Linux Mhi Documentation

The kernel development community

Jun 10, 2024

CONTENTS

CHAPTER

ONE

MHI (MODEM HOST INTERFACE)

This document provides information about the MHI protocol.

1.1 Overview

MHI is a protocol developed by Qualcomm Innovation Center, Inc. It is used by the host processors to control and communicate with modem devices over high speed peripheral buses or shared memory. Even though MHI can be easily adapted to any peripheral buses, it is primarily used with PCIe based devices. MHI provides logical channels over the physical buses and allows transporting the modem protocols, such as IP data packets, modem control messages, and diagnostics over at least one of those logical channels. Also, the MHI protocol provides data acknowledgment feature and manages the power state of the modems via one or more logical channels.

1.2 MHI Internals

1.2.1 MMIO

MMIO (Memory mapped IO) consists of a set of registers in the device hardware, which are mapped to the host memory space by the peripheral buses like PCIe. Following are the major components of MMIO register space:

MHI control registers: Access to MHI configurations registers

MHI BHI registers: BHI (Boot Host Interface) registers are used by the host for downloading the firmware to the device before MHI initialization.

Channel Doorbell array: Channel Doorbell (DB) registers used by the host to notify the device when there is new work to do.

Event Doorbell array: Associated with event context array, the Event Doorbell (DB) registers are used by the host to notify the device when new events are available.

Debug registers: A set of registers and counters used by the device to expose debugging information like performance, functional, and stability to the host.

1.2.2 Data structures

All data structures used by MHI are in the host system memory. Using the physical interface, the device accesses those data structures. MHI data structures and data buffers in the host system memory regions are mapped for the device.

Channel context array: All channel configurations are organized in channel context data array.

Transfer rings: Used by the host to schedule work items for a channel. The transfer rings are organized as a circular queue of Transfer Descriptors (TD).

Event context array: All event configurations are organized in the event context data array.

Event rings: Used by the device to send completion and state transition messages to the host

Command context array: All command configurations are organized in command context data array.

Command rings: Used by the host to send MHI commands to the device. The command rings are organized as a circular queue of Command Descriptors (CD).

1.2.3 Channels

MHI channels are logical, unidirectional data pipes between a host and a device. The concept of channels in MHI is similar to endpoints in USB. MHI supports up to 256 channels. However, specific device implementations may support less than the maximum number of channels allowed.

Two unidirectional channels with their associated transfer rings form a bidirectional data pipe, which can be used by the upper-layer protocols to transport application data packets (such as IP packets, modem control messages, diagnostics messages, and so on). Each channel is associated with a single transfer ring.

1.2.4 Transfer rings

Transfers between the host and device are organized by channels and defined by Transfer Descriptors (TD). TDs are managed through transfer rings, which are defined for each channel between the device and host and reside in the host memory. TDs consist of one or more ring elements (or transfer blocks):

```
[Read Pointer (RP)] ---------->[Ring Element] } TD
[Write Pointer (WP)]-          [Ring Element]
                     -         [Ring Element]
                      --------->[Ring Element]
                                [Ring Element]
```

Below is the basic usage of transfer rings:

- Host allocates memory for transfer ring.
- Host sets the base pointer, read pointer, and write pointer in corresponding channel context.
- Ring is considered empty when RP == WP.
- Ring is considered full when WP + 1 == RP.

- RP indicates the next element to be serviced by the device.
- When the host has a new buffer to send, it updates the ring element with buffer information, increments the WP to the next element and rings the associated channel DB.

1.2.5 Event rings

Events from the device to host are organized in event rings and defined by Event Descriptors (ED). Event rings are used by the device to report events such as data transfer completion status, command completion status, and state changes to the host. Event rings are the array of EDs that resides in the host memory. EDs consist of one or more ring elements (or transfer blocks):

```
[Read Pointer (RP)] ---------->[Ring Element] } ED
[Write Pointer (WP)]-          [Ring Element]
                     -         [Ring Element]
                      --------->[Ring Element]
                                [Ring Element]
```

Below is the basic usage of event rings:

- Host allocates memory for event ring.
- Host sets the base pointer, read pointer, and write pointer in corresponding channel context.
- Both host and device has a local copy of RP, WP.
- Ring is considered empty (no events to service) when WP + 1 == RP.
- Ring is considered full of events when RP == WP.
- When there is a new event the device needs to send, the device updates ED pointed by RP, increments the RP to the next element and triggers the interrupt.

