Linux Accel Documentation

The kernel development community

Jun 10, 2024

CONTENTS

CHAPTER

ONE

INTRODUCTION

The Linux compute accelerators subsystem is designed to expose compute accelerators in a common way to user-space and provide a common set of functionality.

These devices can be either stand-alone ASICs or IP blocks inside an SoC/GPU. Although these devices are typically designed to accelerate Machine-Learning (ML) and/or Deep-Learning (DL) computations, the accel layer is not limited to handling these types of accelerators.

Typically, a compute accelerator will belong to one of the following categories:

- Edge AI - doing inference at an edge device. It can be an embedded ASIC/FPGA, or an IP inside a SoC (e.g. laptop web camera). These devices are typically configured using registers and can work with or without DMA.
- Inference data-center - single/multi user devices in a large server. This type of device can be stand-alone or an IP inside a SoC or a GPU. It will have on-board DRAM (to hold the DL topology), DMA engines and command submission queues (either kernel or user-space queues). It might also have an MMU to manage multiple users and might also enable virtualization (SR-IOV) to support multiple VMs on the same device. In addition, these devices will usually have some tools, such as profiler and debugger.
- Training data-center - Similar to Inference data-center cards, but typically have more computational power and memory b/w (e.g. HBM) and will likely have a method of scaling-up/out, i.e. connecting to other training cards inside the server or in other servers, respectively.

All these devices typically have different runtime user-space software stacks, that are tailored-made to their h/w. In addition, they will also probably include a compiler to generate programs to their custom-made computational engines. Typically, the common layer in user-space will be the DL frameworks, such as PyTorch and TensorFlow.

1.1 Sharing code with DRM

Because this type of devices can be an IP inside GPUs or have similar characteristics as those of GPUs, the accel subsystem will use the DRM subsystem's code and functionality. i.e. the accel core code will be part of the DRM subsystem and an accel device will be a new type of DRM device.

This will allow us to leverage the extensive DRM code-base and collaborate with DRM developers that have experience with this type of devices. In addition, new features that will be added for the accelerator drivers can be of use to GPU drivers as well.

1.2 Differentiation from GPUs

Because we want to prevent the extensive user-space graphic software stack from trying to use an accelerator as a GPU, the compute accelerators will be differentiated from GPUs by using a new major number and new device char files.

Furthermore, the drivers will be located in a separate place in the kernel tree - drivers/accel/.

The accelerator devices will be exposed to the user space with the dedicated 261 major number and will have the following convention:

- device char files - /dev/accel/accel*
- sysfs - /sys/class/accel/accel*/
- debugfs - /sys/kernel/debug/accel/*/

1.3 Getting Started

First, read the DRM documentation at Documentation/gpu/index.rst. Not only it will explain how to write a new DRM driver but it will also contain all the information on how to contribute, the Code Of Conduct and what is the coding style/documentation. All of that is the same for the accel subsystem.

Second, make sure the kernel is configured with CONFIG_DRM_ACCEL.

To expose your device as an accelerator, two changes are needed to be done in your driver (as opposed to a standard DRM driver):

- Add the DRIVER_COMPUTE_ACCEL feature flag in your drm_driver's driver_features field. It is important to note that this driver feature is mutually exclusive with DRIVER_RENDER and DRIVER_MODESET. Devices that want to expose both graphics and compute device char files should be handled by two drivers that are connected using the auxiliary bus framework.
- Change the open callback in your driver fops structure to accel_open(). Alternatively, your driver can use DEFINE_DRM_ACCEL_FOPS macro to easily set the correct function operations pointers structure.

1.4 External References

1.4.1 email threads

- Initial discussion on the New subsystem for acceleration devices - Oded Gabbay (2022)
- patch-set to add the new subsystem - Oded Gabbay (2022)

1.4.2 Conference talks

- LPC 2022 Accelerators BOF outcomes summary - Dave Airlie (2022)

CHAPTER

TWO

ACCEL/QAIC QUALCOMM CLOUD AI DRIVER

The accel/qaic driver supports the Qualcomm Cloud AI machine learning accelerator cards.

