep_route
(
u32 clnt_hdl,
const struct ipa_ep_cfg_route *ipa_ep_cfg
);
```

Parameters

→	clnt_hdl	Opaque handle returned to client in ipa_connect
→	ipa_ep_cfg	Endpoint route configuration parameters

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context.

3.3 Header Configuration

This section describes the header configuration data types and functions.

3.3.1 Data Types

3.3.1.1 struct ipa_ioc_add_hdr

Struct type for definition of header addition parameters.

Definition

```
/**
 * struct ipa_ioc_add_hdr - header addition parameters (support
 * multiple headers and commit)
 * @commit: should headers be written to IPA hardware also?
 * @num_hdrs: num of headers that follow
 * @ipa_hdr_add hdr:    all headers need to go here back to
 *          back, no pointers
 */
```

```
struct ipa_ioc_add_hdr {
    uint8_t commit;
    uint8_t num_hdrs;
    struct ipa_hdr_add hdr[0];
};
```

3.3.1.2 struct ipa_ioc_copy_hdr

Struct type for definition of parameters used for copying a header.

Definition

```
/**
 * struct ipa_ioc_copy_hdr - retrieve a copy of the specified
 * header - caller can then derive the complete header
 * @name: name of the header resource
 * @hdr:    out parameter, contents of specified header,
 *  valid only when ioctl return val is non-negative
 * @hdr_len: out parameter, size of above header
 *  valid only when ioctl return val is non-negative
 * @is_partial: out parameter, indicates whether specified header is
 *     partial valid only when ioctl return val is non-negative
 */
struct ipa_ioc_copy_hdr {
    char name[IPA_RESOURCE_NAME_MAX];
    uint8_t hdr[IPA_HDR_MAX_SIZE];
    uint8_t hdr_len;
    uint8_t is_partial;
};
```

3.3.1.3 struct ipa_ioc_get_hdr

Struct type for definition of parameters used for looking up a header handle given the header name.

Definition

```
/**
 * struct ipa_ioc_get_hdr - header entry lookup parameters, if lookup was
 * successful caller must call put to release the reference count when done
 * @name: name of the header resource
 * @hdl:    out parameter, handle of header entry
 *      valid only when ioctl return val is non-negative
 */
struct ipa_ioc_get_hdr {
    char name[IPA_RESOURCE_NAME_MAX];
    uint32_t hdl;
};
```

3.3.1.4 struct ipa_hdr_del

Struct type for definition of header descriptor used for header deletion.

Definition

```
/**
 * struct ipa_hdr_del - header descriptor includes in and out
 * parameters
 *
 * @hdl: handle returned from header add operation
 * @status: out parameter, status of header remove operation,
 *      0 for success,
 *      -1 for failure
 */
struct ipa_hdr_del {
    uint32_t hdl;
    int status;
};
```

3.3.1.5 struct ipa_ioc_del_hdr

Struct type for definition of header deletion parameters.

Definition

```
/**
 * struct ipa_ioc_del_hdr - header deletion parameters (support
 * multiple headers and commit)
 * @commit: should headers be removed from IPA hardware also?
 * @num_hdls: num of headers being removed
 * @ipa_hdr_del hdl: all handles need to go here back to back, no pointers
 */
struct ipa_ioc_del_hdr {
    uint8_t commit;
    uint8_t num_hdls;
    struct ipa_hdr_del hdl[0];
};
```

3.3.2 Functions

3.3.2.1 ipa_add_hdr

Adds the specified headers to software and optionally commits them to IPA hardware.

Prototype

```
int ipa_add_hdr(struct ipa_ioc_add_hdr *hdrs);
```

Parameters

↔	hdrs	Set of headers to add

Equivalent user space IOCTL

```
int ioctl
(
int fd,
int cmd = IPA_IOC_ADD_HDR,
struct ipa_ioc_add_hdr *hdrs
);
```

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context.

3.3.2.2 ipa_del_hdr

Deletes specified headers from software and optionally commits them to IPA hardware.

Prototype

```
int ipa_del_hdr(struct ipa_ioc_del_hdr *hdls);
```

Parameters

↔	hdls	Set of headers to delete

Equivalent user space IOCTL

```
int ioctl
(
int fd,
int cmd = IPA_IOC_DEL_HDR,
struct ipa_ioc_del_hdr *hdls
);
```

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context.

3.3.2.3 ipa_commit_hdr

Commits the current header table in software to IPA hardware.

Prototype

