I2C Multi-controller and Controller Target Mode in QEMU

Klaus Jensen  <k.jensen@samsung.com>
Samsung Electronics
I’m an **NVMe** proliferator, why care about **I2C**?

- **Because of the NVMe Management Interface**
  - Provides an **out-of-band** interface to manage NVMe devices and enclosures
    - Uses the **Management Component Transport Protocol (MCTP)**
  - The NVMe-MI specification supports MCTP via **SMBus/I2C** or **PCIe VDM** as the out-of-band communication paths
    - This talk is about emulating the SMBus/I2C MCTP binding
Goals

1. Launch an emulated BMC platform with an NVMe-MI device on an I2C bus
   - due to MCTP, this requires the target device to be able to act as a controller on the bus as well (more on that in a bit)

2. Communicate with that device using the Linux kernel MCTP framework (the AF_MCTP socket)
   - relies on i2c controller target mode support (more on that later)
The Management Component Transport Protocol

- MCTP is a transport independent protocol
- The unit of data transfer is the **Packet**
  - One or more packets may be assembled into **Messages**
- Messages are exchanged between **Endpoints**
MCTP Packet

- An MCTP packet consists of:
  1. A physical medium specific header and trailer
  2. MCPT base specification defined packet header
  3. Semi-opaque packet payload to be interpreted by an endpoint

- The MCTP packet header assists with:
  - Identifying the MCTP protocol version
  - Packet routing (source and destination endpoint identifiers)
  - Message assembly (e.g., sequence number)
Message Assembly

- Medium Header
- MCTP Packet Header
- MCTP Packet Payload
- Medium Trailer

Message header only present in the first packet payload

Message Integrity check (if used) is only present in the final packet
I²C Basics

- I²C (“Inter-Integrated Circuit”)
- Synchronous, controller/target, two-line, serial bus
  - Controller controls the Serial Clock (SCL) line
  - The Serial Data (SDA) line is used by both parties depending on the transaction
I2C Basics

• Message-oriented
  – Controller addresses a specific target by transmitting the address on the bus
    • If the target is on the bus, it will ACK by pulling the SDA line
  – Data is transmitted in frames of 8 bits, each ACK’ed by the receiving party

<table>
<thead>
<tr>
<th>START</th>
<th>ADDRESS (7 bits)</th>
<th>R/W (1 bit)</th>
<th>(N) ACK (1 bit)</th>
<th>DATA (8 bits)</th>
<th>(N) ACK (1 bit)</th>
<th>DATA (8 bits)</th>
<th>(N) ACK (1 bit)</th>
<th>STOP</th>
</tr>
</thead>
</table>
In the MCTP SMBus/I2C binding, all transactions are based on the SMBus “Block Write” bus protocol.

<table>
<thead>
<tr>
<th>START</th>
<th>ADDRESS (7 bits)</th>
<th>R/W (1 bit)</th>
<th>(N) ACK (1 bit)</th>
<th>COMMAND (8 bits)</th>
<th>(N) ACK (1 bit)</th>
<th>BYTE COUNT (8 bits)</th>
<th>(N) ACK (1 bit)</th>
<th>DATA (8 bits)</th>
<th>(N) ACK (1 bit)</th>
<th>DATA (8 bits)</th>
<th>(N) ACK (1 bit)</th>
<th>PEC (8 bits)</th>
<th>(N) ACK (1 bit)</th>
<th>STOP</th>
</tr>
</thead>
</table>

**SMBus Block Write**
In the MCTP SMBus/I2C binding, all transactions are based on the SMBus “Block Write” bus protocol. The packet error code (CRC-8) is used to verify the integrity of the data. The number of bytes following 0xF (i.e., “MCTP”) indicates the length of the block write transaction.
Since MCTP uses Block Writes exclusively, the I2C binding modifies the protocol slightly to include the source address as the first data byte.

<table>
<thead>
<tr>
<th>START</th>
<th>ADDRESS (7 bits)</th>
<th>R/W (1 bit)</th>
<th>(N) ACK (1 bit)</th>
<th>COMMAND (8 bits)</th>
<th>(N) ACK (1 bit)</th>
<th>BYTE COUNT (8 bits)</th>
<th>(N) ACK (1 bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOURCE (8 bits)</td>
<td>(N) ACK (1 bit)</td>
<td>DATA (8 bits)</td>
<td>(N) ACK (1 bit)</td>
<td>PEC (8 bits)</td>
<td>(N) ACK (1 bit)</td>
<td>STOP</td>
<td></td>
</tr>
</tbody>
</table>
Basic I2C Emulation in QEMU