2.1 QAIC driver

The QAIC driver is the Kernel Mode Driver (KMD) for the AIC100 family of AI accelerator products.

2.1.1 Interrupts

While the AIC100 DMA Bridge hardware implements an IRQ storm mitigation mechanism, it is still possible for an IRQ storm to occur. A storm can happen if the workload is particularly quick, and the host is responsive. If the host can drain the response FIFO as quickly as the device can insert elements into it, then the device will frequently transition the response FIFO from empty to non-empty and generate MSIs at a rate equivalent to the speed of the workload's ability to process inputs. The lprnet (license plate reader network) workload is known to trigger this condition, and can generate in excess of 100k MSIs per second. It has been observed that most systems cannot tolerate this for long, and will crash due to some form of watchdog due to the overhead of the interrupt controller interrupting the host CPU.

To mitigate this issue, the QAIC driver implements specific IRQ handling. When QAIC receives an IRQ, it disables that line. This prevents the interrupt controller from interrupting the CPU. Then AIC drains the FIFO. Once the FIFO is drained, QAIC implements a "last chance" polling algorithm where QAIC will sleep for a time to see if the workload will generate more activity. The IRQ line remains disabled during this time. If no activity is detected, QAIC exits polling mode and reenables the IRQ line.

This mitigation in QAIC is very effective. The same lprnet usecase that generates 100k IRQs per second (per /proc/interrupts) is reduced to roughly 64 IRQs over 5 minutes while keeping the host system stable, and having the same workload throughput performance (within run to run noise variation).

2.1.2 Neural Network Control (NNC) Protocol

The implementation of NNC is split between the KMD (QAIC) and UMD. In general QAIC understands how to encode/decode NNC wire protocol, and elements of the protocol which require kernel space knowledge to process (for example, mapping host memory to device IOVAs). QAIC understands the structure of a message, and all of the transactions. QAIC does not understand commands (the payload of a passthrough transaction).

QAIC handles and enforces the required little endianness and 64-bit alignment, to the degree that it can. Since QAIC does not know the contents of a passthrough transaction, it relies on the UMD to satisfy the requirements.

The terminate transaction is of particular use to QAIC. QAIC is not aware of the resources that are loaded onto a device since the majority of that activity occurs within NNC commands. As a result, QAIC does not have the means to roll back userspace activity. To ensure that a userspace client's resources are fully released in the case of a process crash, or a bug, QAIC uses the terminate command to let QSM know when a user has gone away, and the resources can be released.

QSM can report a version number of the NNC protocol it supports. This is in the form of a Major number and a Minor number.

Major number updates indicate changes to the NNC protocol which impact the message format, or transactions (impacts QAIC).

Minor number updates indicate changes to the NNC protocol which impact the commands (does not impact QAIC).

2.1.3 uAPI

QAIC defines a number of driver specific IOCTLs as part of the userspace API. This section describes those APIs.

DRM_IOCTL_QAIC_MANAGE
: This IOCTL allows userspace to send a NNC request to the QSM. The call will block until a response is received, or the request has timed out.

DRM_IOCTL_QAIC_CREATE_BO
: This IOCTL allows userspace to allocate a buffer object (BO) which can send or receive data from a workload. The call will return a GEM handle that represents the allocated buffer. The BO is not usable until it has been sliced (see DRM_IOCTL_QAIC_ATTACH_SLICE_BO).

DRM_IOCTL_QAIC_MMAP_BO
: This IOCTL allows userspace to prepare an allocated BO to be mmap'd into the userspace process.

DRM_IOCTL_QAIC_ATTACH_SLICE_BO
: This IOCTL allows userspace to slice a BO in preparation for sending the BO to the device. Slicing is the operation of describing what portions of a BO get sent where to a workload. This requires a set of DMA transfers for the DMA Bridge, and as such, locks the BO to a specific DBC.