```
int ipa_commit_hdr(void);
```

Equivalent user space IOCTL

```
int ioctl
(
int fd,
int cmd = IPA_IOC_COMMIT_HDR
);
```

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context.

3.3.2.4 ipa_get_hdr

Looks up the specified header resource and returns a handle if it exists. If the lookup succeeds, the header entry reference count is increased.

Prototype

```
int ipa_get_hdr(struct ipa_ioc_get_hdr *lookup);
```

Parameters

↔	lookup	Header to look up

Equivalent user space IOCTL

```
int ioctl
(
int fd,
int cmd = IPA_IOC_GET_HDR,
struct ipa_ioc_get_hdr *lookup
);
```

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context. If this function call succeeds, then call ipa_put_hdr..

3.3.2.5 ipa_put_hdr

Releases a specified header handle.

Prototype

```
int ipa_put_hdr(u32 hdr_hdl)
```

Parameters

→	hdr_hdl	Header handle to release

Equivalent user space IOCTL

```
int ioctl
(
int fd,
int cmd = IPA_IOC_PUT_HDR,
u32 hdr_hdl
);
```

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context.

3.3.2.6 ipa_copy_hdr

Copies a header. This function looks up the specified header resource and returns a copy of it (along with its attributes) if it exists. This function is called for partial headers.

Prototype

```
int ipa_copy_hdr(struct ipa_ioc_copy_hdr *copy);
```

Parameters

↔	copy	Header to copy

Equivalent user space IOCTL

```
int ioctl
(
int fd,
int cmd = IPA_IOC_COPY_HDR,
struct ipa_ioc_copy_hdr *copy
);
```

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context.

3.4 Routing Configuration

This section describes the routing configuration data types and functions.

3.4.1 Data Types

3.4.1.1 struct ipa_rt_rule

Struct type defining a routing rule.

Definition

```
/**
 * struct ipa_rt_rule - attributes of a routing rule
 * @dst: dst "client"
 * @hdr_hdl: handle to the dynamic header
    it is not an index or an offset
 * @attrib: attributes of the rule
 */
struct ipa_rt_rule {
    enum ipa_client_type dst;
    uint32_t hdr_hdl;
    struct ipa_rule_attrib attrib;
};
```

3.4.1.2 struct ipa_rt_rule_add

Struct type for definition of routing rule descriptor used in route rule addition API.

Definition

```
/**
 * struct ipa_rt_rule_add - routing rule descriptor includes in
 * and out parameters
 * @rule: actual rule to be added
 * @at_rear:    add at back of routing table, it is NOT possible to add
 *      rules at the rear of the "default" routing tables
 * @rt_rule_hdl: output parameter, handle to rule, valid when status is 0
 * @status: output parameter, status of routing rule add operation,
 *      0 for success,
 *      -1 for failure
 */
struct ipa_rt_rule_add {
    struct ipa_rt_rule rule;
```

```
    uint8_t at_rear;
    uint32_t rt_rule_hdl;
    int status;
};
```

3.4.1.3 struct ipa_ioc_add_rt_rule

Struct type for definition of routing rules to add. This struct supports multiple rules and commit.

Definition

```
/**
 * struct ipa_ioc_add_rt_rule - routing rule addition parameters (supports
 * multiple rules and commit);
 *
 * all rules MUST be added to same table
 * @commit: should rules be written to IPA hardware also?
 * @ip: IP family of rule
 * @rt_tbl_name: name of routing table resource
 * @num_rules: number of routing rules that follow
 * @ipa_rt_rule_add rules: all rules need to go back to back here,
 *        no pointers
 */
struct ipa_ioc_add_rt_rule {
    uint8_t commit;
    enum ipa_ip_type ip;
    char rt_tbl_name[IPA_RESOURCE_NAME_MAX];
    uint8_t num_rules;
    struct ipa_rt_rule_add rules[0];
};
```

3.4.1.4 struct ipa_rt_rule_del

Struct type for definition of routing rule descriptor used in routing rule deletion API.

Definition

```
/**
 * struct ipa_rt_rule_del - routing rule descriptor includes in
 * and out parameters
 * @hdl: handle returned from route rule add operation
 * @status: output parameter, status of route rule delete operation,
 *      0 for success,
 *      -1 for failure
 */
struct ipa_rt_rule_del {
    uint32_t hdl;
    int status;
```