1.2.6 Ring Element

A Ring Element is a data structure used to transfer a single block of data between the host and the device. Transfer ring element types contain a single buffer pointer, the size of the buffer, and additional control information. Other ring element types may only contain control and status information. For single buffer operations, a ring descriptor is composed of a single element. For large multi-buffer operations (such as scatter and gather), elements can be chained to form a longer descriptor.

1.3 MHI Operations

1.3.1 MHI States

MHI_STATE_RESET

MHI is in reset state after power-up or hardware reset. The host is not allowed to access device MMIO register space.

MHI_STATE_READY

MHI is ready for initialization. The host can start MHI initialization by programming MMIO registers.

MHI_STATE_M0

MHI is running and operational in the device. The host can start channels by issuing channel start command.

MHI_STATE_M1

MHI operation is suspended by the device. This state is entered when the device detects inactivity at the physical interface within a preset time.

MHI_STATE_M2

MHI is in low power state. MHI operation is suspended and the device may enter lower power mode.

MHI_STATE_M3

MHI operation stopped by the host. This state is entered when the host suspends MHI operation.

1.3.2 MHI Initialization

After system boots, the device is enumerated over the physical interface. In the case of PCIe, the device is enumerated and assigned BAR-0 for the device's MMIO register space. To initialize the MHI in a device, the host performs the following operations:

- Allocates the MHI context for event, channel and command arrays.
- Initializes the context array, and prepares interrupts.
- Waits until the device enters READY state.
- Programs MHI MMIO registers and sets device into MHI_M0 state.
- Waits for the device to enter M0 state.

1.3.3 MHI Data Transfer

MHI data transfer is initiated by the host to transfer data to the device. Following are the sequence of operations performed by the host to transfer data to device:

- Host prepares TD with buffer information.
- Host increments the WP of the corresponding channel transfer ring.
- Host rings the channel DB register.
- Device wakes up to process the TD.
- Device generates a completion event for the processed TD by updating ED.
- Device increments the RP of the corresponding event ring.
- Device triggers IRQ to wake up the host.
- Host wakes up and checks the event ring for completion event.
- Host updates the WP of the corresponding event ring to indicate that the data transfer has been completed successfully.

CHAPTER

TWO

MHI TOPOLOGY

This document provides information about the MHI topology modeling and representation in the kernel.

2.1 MHI Controller

MHI controller driver manages the interaction with the MHI client devices such as the external modems and WiFi chipsets. It is also the MHI bus master which is in charge of managing the physical link between the host and device. It is however not involved in the actual data transfer as the data transfer is taken care by the physical bus such as PCIe. Each controller driver exposes channels and events based on the client device type.

Below are the roles of the MHI controller driver:

- Turns on the physical bus and establishes the link to the device
- Configures IRQs, IOMMU, and IOMEM
- Allocates struct mhi_controller and registers with the MHI bus framework with channel and event configurations using mhi_register_controller.
- Initiates power on and shutdown sequence
- Initiates suspend and resume power management operations of the device.

2.2 MHI Device

MHI device is the logical device which binds to a maximum of two MHI channels for bi-directional communication. Once MHI is in powered on state, the MHI core will create MHI devices based on the channel configuration exposed by the controller. There can be a single MHI device for each channel or for a couple of channels.

Each supported device is enumerated in:

```
/sys/bus/mhi/devices/
```

2.3 MHI Driver

MHI driver is the client driver which binds to one or more MHI devices. The MHI driver sends and receives the upper-layer protocol packets like IP packets, modem control messages, and diagnostics messages over MHI. The MHI core will bind the MHI devices to the MHI driver.

Each supported driver is enumerated in:

```
/sys/bus/mhi/drivers/
```

Below are the roles of the MHI driver:

- Registers the driver with the MHI bus framework using mhi_driver_register.
- Prepares the device for transfer by calling mhi_prepare_for_transfer.
- Initiates data transfer by calling mhi_queue_transfer.
- Once the data transfer is finished, calls mhi_unprepare_from_transfer to end data transfer.