- In QEMU, we model this with the `I2CBus`
  - The I2C controller device model owns the bus and targets are children on it
    - Controllers sends START and STOP conditions
      - `int i2c_start_send(I2CBus *bus, uint8_t addr);`
      - `void i2c_end_transfer(I2CBus *bus);`
    - Controller may act as transmitter or receiver
      - `int i2c_send(I2CBus *bus, uint8_t data);`
      - `uint8_t i2c_recv(I2CBus *bus);`
Basic I2C Emulation in QEMU

• In QEMU, we model this with the `I2CBus`
  – The I2C controller device model owns the bus and targets are children on it
  • Controllers sends START and
    – `int i2c_start_send(I2CBus *bus, uint8_t addr);`
    – `void i2c_end_transfer(I2CBus *bus);`
  • Controller may act as transmitter or receiver
    – `int i2c_send(I2CBus *bus, uint8_t data);`
    – `uint8_t i2c_recv(I2CBus *bus);`

(N)ACK indicated by return value
Basic I2C Emulation in QEMU

- Bus targets implement the I2CTargetClass
  - `send()`, `recv()` callbacks
    - called from `i2c_send()`, `i2c_recv()`
  - an `event()` callback
    - in response to `i2c_start_transfer()`, `i2c_end_transfer()`
Single-controller Transactions

i2c_start_transfer(bus, addr)
  target->event(START_SEND)
i2c_send(bus, data)
  target->send(data)
i2c_send(bus, data)
...
i2c_end_transfer(bus)
  target->event(FINISH)
How does the target “reply”? 

- MCTP uses controller-transmits exclusively
  - How can the target device reply? (i.e. act as a controller)
  - **Spoiler Alert** -
How does the target “reply”?  

- MCTP uses controller-transmits exclusively
  - How can the target device reply? (i.e. act as a controller)
  - Spoiler Alert - it can’t

- The I2C core does not support multiple controllers
  - However, nothing *explicitly* prevents a target device from getting a reference to the bus and calling `i2c_start_transfer()`
    - … it’s just a little wonky
Multi-controller Transactions

A small “band aid” in \texttt{i2c\_end\_transfer()} can actually make this work. \textit{Sort of.}

\begin{itemize}
  \item \texttt{i2c\_start\_transfer}(\texttt{bus, addr})
  \item target->event(START\_SEND)
  \item \texttt{i2c\_send}(\texttt{bus, data})
  \item \texttt{i2c\_send}(\texttt{bus, data})
  \item \textit{...}
  \item \texttt{i2c\_end\_transfer}(\texttt{bus})
  \item target->event(FINISH)
  \item bus = qdev\_get\_parent\_bus()
  \item \texttt{i2c\_start\_transfer}(\texttt{bus, addr})
  \item target->event(START\_SEND)
  \item \texttt{i2c\_send}(\texttt{bus, data})
  \item \texttt{i2c\_send}(\texttt{bus, data})
  \item \textit{...}
  \item \texttt{i2c\_end\_transfer}(\texttt{bus})
  \item target->event(FINISH)
  \item \texttt{i2c\_start\_transfer}(\texttt{bus, addr})
  \item \textit{...}
\end{itemize}
When in doubt - defer the problem...

• What does hardware do? **Arbitration**
  – Just talk until I’m the only one talking
  – Give up after a while and try again.

• In QEMU, we can have the controllers queue up nicely in a line instead
  – Register a callback (a bottom half) to be called when it is my turn
Multi-controller Transactions

**Solution:** register and schedule a bottom half

```c
i2c_start_transfer(bus, addr)
    target->event(START_SEND)
i2c_send(bus, data)
i2c_send(bus, data)
...
i2c_end_transfer(bus)
    target->event(FINISH)
    i2c_bus_acquire(bus, bh)
    qemu_bh_schedule(bh)

bh
i2c_start_transfer(bus, addr)
    target->event(START_SEND)
i2c_send(bus, data)
i2c_send(bus, data)
...
i2c_end_transfer(bus)
i2c_bus_release(bus)
```
Multi-controller Transactions

**Solution**: register and schedule a bottom half

```c
i2c_start_transfer(bus, addr)
    target->event(START_SEND)
i2c_send(bus, data)
i2c_send(bus, data)
...
i2c_end_transfer(bus)
    target->event(FINISH)
    i2c_bus_acquire(bus, bh)
    qemu_bh_schedule(bh)

bh
    i2c_start_transfer(bus, addr)
        target->event(START_SEND)
i2c_send(bus, data)
i2c_send(bus, data)
...
i2c_end_transfer(bus)
i2c_bus_release(bus)
```

The “bus owner” remains the **default** Controller when no bus child has acquired the bus.
Not the Full Story

- Something must drive the “requesting” controller?
  - E.g. the Aspeed I2C host bus driver
Host-driven Single-controller Transactions