```
};
```

3.4.1.5 struct ipa_ioc_del_rt_rule

Struct type for definition of routing rules to delete.

Definition

```
/**
 * struct ipa_ioc_del_rt_rule - routing rule deletion parameters (supports
 * multiple headers and commit)
 * @commit: should rules be removed from IPA hardware also?
 * @ip: IP family of rules
 * @num_hdls: num of rules being removed
 * @ipa_rt_rule_del hdl: all handles need to go back to back here, no
 *       pointers
 */
struct ipa_ioc_del_rt_rule {
    uint8_t commit;
    enum ipa_ip_type ip;
    uint8_t num_hdls;
    struct ipa_rt_rule_del hdl[0];
};
```

3.4.1.6 struct ipa_ioc_get_rt_tbl

Struct type for definition parameters for looking up a route table.

Definition

```
/**
 * struct ipa_ioc_get_rt_tbl - routing table lookup parameters, if lookup
 * was successful caller must call put to release the reference
 * count when done
 * @ip: IP family of table
 * @name: name of routing table resource
 * @hdl:    output parameter, handle of routing table, valid only when
 *      ioctl return val is non-negative
 */
struct ipa_ioc_get_rt_tbl {
    enum ipa_ip_type ip;
    char name[IPA_RESOURCE_NAME_MAX];
    uint32_t hdl;
};
```

3.4.2 Functions

3.4.2.1 ipa_add_rt_rule

Configures routing rules. This function adds the specified routing rules to software and optionally commits them to IPA hardware.

Prototype

```
int ipa_add_rt_rule(struct ipa_ioc_add_rt_rule *rules);
```

Parameters

↔	rules	Set of routing rules to add

Equivalent user space IOCTL

```
int ioctl
(
int fd,
int cmd = IPA_IOC_ADD_RT_RULE,
struct ipa_ioc_add_rt_rule *rules
);
```

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context.

3.4.2.2 ipa_del_rt_rule

Deletes the specified routing rules from software and optionally from IPA hardware.

Prototype

```
int ipa_del_rt_rule(struct ipa_ioc_del_rt_rule *hdls);
```

Parameters

↔	hdls	Set of routing rules to delete

Equivalent user space IOCTL

```
int ioctl
(
int fd,
int cmd = IPA_IOC_DEL_RT_RULE,
struct ipa_ioc_del_rt_rule *hdls
);
```

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context.

3.4.2.3 ipa_commit_rt

Commits the current software routing table of the specified type to IPA hardware.

Prototype

```
int ipa_commit_rt(enum ipa_ip_type ip);
```

Parameters

→	ip	The family of routing tables to commit

Equivalent user space IOCTL

```
int ioctl
(
int fd,
int cmd = IPA_IOC_COMMIT_RT
);
```

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context.

3.4.2.4 ipa_get_rt_tbl

Looks up a specified routing table and returns a handle if it exists. If the lookup succeeds the routing table reference count is increased.

Prototype

```
int ipa_get_rt_tbl(struct ipa_ioc_get_rt_tbl *lookup);
```

Parameters

↔	`lookup`	Routing table to look up and its handle

Equivalent user space IOCTL

```
int ioctl
(
int fd,
int cmd = IPA_IOC_GET_RT_TBL,
struct ipa_ioc_get_rt_tbl *lookup
);
```

Return value

`(int)`

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context. If this function call succeeds, then call ipa_put_rt_tbl.

3.4.2.5 ipa_put_rt_tbl

Releases the specified routing table handle.

Prototype

```
int ipa_put_rt_tbl(u32 rt_tbl_hdl);
```

Parameters

→	`rt_tbl_hdl`	Handle of routing table to release

Equivalent user space IOCTL

```
int ioctl
(
int fd,
int cmd = IPA_IOC_PUT_RT_TBL,
uint32_t rt_tbl_hdl
);
```

Return value

(int)

0 on success; negative on failure.

Dependencies

Caller must obtain the handle with ipa_get_rt_tbl before using this function.

Side effects

May block; do not call from an atomic context.

3.5 Filtering Configuration

This section describes the filtering configuration data types and functions.

3.5.1 Data Types

3.5.1.1 enum ipa_flt_action

Enumeration of the action to take when a filter rule matches.

Definition