DRM_IOCTL_QAIC_EXECUTE_BO
: This IOCTL allows userspace to submit a set of sliced BOs to the device. The call is non-blocking. Success only indicates that the BOs have been queued to the device, but does not guarantee they have been executed.

DRM_IOCTL_QAIC_PARTIAL_EXECUTE_BO
: This IOCTL operates like DRM_IOCTL_QAIC_EXECUTE_BO, but it allows userspace to shrink the BOs sent to the device for this specific call. If a BO typically has N inputs, but only a subset of those is available, this IOCTL allows userspace to indicate that only the first M bytes of the BO should be sent to the device to minimize data transfer overhead. This IOCTL dynamically recomputes the slicing, and therefore has some processing overhead before the BOs can be queued to the device.

DRM_IOCTL_QAIC_WAIT_BO
: This IOCTL allows userspace to determine when a particular BO has been processed by the device. The call will block until either the BO has been processed and can be re-queued to the device, or a timeout occurs.

DRM_IOCTL_QAIC_PERF_STATS_BO
: This IOCTL allows userspace to collect performance statistics on the most recent execution of a BO. This allows userspace to construct an end to end timeline of the BO processing for a performance analysis.

DRM_IOCTL_QAIC_PART_DEV
: This IOCTL allows userspace to request a duplicate "shadow device". This extra accelN device is associated with a specific partition of resources on the AIC100 device and can be used for limiting a process to some subset of resources.

2.1.4 Userspace Client Isolation

AIC100 supports multiple clients. Multiple DBCs can be consumed by a single client, and multiple clients can each consume one or more DBCs. Workloads may contain sensitive information therefore only the client that owns the workload should be allowed to interface with the DBC.

Clients are identified by the instance associated with their open(). A client may only use memory they allocate, and DBCs that are assigned to their workloads. Attempts to access resources assigned to other clients will be rejected.

2.1.5 Module parameters

QAIC supports the following module parameters:

datapath_polling (bool)

Configures QAIC to use a polling thread for datapath events instead of relying on the device interrupts. Useful for platforms with broken multiMSI. Must be set at QAIC driver initialization. Default is 0 (off).

mhi_timeout_ms (unsigned int)

Sets the timeout value for MHI operations in milliseconds (ms). Must be set at the time the driver detects a device. Default is 2000 (2 seconds).

control_resp_timeout_s (unsigned int)

Sets the timeout value for QSM responses to NNC messages in seconds (s). Must be set at the time the driver is sending a request to QSM. Default is 60 (one minute).

wait_exec_default_timeout_ms (unsigned int)

Sets the default timeout for the wait_exec ioctl in milliseconds (ms). Must be set prior to the waic_exec ioctl call. A value specified in the ioctl call overrides this for that call. Default is 5000 (5 seconds).

datapath_poll_interval_us (unsigned int)

Sets the polling interval in microseconds (us) when datapath polling is active. Takes effect at the next polling interval. Default is 100 (100 us).

2.2 Qualcomm Cloud AI 100 (AIC100)

2.2.1 Overview

The Qualcomm Cloud AI 100/AIC100 family of products (including SA9000P - part of Snapdragon Ride) are PCIe adapter cards which contain a dedicated SoC ASIC for the purpose of efficiently running Artificial Intelligence (AI) Deep Learning inference workloads. They are AI accelerators.

The PCIe interface of AIC100 is capable of PCIe Gen4 speeds over eight lanes (x8). An individual SoC on a card can have up to 16 NSPs for running workloads. Each SoC has an A53 management CPU. On card, there can be up to 32 GB of DDR.

Multiple AIC100 cards can be hosted in a single system to scale overall performance. AIC100 cards are multi-user capable and able to execute workloads from multiple users in a concurrent manner.

2.2.2 Hardware Description

An AIC100 card consists of an AIC100 SoC, on-card DDR, and a set of misc peripherals (PMICs, etc).

An AIC100 card can either be a PCIe HHHL form factor (a traditional PCIe card), or a Dual M.2 card. Both use PCIe to connect to the host system.