- **Host** - i2c bus driver
- **i2c controller**
- **QEMU**
- **i2c target**

**Code Snippet**

```c
aspeed_i2c_bus_write()
  aspeed_i2c_bus_handle_cmd()
  i2c_start_transfer()
  aspeed_i2c_bus_raise_interrupt()

aspeed_i2c_bus_write()
  aspeed_i2c_bus_handle_cmd()
  aspeed_i2c_bus_send()
  i2c_send()
  aspeed_i2c_bus_raise_interrupt()

aspeed_i2c_bus_write()
  aspeed_i2c_bus_handle_cmd()
  i2c_end_transfer()
  aspeed_i2c_bus_raise_interrupt()
```
Host-driven Multi-controller Transactions

```
host i2c bus driver
    ↓
i2c controller

i2c controller
    ← i2c_send()

QEMU

aspeed_i2c_bus_write()
aspeed_i2c_bus_handle_cmd()
i2c_end_transfer()
target->event(FINISH)
i2c_bus_acquire(bus, bh)
qemu_bh_schedule(bh)
aspeed_i2c_bus_raise_interrupt()

bh
i2c_start_transfer()
i2c_send()
aspeed_i2c_bus_target_send()
aspeed_i2c_bus_raise_interrupt()
i2c_send()
aspeed_i2c_bus_target_send()
aspeed_i2c_bus_raise_interrupt()

... ...
i2c_end_transfer()
```
Host-driven Multi-controller Transactions

Where does the data go?

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controller models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?

Overwrites previously sent byte if controller doesn’t buffer it

The I2C Controllers models does not implement I2CTargetClass.

Who is the target of this?
How can the host read the reply?

• I2C controllers/drivers may support “Target Mode”
  – Allows the host driver to
    • read data sent to the controller when acting as a target-receiver, or to
  – Spoiler Alert -
How can the host read the reply?

- I2C controllers/drivers may support “Target Mode”
  - Allows the host driver to
    - read data sent to the controller when acting as a target-receiver, or to
  - **Spoiler Alert** - no controllers in QEMU supports this
    - Oh, and, by the way, it breaks a fundamental rule in the i2c core
      - **All transfers must complete immediately**
1st Problem

• Fix “All transfer must complete immediately”
  – Introduce `i2c_send_async()` and `START_ASYNC` event
    • Reuse the bottom half and turn it into a state machine
    • Add an explicit `i2c_ack()` to schedule the bottom half

  – Does not impact existing device models (they just need to NACK the `START_ASYNC` event)

  – Asynchronous device models (like board I2C controllers) can choose not to implement the synchronous `i2c_send()`
Add Target Mode in the Aspeed I2C Controller

- **Why the Aspeed?** Because the Linux kernel driver supports target mode for that, and
- in QEMU, the Aspeed I2C controller model authors left it as a “fill in the blanks” exercise

- Surprisingly straight-forward given the kernel driver code

```plaintext
hw/i2c/aspeed_i2c.c    |  89 +++++++++--
include/hw/i2c/aspeed_i2c.h |  8 +
2 files changed, 88 insertions(+), 9 deletions(-)
```
2nd Problem

- **Add an** AspeedI2CBusTarget **as a target on the** AspeedI2CBus **model**

- **Implement the target callbacks**
  - `event()`
    - On `START_SEND_ASYNC`, set relevant interrupt bits
    - On `FINISH`, set `NORMAL_STOP` interrupt bit
  - `send_async()`
    - Copy byte to `BYTE_BUF` register and set `RX_DONE` interrupt bit

- **Call** `i2c_ack()` **when interrupt acknowledged**
Host-driven Multi-controller Transactions

host i2c bus driver

i2c-set

i2c controller

i2c_send()

i2c target

QEMU

aspeed_i2c_bus_write()
aspeed_i2c_bus_handle_cmd()
  i2c_end_transfer()
    target->event(FINISH)
    i2c_bus_acquire(bus, bh)
    qemu_bh_schedule(bh)
    aspeed_i2c_bus_raise_interrupt()
Host-driven Multi-controller Transactions

- **host i2c bus driver**
  - `i2c-set`
  - `i2c target`
  - `i2c_send_async()`
  - `i2c_ack()`
  - **QEMU**
    - `aspeed_i2c_bus_write()`
    - `aspeed_i2c_bus_handle_cmd()`
    - `i2c_end_transfer()`
    - `target->event(FINISH)`
    - `i2c_bus_acquire(bus, bh)`
    - `qemu_bh_schedule(bh)`
    - `aspeed_i2c_bus_release(bus)`