```
/**
 * enum ipa_flt_action - action field of filtering rule
 *
 * Pass to routing: 5'd0
 * Pass to source NAT: 5'd1
 * Pass to destination NAT: 5'd2
 * Pass to default output pipe (e.g., A5): 5'd3
 */
enum ipa_flt_action {
    IPA_PASS_TO_ROUTING,
    IPA_PASS_TO_SRC_NAT,
    IPA_PASS_TO_DST_NAT,
    IPA_PASS_TO_EXCEPTION
};
```

3.5.1.2 struct ipa_flt_rule

Struct type definition of a filter rule.

Definition

```
/**
 * struct ipa_flt_rule - attributes of a filtering rule
 * @action: action field
 * @rt_tbl_hdl: handle of table from "get"
 * @attrib: attributes of the rule
 */
struct ipa_flt_rule {
    enum ipa_flt_action action;
    uint32_t rt_tbl_hdl;
    struct ipa_rule_attrib attrib;
};
```

3.5.1.3 struct ipa_flt_rule_add

Struct type definition of parameters for filter rule addition.

Definition

```
/**
 * struct ipa_flt_rule_add - filtering rule descriptor includes
 * in and out parameters
 * @rule: actual rule to be added
 * @at_rear: add at back of filtering table?
 * @flt_rule_hdl: out parameter, handle to rule, valid when status is 0
 * @status: output parameter, status of filtering rule add operation,
 *      0 for success,
 *      -1 for failure
 *
 */
struct ipa_flt_rule_add {
    struct ipa_flt_rule rule;
    uint8_t at_rear;
    uint32_t flt_rule_hdl;
    int status;
};
```

3.5.1.4 struct ipa_ioc_add_flt_rule

Struct type for definition of parameters used for filter rule addition.

Definition

```
/**
 * struct ipa_ioc_add_flt_rule - filtering rule addition parameters
 * (supports multiple rules and commit)
 * all rules MUST be added to same table
 * @commit: should rules be written to IPA hardware also?
 * @ip: IP family of rule
 * @ep: which "clients" pipe does this rule apply to?
 *  valid only when global is 0
 * @global: does this apply to global filter table of specific IP family
 * @num_rules: number of filtering rules that follow
 * @rules: all rules need to go back to back here, no pointers
 */
struct ipa_ioc_add_flt_rule {
    uint8_t commit;
    enum ipa_ip_type ip;
    enum ipa_client_type ep;
    uint8_t global;
    uint8_t num_rules;
    struct ipa_flt_rule_add rules[0];
};
```

3.5.1.5 struct ipa_flt_rule_del

Struct type for definition of filter rule descriptor used in filter deletion.

Definition

```
/**
 * struct ipa_flt_rule_del - filtering rule descriptor includes
 * in and out parameters
 *
 * @hdl: handle returned from filtering rule add operation
 * @status: output parameter, status of filtering rule delete operation,
 *      0 for success,
 *      -1 for failure
 */
struct ipa_flt_rule_del {
    uint32_t hdl;
    int status;
};
```

3.5.1.6 struct ipa_ioc_del_flt_rule

Struct type for definition of parameters used for filter rule deletion.

Definition

```
/**
 * struct ipa_ioc_del_flt_rule - filtering rule deletion parameters
 * (supports multiple headers and commit)
 * @commit: should rules be removed from IPA hardware also?
 * @ip: IP family of rules
 * @num_hdls: num of rules being removed
 * @hdl: all handles need to go back to back here, no pointers
 */
struct ipa_ioc_del_flt_rule {
    uint8_t commit;
    enum ipa_ip_type ip;
    uint8_t num_hdls;
    struct ipa_flt_rule_del hdl[0];
};
```

3.5.2 Functions

3.5.2.1 ipa_add_flt_rule

Adds the specified filtering rules to software and optionally commits them to IPA hardware.

Prototype

```
int ipa_add_flt_rule(struct ipa_ioc_add_flt_rule *rules);
```

Parameters

↔	rules	Set of filtering rules to add

Equivalent user space IOCTL

```
int ioctl
(
int fd,
int cmd = IPA_IOC_ADD_FLT_RULE,
struct ipa_ioc_add_flt_rule *rules
);
```

Return value