As a PCIe endpoint/adapter, AIC100 uses the standard VendorID(VID)/ DeviceID(DID) combination to uniquely identify itself to the host. AIC100 uses the standard Qualcomm VID (0x17cb). All AIC100 SKUs use the same AIC100 DID (0xa100).

AIC100 does not implement FLR (function level reset).

AIC100 implements MSI but does not implement MSI-X. AIC100 requires 17 MSIs to operate (1 for MHI, 16 for the DMA Bridge).

As a PCIe device, AIC100 utilizes BARs to provide host interfaces to the device hardware. AIC100 provides 3, 64-bit BARs.

- The first BAR is 4K in size, and exposes the MHI interface to the host.
- The second BAR is 2M in size, and exposes the DMA Bridge interface to the host.
- The third BAR is variable in size based on an individual AIC100's configuration, but defaults to 64K. This BAR currently has no purpose.

From the host perspective, AIC100 has several key hardware components -

- MHI (Modem Host Interface)

- QSM (QAIC Service Manager)
- NSPs (Neural Signal Processor)
- DMA Bridge
- DDR

MHI

AIC100 has one MHI interface over PCIe. MHI itself is documented at Documentation/mhi/index.rst MHI is the mechanism the host uses to communicate with the QSM. Except for workload data via the DMA Bridge, all interaction with the device occurs via MHI.

QSM

QAIC Service Manager. This is an ARM A53 CPU that runs the primary firmware of the card and performs on-card management tasks. It also communicates with the host via MHI. Each AIC100 has one of these.

NSP

Neural Signal Processor. Each AIC100 has up to 16 of these. These are the processors that run the workloads on AIC100. Each NSP is a Qualcomm Hexagon (Q6) DSP with HVX and HMX. Each NSP can only run one workload at a time, but multiple NSPs may be assigned to a single workload. Since each NSP can only run one workload, AIC100 is limited to 16 concurrent workloads. Workload “scheduling” is under the purview of the host. AIC100 does not automatically timeslice.

DMA Bridge

The DMA Bridge is custom DMA engine that manages the flow of data in and out of workloads. AIC100 has one of these. The DMA Bridge has 16 channels, each consisting of a set of request/response FIFOs. Each active workload is assigned a single DMA Bridge channel. The DMA Bridge exposes hardware registers to manage the FIFOs (head/tail pointers), but requires host memory to store the FIFOs.

DDR

AIC100 has on-card DDR. In total, an AIC100 can have up to 32 GB of DDR. This DDR is used to store workloads, data for the workloads, and is used by the QSM for managing the device. NSPs are granted access to sections of the DDR by the QSM. The host does not have direct access to the DDR, and must make requests to the QSM to transfer data to the DDR.

2.2.3 High-level Use Flow

AIC100 is a multi-user, programmable accelerator typically used for running neural networks in inferencing mode to efficiently perform AI operations. AIC100 is not intended for training neural networks. AIC100 can be utilized for generic compute workloads.

Assuming a user wants to utilize AIC100, they would follow these steps:

1. Compile the workload into an ELF targeting the NSP(s)
2. Make requests to the QSM to load the workload and related artifacts into the device DDR
3. Make a request to the QSM to activate the workload onto a set of idle NSPs
4. Make requests to the DMA Bridge to send input data to the workload to be processed, and other requests to receive processed output data from the workload.
5. Once the workload is no longer required, make a request to the QSM to deactivate the workload, thus putting the NSPs back into an idle state.
6. Once the workload and related artifacts are no longer needed for future sessions, make requests to the QSM to unload the data from DDR. This frees the DDR to be used by other users.

2.2.4 Boot Flow

AIC100 uses a flashless boot flow, derived from Qualcomm MSMs.

When AIC100 is first powered on, it begins executing PBL (Primary Bootloader) from ROM. PBL enumerates the PCIe link, and initializes the BHI (Boot Host Interface) component of MHI.

Using BHI, the host points PBL to the location of the SBL (Secondary Bootloader) image. The PBL pulls the image from the host, validates it, and begins execution of SBL.

