NTB Drivers

NTB (Non-Transparent Bridge) is a type of PCI-Express bridge chip that connects the separate memory systems of two or more computers to the same PCI-Express fabric. Existing NTB hardware supports a common feature set: doorbell registers and memory translation windows, as well as non common features like scratchpad and message registers. Scratchpad registers are read-and-writable registers that are accessible from either side of the device, so that peers can exchange a small amount of information at a fixed address. Message registers can be utilized for the same purpose. Additionally they are provided with special status bits to make sure the information isn't rewritten by another peer. Doorbell registers provide a way for peers to send interrupt events. Memory windows allow translated read and write access to the peer memory.

NTB Core Driver (ntb)

The NTB core driver defines an api wrapping the common feature set, and allows clients interested in NTB features to discover NTB the devices supported by hardware drivers. The term "client" is used here to mean an upper layer component making use of the NTB api. The term "driver," or "hardware driver," is used here to mean a driver for a specific vendor and model of NTB hardware.

NTB Client Drivers

NTB client drivers should register with the NTB core driver. After registering, the client probe and remove functions will be called appropriately as ntb hardware, or hardware drivers, are inserted and removed. The registration uses the Linux Device framework, so it should feel familiar to anyone who has written a pci driver.

NTB Typical client driver implementation

Primary purpose of NTB is to share some peace of memory between at least two systems. So the NTB device features like Scratchpad/Message registers are mainly used to perform the proper memory window initialization. Typically there are two types of memory window interfaces supported by the NTB API: inbound translation configured on the local ntb port and outbound translation configured by the peer, on the peer ntb port. The first type is depicted on the next figure:

```
Inbound translation:

Memory:              Local NTB Port:      Peer NTB Port:      Peer MMIO:
 ____________
| dma-mapped |-ntb_mw_set_trans(addr)  |
| memory     |        _v____________   |   ______________
| (addr)     |<======| MW xlat addr |<====| MW base addr |<== memory-mapped IO
|------------|       |--------------|  |  |--------------|
```

So typical scenario of the first type memory window initialization looks: 1) allocate a memory region, 2) put translated address to NTB config, 3) somehow notify a peer device of performed initialization, 4) peer device maps corresponding outbound memory window so to have access to the shared memory region.

The second type of interface, that implies the shared windows being initialized by a peer device, is depicted on the figure:

```
Outbound translation:

Memory:        Local NTB Port:    Peer NTB Port:      Peer MMIO:
 ____________                      ______________
| dma-mapped |                |   | MW base addr |<== memory-mapped IO
| memory     |                |   |--------------|
| (addr)     |<===================| MW xlat addr |<-ntb_peer_mw_set_trans(addr)
|------------|                |   |--------------|
```

Typical scenario of the second type interface initialization would be: 1) allocate a memory region, 2) somehow deliver a translated address to a peer device, 3) peer puts the translated address to NTB config, 4) peer device maps outbound memory window so to have access to the shared memory region.

As one can see the described scenarios can be combined in one portable algorithm.

Local device:
1. Allocate memory for a shared window
2. Initialize memory window by translated address of the allocated region (it may fail if local memory window initialization is unsupported)
3. Send the translated address and memory window index to a peer device

Peer device:
1. Initialize memory window with retrieved address of the allocated by another device memory region (it may fail if peer memory window initialization is unsupported)
2. Map outbound memory window

In accordance with this scenario, the NTB Memory Window API can be used as follows:

Local device:

1. ntb_mw_count(pidx) - retrieve number of memory ranges, which can be allocated for memory windows between local device and peer device of port with specified index.
2. ntb_get_align(pidx, midx) - retrieve parameters restricting the shared memory region alignment and size. Then memory can be properly allocated.
3. Allocate physically contiguous memory region in compliance with restrictions retrieved in 2).
4. ntb_mw_set_trans(pidx, midx) - try to set translation address of the memory window with specified index for the defined peer device (it may fail if local translated address setting is not supported)
5. Send translated base address (usually together with memory window number) to the peer device using, for instance, scratchpad or message registers.

Peer device:

1. ntb_peer_mw_set_trans(pidx, midx) - try to set received from other device (related to pidx) translated address for specified memory window. It may fail if retrieved address, for instance, exceeds maximum possible address or isn't properly aligned.
2. ntb_peer_mw_get_addr(widx) - retrieve MMIO address to map the memory window so to have an access to the shared memory.

Also it is worth to note, that method ntb_mw_count(pidx) should return the same value as ntb_peer_mw_count() on the peer with port index - pidx.

NTB Transport Client (ntb_transport) and NTB Netdev (ntb_netdev)

The primary client for NTB is the Transport client, used in tandem with NTB Netdev. These drivers function together to create a logical link to the peer, across the ntb, to exchange packets of network data. The Transport client establishes a logical link to the peer, and creates queue pairs to exchange messages and data. The NTB Netdev then creates an ethernet device using a Transport queue pair. Network data is copied between socket buffers and the Transport queue pair buffer. The Transport client may be used for other things besides Netdev, however no other applications have yet been written.