```
(int)
```

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context.

3.5.2.2 ipa_del_flt_rule

Deletes the specified filtering rules from software and optionally commits them to IPA hardware.

Prototype

```
int ipa_del_flt_rule(struct ipa_ioc_del_flt_rule *hdls);
```

Parameters

↔	`hdls`	Set of filtering rules to delete

Equivalent user space IOCTL

```
int ioctl
(
int fd,
int cmd = IPA_IOC_DEL_FLT_RULE,
struct ipa_ioc_del_flt_rule *hdls
);
```

Return value

`(int)`

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context.

3.5.2.3 ipa_commit_flt

Commits filter rules to IPA hardware. This function commits the current software filtering table of a specified type to IPA hardware.

Prototype

```
int ipa_commit_flt(enum ipa_ip_type ip);
```

Parameters

→	`ip`	The family of filter tables to commit

Equivalent user space IOCTL

```
int ioctl
(
int fd,
int cmd = IPA_IOC_COMMIT_FLT,
enum ipa_ip_type ip
);
```

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context.

3.5.2.4 ipa_reset_flt

Resets the software filtering tables of a specified IP type.

NOTE: This function does not commit the change to hardware.

Prototype

```
int ipa_reset_flt(enum ipa_ip_type ip);
```

Parameters

→	ip	The family of filter tables to reset

Equivalent user space IOCTL

```
int ioctl
(
int fd,
int cmd = IPA_IOC_RESET_FLT,
enum ipa_ip_type ip
);
```

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

May block; do not call from an atomic context.

3.6 NAT Configuration

This section describes the NAT configuration data types and functions.

3.6.1 Data Types

3.6.1.1 struct ipa_ioc_nat_alloc_mem

Struct type used in IPA_IOC_ALLOC_NAT_MEM IOCTL.

Definition

```
/**
 * struct ipa_ioc_nat_alloc_mem - NAT table memory allocation
 * properties
 * @dev_name: input parameter, the name of table
 * @size: input parameter, size of table in bytes
 * @offset: output parameter, offset into page in case of system memory
 */
struct ipa_ioc_nat_alloc_mem {
    char dev_name[IPA_RESOURCE_NAME_MAX];
    size_t size;
    off_t offset;
};
```

3.6.1.2 struct ipa_ioc_v4_nat_init

Struct type used in IPA_IOC_V4_INIT_NAT IOCTL.

Definition

```
/**
 * struct ipa_ioc_v4_nat_init – NAT table initialization
 * parameters
 * @tbl_index: input parameter, index of the table
 * @ipv4_rules_offset: input parameter, IPv4 rules address offset
 * @expn_rules_offset: input parameter, IPv4 expansion rules address offset
 * @index_offset: input parameter, index rules offset
 * @index_expn_offset: input parameter, index expansion rules offset
 * @table_entries: input parameter, IPv4 rules table size in entries
 * @expn_table_entries: input parameter, IPv4 expansion rules table size
 * @ip_addr: input parameter, public IP address
 */
struct ipa_ioc_v4_nat_init {
```

```
    uint8_t tbl_index;
    uint32_t ipv4_rules_offset;
    uint32_t expn_rules_offset;

    uint32_t index_offset;
    uint32_t index_expn_offset;

    uint16_t table_entries;
    uint16_t expn_table_entries;
    uint32_t ip_addr;
};
```

3.6.1.3 struct ipa_ioc_v4_nat_del

Struct type used in IPA_IOC_V4_DEL_NAT IOCTL.

Definition

```
/**
 * struct ipa_ioc_v4_nat_del - NAT table delete parameter
 * @table_index: input parameter, index of the table
 * @public_ip_addr: input parameter, public IP address
 */
struct ipa_ioc_v4_nat_del {
    uint8_t table_index;
    uint32_t public_ip_addr;
};
```

3.6.1.4 struct ipa_ioc_nat_dma_one

Struct type to hold NAT DMA command parameters.

Definition

```
/**
 * struct ipa_ioc_nat_dma_one - NAT DMA command parameter
 * @table_index: input parameter, index of the table
 * @base_addr:  type of table, from which the base address of the table
 *      can be inferred
 * @offset: destination offset within the NAT table
 * @data: data to be written.
 */
struct ipa_ioc_nat_dma_one {
    uint8_t table_index;
    uint8_t base_addr;
    uint32_t offset;
    uint16_t data;

```