SBL initializes MHI, and uses MHI to notify the host that the device has entered the SBL stage. SBL performs a number of operations:

- SBL initializes the majority of hardware (anything PBL left uninitialized), including DDR.
- SBL offloads the bootlog to the host.
- SBL synchronizes timestamps with the host for future logging.
- SBL uses the Sahara protocol to obtain the runtime firmware images from the host.

Once SBL has obtained and validated the runtime firmware, it brings the NSPs out of reset, and jumps into the QSM.

The QSM uses MHI to notify the host that the device has entered the QSM stage (AMSS in MHI terms). At this point, the AIC100 device is fully functional, and ready to process workloads.

2.2.5 Userspace components

Compiler

An open compiler for AIC100 based on upstream LLVM can be found at: https://github.com/quic/software-kit-for-qualcomm-cloud-ai-100-cc

Usermode Driver (UMD)

An open UMD that interfaces with the qaic kernel driver can be found at: https://github.com/quic/software-kit-for-qualcomm-cloud-ai-100

Sahara loader

An open implementation of the Sahara protocol called kickstart can be found at: https://github.com/andersson/qdl

2.2.6 MHI Channels

AIC100 defines a number of MHI channels for different purposes. This is a list of the defined channels, and their uses.

Channel name	IDs	EEs	Purpose
QAIC_LOOPBACK	0 & 1	AMSS	Any data sent to the device on this channel is sent back to the host.
QAIC_SAHARA	2 & 3	SBL	Used by SBL to obtain the runtime firmware from the host.
QAIC_DIAG	4 & 5	AMSS	Used to communicate with QSM via the DIAG protocol.
QAIC_SSR	6 & 7	AMSS	Used to notify the host of subsystem restart events, and to offload SSR crashdumps.
QAIC_QDSS	8 & 9	AMSS	Used for the Qualcomm Debug Subsystem.
QAIC_CONTROL	10 & 11	AMSS	Used for the Neural Network Control (NNC) protocol. This is the primary channel between host and QSM for managing workloads.
QAIC_LOGGING	12 & 13	SBL	Used by the SBL to send the bootlog to the host.
QAIC_STATUS	14 & 15	AMSS	Used to notify the host of Reliability, Accessibility, Serviceability (RAS) events.
QAIC_TELEMETRY	16 & 17	AMSS	Used to get/set power/thermal/etc attributes.
QAIC_DEBUG	18 & 19	AMSS	Not used.
QAIC_TIMESYNC	20 & 21	SBL/AMSS	Used to synchronize timestamps in the device side logs with the host time source.

2.2.7 DMA Bridge

Overview

The DMA Bridge is one of the main interfaces to the host from the device (the other being MHI). As part of activating a workload to run on NSPs, the QSM assigns that network a DMA Bridge channel. A workload's DMA Bridge channel (DBC for short) is solely for the use of that workload and is not shared with other workloads.

Each DBC is a pair of FIFOs that manage data in and out of the workload. One FIFO is the request FIFO. The other FIFO is the response FIFO.

Each DBC contains 4 registers in hardware:

- Request FIFO head pointer (offset 0x0). Read only by the host. Indicates the latest item in the FIFO the device has consumed.
- Request FIFO tail pointer (offset 0x4). Read/write by the host. Host increments this register to add new items to the FIFO.
- Response FIFO head pointer (offset 0x8). Read/write by the host. Indicates the latest item in the FIFO the host has consumed.
- Response FIFO tail pointer (offset 0xc). Read only by the host. Device increments this register to add new items to the FIFO.

The values in each register are indexes in the FIFO. To get the location of the FIFO element pointed to by the register: FIFO base address + register * element size.

DBC registers are exposed to the host via the second BAR. Each DBC consumes 4KB of space in the BAR.

The actual FIFOs are backed by host memory. When sending a request to the QSM to activate a network, the host must donate memory to be used for the FIFOs. Due to internal mapping limitations of the device, a single contiguous chunk of memory must be provided per DBC, which hosts both FIFOs. The request FIFO will consume the beginning of the memory chunk, and the response FIFO will consume the end of the memory chunk.