- **i2c controller**
  - `bh`
    - `i2c_start_send_async(bus, addr)`
    - `aspeed_i2c_bus_slave_event()`
    - `aspeed_i2c_bus_raise_interrupt(SLAVE_ADDR_RX_MATCH)`
    - `aspeed_i2c_bus_write()`
    - `i2c_ack(bus)`
    - `qemu_bh_schedule(bh)`
    - `bh`
      - `i2c_send_async(bus, byte)`
      - `aspeed_i2c_bus_slave_send_async()`
      - `aspeed_i2c_bus_raise_interrupt(RX_DONE)`
      - `aspeed_i2c_bus_write()`
      - `i2c_ack(bus)`
      - `qemu_bh_schedule(bh)`
      - `bh`
        - `i2c_end_transfer(bus)`
        - `aspeed_i2c_bus_slave_event()`
        - `aspeed_i2c_bus_raise_interrupt(STOP)`
        - `i2c_bus_release(bus)`
Testing with a kernel target device

- Linux includes an EEPROM target device
  - Create the target device on the bus
    ```
    # echo slave-24c02 0x1064 > /sys/bus/i2c/devices/i2c-15/new_device
    i2c i2c-15: new_device: Instantiated device slave-24c02 at 0x64
    ```
    - The device at 0x42 is a toy “echo” device that will write back to an EEPROM what it is instructed to
      - Write 0xAA at offset 0x00 to the EEPROM device on 0x64
        ```
        # i2cset -y 15 0x42 0x64 0x00 0xaa i
        # hexdump /sys/bus/i2c/devices/15-1064/slave-eeprom
        0000000 ffaa ffff ffff ffff ffff ffff ffff ffff ffff ffff ffff ffff ffff ffff ffff ffff
        0000010 ffff ffff ffff ffff ffff ffff ffff ffff ffff
        *
        0000100
        ```
Putting it to work with MCTP

- Add an abstract MCTP I2C Target device model
  - Handles the I2C encapsulation and message assembly/disassembly
  - Handles MCTP control messages
  - Implements the `send()` and `event()` callbacks
    - `send()` soaks up the I2C message
    - `event()` verifies the message and delivers it to the deriving device model on FINISH
      - The deriving device calls `i2c_mctp_schedule_send()` when it has processed the MCTP message
      - The bottom half uses `i2c_send_async()` to send the response to the kernel MCTP target device.
Putting it to work with MCTP

- Modify a device tree for an Aspeed 2600 EVB

```c
#include "aspeed-ast2600-evb.dts"
#include <dt-bindings/i2c/i2c.h>

&i2c15 {
    multi-master;
    mctp-controller;

    mctp@10 {
        compatible = "mctp-i2c-controller";
        reg = <(0x10 | I2C_OWN_SLAVE_ADDRESS)>;
    }
};
```

Tells the kernel mctp-i2c driver to configure target mode on address 0x10

External Buildroot to get up and running quickly:
https://irrelevant.dk/g/buildroots.git/tree/mctp-i2c
Putting it to work with MCTP

qemu-system-arm \
  -nographic \
  -M ast2600-evb \
  -kernel output/images/zImage \
  -initrd output/images/rootfs.cpio \
  -dtb output/images/aspeed-ast2600-evb-nmi.dtb \
  -device nmi-i2c,address=0x3a,eid=0x9 \
  -serial mon:stdio

[ 1.341308] mctp: management component transport protocol core
[ 2.814937] i2c-core: driver [mctp-i2c-interface] registered
[ 3.156248] i2c i2c-15: of_i2c: register /ahb/apb/bus@1e78a000/i2c-bus@800/mctp@10
Putting it to work with MCTP

qemu-system-arm
   -nographic
   -M ast2600-evb
   -kernel output/images/zImage
   -initrd output/images/rootfs.cpio
   -dtb output/images/aspeed-ast2600-evb-nmi.dtb
   -device nmi-i2c,address=0x3a,eid=0x9
   -serial mon:stdio

[ 1.341308] mctp: management component transport protocol
[ 2.814937] i2c-core: driver [mctp-i2c-interface] registered
[ 3.156248] i2c i2c-15: of_i2c: register /ahb/apb/bus@1e78a000/i2c-bus@800/mctp@10

# mctp addr add 8 dev mctpi2c15
# mctp link set mctpi2c15 up
# mctp route add 9 via mctpi2c15
# mctp neigh add 9 dev mctpi2c15 lladdr 0x3a
# mi-mctp 1 9 info
NVMe MI subsys info:
   num ports: 1
   major ver: 1
   minor ver: 1
NVMe MI port info:
   port 0
      type SMBus[2]
      MCTP MTU: 64
      MEB size: 0
      SMBus address: 0x00
      VPD access freq: 0x00
      MCTP address: 0x3a
      MCTP access freq: 0x01
      NVMe basic management: disabled