```
};
```

3.6.1.5 struct ipa_ioc_nat_dma_cmd

Struct type used in IPA_IOC_NAT_DMA IOCTL.

Definition

```
/**
 * struct ipa_ioc_nat_dma_cmd - To hold multiple NAT DAT commands
 * @entries: number of DMA commands in use
 * @dma: data pointer to the DMA commands
 */
struct ipa_ioc_nat_dma_cmd {
    uint8_t entries;
    struct ipa_ioc_nat_dma_one dma[0];

};
```

3.6.2 Functions

3.6.2.1 IPA_IOC_ALLOC_NAT_MEM

Allocates memory for NAT table entries. Based on size, memory is allocated from either IPA local memory or system memory.

Prototype (user space IOCTL)

```
int ioctl
(
int fd,
int cmd = IPA_IOC_ALLOC_NAT_MEM,
struct ipa_ioc_nat_alloc_mem *mem
);
```

Return value

`(int)`

0 on success; negative on failure.

Dependencies

N/A

Side effects

N/A

3.6.2.2 IPA_IOC_V4_INIT_NAT

Allocates memory and initializes the NAT table.

This function must be called before adding any entries to the table.

Prototype (user space IOCTL)

```
int ioctl
(
int fd,
int cmd = IPA_IOC_V4_INIT_NAT,
struct ipa_ioc_v4_nat_init *init
);
```

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

N/A

3.6.2.3 IPA_IOC_NAT_DMA

Posts a NAT DMA command to IPA hardware to perform atomic updates to the NAT table.

Prototype (user space IOCTL)

```
int ioctl
(
int fd,
int cmd = IPA_IOC_NAT_DMA,
struct ipa_ioc_nat_dma_cmd *dma
);
```

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

N/A

3.6.2.4 IPA_IOC_V4_DEL_NAT

Deletes the NAT table.

Prototype (user space IOCTL)

```
int ioctl
(
int fd,
int cmd = IPA_IOC_V4_DEL_NAT,
struct ipa_ioc_v4_nat_del *del
);
```

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

N/A

3.7 Data Transfer

This section describes the data transfer data types and functions.

3.7.1 Data Types

3.7.1.1 enum ipa_dp_evt_type

Enumeration of event types dispatched to client.

Definition

```
/**
 * enum ipa_dp_evt_type - type of event client callback is
 * invoked for on data path
 * @IPA_RECEIVE: data is struct sk_buff
 * @IPA_WRITE_DONE: data is struct sk_buff
 */
enum ipa_dp_evt_type {
    IPA_RECEIVE,
    IPA_WRITE_DONE,
};
```

3.7.1.2 struct ipa_tx_meta

Struct type defining metadata parameters of a packet. This struct is used in ipa_tx_dp.

Definition

```
/**
 * struct ipa_tx_meta - metadata for the Tx packet
 * @mbim_stream_id: the stream ID used in NDP signature
 * @mbim_stream_id_valid:    is above field valid?
 */
struct ipa_tx_meta {
    u8 mbim_stream_id;
    bool mbim_stream_id_valid;
};
```

3.7.2 Functions

3.7.2.1 ipa_tx_dp

Transmits a packet.

This function is a data path Tx handler, used for both the software data path that bypasses most IPA hardware blocks *and* the regular hardware data path for WLAN A-MPDU traffic only. If dst is a "valid" CONS type, then the software data path is used. If dst is WLAN_AMPDU PROD type, then the hardware data path for WLAN A-MPDU is used. Anything else is an error. For errors, the client must free the skb as needed. For success, the IPA driver later invokes the client calback if one was supplied. That callback is assumed to free the skb. If no callback is supplied, the IPA driver frees the skb internally.

Prototype

```
int ipa_tx_dp
(
enum ipa_client_type dst,
struct sk_buff *skb,
struct ipa_tx_meta *metadata
);
```

Parameters

→	`dst`	Destination of the packet
→	`skb`	Packet to send
→	`metadata`	Packet metadata

Return value

`(int)`

0 on success; negative on failure.

Dependencies

N/A

Side effects

None

3.8 Interface Registration and Lookup

This section describes the interface registration and lookup data types and functions.

3.8.1 Data Types

3.8.1.1 struct ipa_ioc_query_intf

Struct type for looking up the number of Tx and Rx properties of the interface.

Definition