Request FIFO

A request FIFO element has the following structure:

```
struct request_elem {
      u16 req_id;
      u8  seq_id;
      u8  pcie_dma_cmd;
      u32 reserved;
      u64 pcie_dma_source_addr;
      u64 pcie_dma_dest_addr;
      u32 pcie_dma_len;
      u32 reserved;
      u64 doorbell_addr;
      u8  doorbell_attr;
      u8  reserved;
      u16 reserved;
```

```
      u32 doorbell_data;
      u32 sem_cmd0;
      u32 sem_cmd1;
      u32 sem_cmd2;
      u32 sem_cmd3;
};
```

Request field descriptions:

req_id
request ID. A request FIFO element and a response FIFO element with the same request ID refer to the same command.

seq_id
sequence ID within a request. Ignored by the DMA Bridge.

pcie_dma_cmd
describes the DMA element of this request.

- Bit(7) is the force msi flag, which overrides the DMA Bridge MSI logic and generates a MSI when this request is complete, and QSM configures the DMA Bridge to look at this bit.
- Bits(6:5) are reserved.
- Bit(4) is the completion code flag, and indicates that the DMA Bridge shall generate a response FIFO element when this request is complete.
- Bit(3) indicates if this request is a linked list transfer(0) or a bulk transfer(1).
- Bit(2) is reserved.
- Bits(1:0) indicate the type of transfer. No transfer(0), to device(1), from device(2). Value 3 is illegal.

pcie_dma_source_addr
source address for a bulk transfer, or the address of the linked list.

pcie_dma_dest_addr
destination address for a bulk transfer.

pcie_dma_len
length of the bulk transfer. Note that the size of this field limits transfers to 4G in size.

doorbell_addr
address of the doorbell to ring when this request is complete.

doorbell_attr
doorbell attributes.

- Bit(7) indicates if a write to a doorbell is to occur.
- Bits(6:2) are reserved.
- Bits(1:0) contain the encoding of the doorbell length. 0 is 32-bit, 1 is 16-bit, 2 is 8-bit, 3 is reserved. The doorbell address must be naturally aligned to the specified length.

doorbell_data
data to write to the doorbell. Only the bits corresponding to the doorbell length are valid.

sem_cmdN
semaphore command.

- Bit(31) indicates this semaphore command is enabled.
- Bit(30) is the to-device DMA fence. Block this request until all to-device DMA transfers are complete.
- Bit(29) is the from-device DMA fence. Block this request until all from-device DMA transfers are complete.
- Bits(28:27) are reserved.
- Bits(26:24) are the semaphore command. 0 is NOP. 1 is init with the specified value. 2 is increment. 3 is decrement. 4 is wait until the semaphore is equal to the specified value. 5 is wait until the semaphore is greater or equal to the specified value. 6 is "P", wait until semaphore is greater than 0, then decrement by 1. 7 is reserved.
- Bit(23) is reserved.
- Bit(22) is the semaphore sync. 0 is post sync, which means that the semaphore operation is done after the DMA transfer. 1 is presync, which gates the DMA transfer. Only one presync is allowed per request.
- Bit(21) is reserved.
- Bits(20:16) is the index of the semaphore to operate on.
- Bits(15:12) are reserved.
- Bits(11:0) are the semaphore value to use in operations.

Overall, a request is processed in 4 steps:

1. If specified, the presync semaphore condition must be true
2. If enabled, the DMA transfer occurs
3. If specified, the postsync semaphore conditions must be true
4. If enabled, the doorbell is written

By using the semaphores in conjunction with the workload running on the NSPs, the data pipeline can be synchronized such that the host can queue multiple requests of data for the workload to process, but the DMA Bridge will only copy the data into the memory of the workload when the workload is ready to process the next input.

Response FIFO

Once a request is fully processed, a response FIFO element is generated if specified in pcie_dma_cmd. The structure of a response FIFO element:

```
struct response_elem {
      u16 req_id;
      u16 completion_code;
};
```

req_id
matches the req_id of the request that generated this element.

completion_code
: status of this request. 0 is success. Non-zero is an error.