NTB Ping Pong Test Client (ntb_pingpong)

The Ping Pong test client serves as a demonstration to exercise the doorbell and scratchpad registers of NTB hardware, and as an example simple NTB client. Ping Pong enables the link when started, waits for the NTB link to come up, and then proceeds to read and write the doorbell scratchpad registers of the NTB. The peers interrupt each other using a bit mask of doorbell bits, which is shifted by one in each round, to test the behavior of multiple doorbell bits and interrupt vectors. The Ping Pong driver also reads the first local scratchpad, and writes the value plus one to the first peer scratchpad, each round before writing the peer doorbell register.

Module Parameters:

- unsafe - Some hardware has known issues with scratchpad and doorbell
  registers. By default, Ping Pong will not attempt to exercise such hardware. You may override this behavior at your own risk by setting unsafe=1.
- delay_ms - Specify the delay between receiving a doorbell
  interrupt event and setting the peer doorbell register for the next round.
- init_db - Specify the doorbell bits to start new series of rounds. A new
  series begins once all the doorbell bits have been shifted out of range.
- dyndbg - It is suggested to specify dyndbg=+p when loading this module, and
  then to observe debugging output on the console.

NTB Tool Test Client (ntb_tool)

The Tool test client serves for debugging, primarily, ntb hardware and drivers. The Tool provides access through debugfs for reading, setting, and clearing the NTB doorbell, and reading and writing scratchpads.

The Tool does not currently have any module parameters.

Debugfs Files:

- *debugfs*/ntb_tool/*hw*/
  A directory in debugfs will be created for each NTB device probed by the tool. This directory is shortened to *hw* below.
- *hw*/db
  This file is used to read, set, and clear the local doorbell. Not all operations may be supported by all hardware. To read the doorbell, read the file. To set the doorbell, write *s* followed by the bits to set (eg: *echo 's 0x0101' > db*). To clear the doorbell, write *c* followed by the bits to clear.
- *hw*/mask
  This file is used to read, set, and clear the local doorbell mask. See *db* for details.

- *hw*/peer_db
  This file is used to read, set, and clear the peer doorbell. See *db* for details.
- *hw*/peer_mask
  This file is used to read, set, and clear the peer doorbell mask. See *db* for details.
- *hw*/spad
  This file is used to read and write local scratchpads. To read the values of all scratchpads, read the file. To write values, write a series of pairs of scratchpad number and value (eg: *echo '4 0x123 7 0xabc' > spad #* to set scratchpads *4* and *7* to *0x123* and *0xabc*, respectively).
- *hw*/peer_spad
  This file is used to read and write peer scratchpads. See *spad* for details.

NTB MSI Test Client (ntb_msi_test)

The MSI test client serves to test and debug the MSI library which allows for passing MSI interrupts across NTB memory windows. The test client is interacted with through the debugfs filesystem:

- *debugfs*/ntb_tool/*hw*/
  A directory in debugfs will be created for each NTB device probed by the tool. This directory is shortened to *hw* below.
- *hw*/port
  This file describes the local port number
- *hw*/irq*_occurrences
  One occurrences file exists for each interrupt and, when read, returns the number of times the interrupt has been triggered.
- *hw*/peer*/port
  This file describes the port number for each peer
- *hw*/peer*/count
  This file describes the number of interrupts that can be triggered on each peer
- *hw*/peer*/trigger
  Writing an interrupt number (any number less than the value specified in count) will trigger the interrupt on the specified peer. That peer's interrupt's occurrence file should be incremented.

NTB Hardware Drivers

NTB hardware drivers should register devices with the NTB core driver. After registering, clients probe and remove functions will be called.

NTB Intel Hardware Driver (ntb_hw_intel)

The Intel hardware driver supports NTB on Xeon and Atom CPUs.

Module Parameters:

- b2b_mw_idx
  If the peer ntb is to be accessed via a memory window, then use this memory window to access the peer ntb. A value of zero or positive starts from the first mw idx, and a negative value starts from the last mw idx. Both sides MUST set the same value here! The default value is *-1*.
- b2b_mw_share
  If the peer ntb is to be accessed via a memory window, and if the memory window is large enough, still allow the client to use the second half of the memory window for address translation to the peer.
- xeon_b2b_usd_bar2_addr64
  If using B2B topology on Xeon hardware, use this 64 bit address on the bus between the NTB devices for the window at BAR2, on the upstream side of the link.
- xeon_b2b_usd_bar4_addr64 - See *xeon_b2b_bar2_addr64*.
- xeon_b2b_usd_bar4_addr32 - See *xeon_b2b_bar2_addr64*.
- xeon_b2b_usd_bar5_addr32 - See *xeon_b2b_bar2_addr64*.
- xeon_b2b_dsd_bar2_addr64 - See *xeon_b2b_bar2_addr64*.
- xeon_b2b_dsd_bar4_addr64 - See *xeon_b2b_bar2_addr64*.
- xeon_b2b_dsd_bar4_addr32 - See *xeon_b2b_bar2_addr64*.
- xeon_b2b_dsd_bar5_addr32 - See *xeon_b2b_bar2_addr64*.