```
/**
 * struct ipa_ioc_query_intf - used to look up number of Tx and
 * Rx properties of interface
 * @name: name of interface
 * @num_tx_props:   output parameter, number of Tx properties
 *          valid only when ioctl return val is non-negative
 * @num_rx_props:   output parameter, number of Rx properties
 *          valid only when ioctl return val is non-negative
 */
struct ipa_ioc_query_intf {
    char name[IPA_RESOURCE_NAME_MAX];
    uint32_t num_tx_props;
    uint32_t num_rx_props;
};
```

3.8.1.2 struct ipa_ioc_tx_intf_prop

Struct type defining a Tx interface property.

Definition

```
/**
 * struct ipa_ioc_tx_intf_prop - interface Tx property
 * @ip: IP family of routing rule
 * @attrib: routing rule
 * @dst_pipe: routing output pipe
 * @hdr_name: name of associated header if any, empty string when no header
 */
```

```
struct ipa_ioc_tx_intf_prop {
    enum ipa_ip_type ip;
    struct ipa_rule_attrib attrib;
    enum ipa_client_type dst_pipe;
    char hdr_name[IPA_RESOURCE_NAME_MAX];
};
```

3.8.1.3 struct ipa_ioc_query_intf_tx_props

Struct type for looking up interface Tx properties.

Definition

```
/**
 * struct ipa_ioc_query_intf_tx_props - interface Tx properties
 * @name: name of interface
 * @num_tx_props: number of Tx properties
 * @tx[0]: output parameter, the Tx properties go here back to back
 */
struct ipa_ioc_query_intf_tx_props {
    char name[IPA_RESOURCE_NAME_MAX];
    uint32_t num_tx_props;
    struct ipa_ioc_tx_intf_prop tx[0];
};
```

3.8.1.4 struct ipa_ioc_rx_intf_prop

Struct type defining a Rx interface property.

Definition

```
/**
 * struct ipa_ioc_rx_intf_prop - interface Rx property
 * @ip: IP family of filtering rule
 * @attrib: filtering rule
 * @src_pipe: input pipe
 */
struct ipa_ioc_rx_intf_prop {
    enum ipa_ip_type ip;
    struct ipa_rule_attrib attrib;
    enum ipa_client_type src_pipe;
};
```

3.8.1.5 struct ipa_ioc_query_intf_rx_props

Struct type for looking up interface Rx properties.

Definition

```
/**
 * struct ipa_ioc_query_intf_rx_props - interface Rx properties
 * @name: name of interface
 * @num_rx_props: number of Rx properties
 * @rx: output parameter, the Rx properties go here back to back
 */
struct ipa_ioc_query_intf_rx_props {
    char name[IPA_RESOURCE_NAME_MAX];
    uint32_t num_rx_props;
    struct ipa_ioc_rx_intf_prop rx[0];
};
```

3.8.1.6 struct ipa_tx_intf

Struct type defining interface Tx properties.

Definition

```
/**
 * struct ipa_tx_intf - interface Tx properties
 * @num_props:  number of Tx properties
 * @prop:   the Tx properties array
 */
struct ipa_tx_intf {
    u32 num_props;
    struct ipa_ioc_tx_intf_prop *prop;
};
```

3.8.1.7 struct ipa_rx_intf

Struct type defining interface Rx properties.

Definition

```
/**
 * struct ipa_rx_intf - interface Rx properties
 * @num_props:  number of Rx properties
 * @prop:   the Rx properties array
 */
struct ipa_rx_intf {
    u32 num_props;
    struct ipa_ioc_rx_intf_prop *prop;
};
```

3.8.2 Functions

3.8.2.1 ipa_register_intf

Registers an interface and its associated Tx and Rx properties with the IPA driver. This allows configuration of rules from the user space.

Prototype

```
int ipa_register_intf
(
const char *name,
const struct ipa_tx_intf *tx,
const struct ipa_rx_intf *rx
);
```

Parameters

→	name	Interface name
→	tx	Tx properties of the interface
→	rx	Rx properties of the interface

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

None

3.8.2.2 ipa_deregister_intf

Deregisters a previously registered interface with the IPA driver.

Prototype

```
int ipa_deregister_intf(const char *name);
```

Parameters

→	name	Interface name

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

None

3.8.2.3 IPA_IOC_QUERY_INTF

Looks up an interface and obtains its handle and the number of its registered Tx and Rx properties. This function is used as part of querying the Tx and Rx properties for configuration of various rules from user space.

Prototype (user space IOCTL)