The DMA Bridge will generate a MSI to the host as a reaction to activity in the response FIFO of a DBC. The DMA Bridge hardware has an IRQ storm mitigation algorithm, where it will only generate a MSI when the response FIFO transitions from empty to non-empty (unless force MSI is enabled and triggered). In response to this MSI, the host is expected to drain the response FIFO, and must take care to handle any race conditions between draining the FIFO, and the device inserting elements into the FIFO.

2.2.8 Neural Network Control (NNC) Protocol

The NNC protocol is how the host makes requests to the QSM to manage workloads. It uses the QAIC_CONTROL MHI channel.

Each NNC request is packaged into a message. Each message is a series of transactions. A passthrough type transaction can contain elements known as commands.

QSM requires NNC messages be little endian encoded and the fields be naturally aligned. Since there are 64-bit elements in some NNC messages, 64-bit alignment must be maintained.

A message contains a header and then a series of transactions. A message may be at most 4K in size from QSM to the host. From the host to the QSM, a message can be at most 64K (maximum size of a single MHI packet), but there is a continuation feature where message N+1 can be marked as a continuation of message N. This is used for exceedingly large DMA xfer transactions.

Transaction descriptions

passthrough
: Allows userspace to send an opaque payload directly to the QSM. This is used for NNC commands. Userspace is responsible for managing the QSM message requirements in the payload.

dma_xfer
: DMA transfer. Describes an object that the QSM should DMA into the device via address and size tuples.

activate
: Activate a workload onto NSPs. The host must provide memory to be used by the DBC.

deactivate
: Deactivate an active workload and return the NSPs to idle.

status
: Query the QSM about it's NNC implementation. Returns the NNC version, and if CRC is used.

terminate
: Release a user's resources.

dma_xfer_cont
: Continuation of a previous DMA transfer. If a DMA transfer cannot be specified in a single message (highly fragmented), this transaction can be used to specify more ranges.

validate_partition
Query to QSM to determine if a partition identifier is valid.

Each message is tagged with a user id, and a partition id. The user id allows QSM to track resources, and release them when the user goes away (eg the process crashes). A partition id identifies the resource partition that QSM manages, which this message applies to.

Messages may have CRCs. Messages should have CRCs applied until the QSM reports via the status transaction that CRCs are not needed. The QSM on the SA9000P requires CRCs for black channel safing.

2.2.9 Subsystem Restart (SSR)

SSR is the concept of limiting the impact of an error. An AIC100 device may have multiple users, each with their own workload running. If the workload of one user crashes, the fallout of that should be limited to that workload and not impact other workloads. SSR accomplishes this.

If a particular workload crashes, QSM notifies the host via the QAIC_SSR MHI channel. This notification identifies the workload by it's assigned DBC. A multi-stage recovery process is then used to cleanup both sides, and get the DBC/NSPs into a working state.

When SSR occurs, any state in the workload is lost. Any inputs that were in process, or queued by not yet serviced, are lost. The loaded artifacts will remain in on-card DDR, but the host will need to re-activate the workload if it desires to recover the workload.

2.2.10 Reliability, Accessibility, Serviceability (RAS)

AIC100 is expected to be deployed in server systems where RAS ideology is applied. Simply put, RAS is the concept of detecting, classifying, and reporting errors. While PCIe has AER (Advanced Error Reporting) which factors into RAS, AER does not allow for a device to report details about internal errors. Therefore, AIC100 implements a custom RAS mechanism. When a RAS event occurs, QSM will report the event with appropriate details via the QAIC_STATUS MHI channel. A sysadmin may determine that a particular device needs additional service based on RAS reports.

2.2.11 Telemetry

QSM has the ability to report various physical attributes of the device, and in some cases, to allow the host to control them. Examples include thermal limits, thermal readings, and power readings. These items are communicated via the QAIC_TELEMETRY MHI channel.