```
int ioctl
(
int fd,
int cmd = IPA_IOC_QUERY_INTF,
struct ipa_ioc_query_intf *lookup
);
```

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

None

3.8.2.4 IPA_IOC_QUERY_INTF_TX_PROPS

Queries and obtains the Tx properties of an interface.

Prototype (user space IOCTL)

```
int ioctl
(
int fd,
int cmd = IPA_IOC_QUERY_INTF_TX_PROPS,
struct ipa_ioc_query_intf_tx_props *tx
);
```

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

None

3.8.2.5 IPA_IOC_QUERY_INTF_RX_PROPS

Queries and obtains the Rx properties of an interface.

Prototype (user space IOCTL)

```
int ioctl
(
int fd,
int cmd = IPA_IOC_QUERY_INTF_RX_PROPS,
struct ipa_ioc_query_intf_rx_props *rx
);
```

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

None

3.9 Generic Notification Service

This section describes the generic notification service data types and functions.

3.9.1 Data Types

3.9.1.1 struct ipa_msg_meta

Struct type defining the format of the metadata corresponding to a message.

Definition

```
/**
 * struct ipa_msg_meta - Format of the message metadata.
 * @msg_type: the type of the message
 * @rsvd: reserved bits for future use.
 * @msg_len: the length of the message in bytes
 *
```

```
 * For push model:
 * Client in user space should issue a read on the device (/dev/ipa) with a
 * sufficiently large buffer in a continuous loop, call will block when there is
 * no message to read. Upon return, client can read the ipa_msg_meta from start
 * of buffer to find out type and length of message
 * size of buffer supplied >= (size of largest message + size of metadata)
 *
 * For pull model:
 * Client in user space can also issue a pull msg IOCTL to device (/dev/ipa)
 * with a payload containing space for the ipa_msg_meta and the message specific
 * payload length.
 * size of buffer supplied == (len of specific message  + size of metadata)
 */
struct ipa_msg_meta {
    uint8_t msg_type;
    uint8_t rsvd;
    uint16_t msg_len;
};
```

3.9.1.2 struct ipa_wlan_msg

Struct type defining the WLAN message from WLAN client drivers to user space.

Definition

```
/**
 * struct ipa_wlan_msg - To hold information about WLAN client
 * @name: name of the WLAN interface
 * @mac_addr: MAC address of WLAN client
 *
 * WLAN drivers need to pass name of WLAN interface and MAC address of
 * WLAN client along with ipa_wlan_event, whenever a WLAN client is
 * connected/disconnected/moved to power save/come out of power save
 */
struct ipa_wlan_msg {
  char name[IPA_RESOURCE_NAME_MAX];
   uint8_t mac_addr[IPA_MAC_ADDR_SIZE];
};
```

3.9.2 Functions

3.9.2.1 ipa_send_msg

Sends a message.

The client supplies the message metadata and payload that the IPA driver buffers until they are read from user space. After the read, the IPA driver invokes the callback supplied to free the message payload. The client must not touch or free the message payload after calling this function.

Prototype

```
int ipa_send_msg
(
struct ipa_msg_meta *meta,
void *buff,
ipa_msg_free_fn callback
);
```

Parameters

→	`meta`	Message metadata
→	`buff`	Message payload
→	`callback`	Callback function invoked to free message upon read by user space

Return value

`(int)`

0 on success; negative on failure.

Dependencies

N/A

Side effects

N/A

3.9.2.2 ipa_register_pull_msg

Registers a pull message type.

This function registers the message callback by a kernel client with the IPA driver for the IPA driver to pull a message on demand.

Prototype

```
int ipa_register_pull_msg
(
struct ipa_msg_meta *meta,
ipa_msg_pull_fn callback
);
```

Parameters

→	meta	Message metadata
→	callback	Callback function to notify the client that the message was received

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

N/A

3.9.2.3 ipa_deregister_pull_msg

Deregisters a message callback for a pull message.

Prototype

```
int ipa_deregister_pull_msg(struct ipa_msg_meta *meta);
```

Parameters

→	meta	Message metadata

Return value

(int)

0 on success; negative on failure.

Dependencies

N/A

Side effects

N/A