Contents

1.0 Introduction .............................................................................................................. 5  
   1.1 Purpose of this Document ..................................................................................... 7  
   1.2 Document Scope ................................................................................................. 7  
   1.3 Terminology ....................................................................................................... 7  
   1.4 Related Documents ............................................................................................. 8  
   1.5 Feature List ....................................................................................................... 8  
2.0 Hardware and Software Overview ............................................................................. 9  
   2.1 Interrupt Operation ............................................................................................. 9  
   2.2 OS System Support ............................................................................................. 9  
   2.3 Development Tools Information ............................................................................. 9  
   2.4 Architectural View ............................................................................................. 10  
3.0 Non-Transparent Bridge Driver Client API Overview ................................................ 13  
   3.1 Interface .......................................................................................................... 13  
   3.2 Basic API Flow .................................................................................................. 14  
   3.3 Dependencies ................................................................................................... 15  
   3.4 Key Assumptions............................................................................................... 15  
   3.5 Error Handling .................................................................................................. 16  
4.0 Sample Client Protocol Overview ............................................................................. 17  
   4.1 Interface .......................................................................................................... 17  
      4.1.1 Doorbell Definition.................................................................................. 17  
      4.1.2 Scratchpad Registers .............................................................................. 18  
   4.2 Heartbeat ......................................................................................................... 18  
   4.3 Data Transfer with Encryption/Decryption ............................................................. 19  
   4.4 Negotiation or Quiescing .................................................................................... 20  
5.0 Non-Transparent Bridge Driver Details ................................................................... 22  
   5.1 Overview ......................................................................................................... 22  
   5.2 Dependencies ................................................................................................... 22  
   5.3 Resource Usage and Power Management .............................................................. 22  
6.0 API Usage ............................................................................................................... 24  
   6.1 Introduction ..................................................................................................... 24  
   6.2 Initialization and Clean Up ................................................................................. 24  
   6.3 Registration/Deregistration ................................................................................. 24  
   6.4 Writing to Translate Address and Limit Registers ................................................... 26  
   6.5 Writing to Doorbell and Scratchpad Registers ....................................................... 27  
   6.6 Add, Reset, Get Policy ........................................................................................ 28  
   6.7 Get BAR Address ............................................................................................... 29  
   6.8 Read/Write to Remote Limit ................................................................................ 29  

Figures

1  Usage Models for the NTB ........................................................................................... 6  
2  Interrupts Driven by Doorbell “Rings” ....................................................................... 9  
3  Relationship of NTB Driver’s Client API and Client Driver Protocol .......................... 10  
4  Memory Translation Windows .................................................................................. 11  
5  Back to Back Configuration ....................................................................................... 12  
6  Back to Back Communications .................................................................................. 12  
7  Sample API Flow Diagram ........................................................................................ 14
Revision History

<table>
<thead>
<tr>
<th>Document Number</th>
<th>Revision Number</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>323402</td>
<td>001</td>
<td>Initial release.</td>
<td>February 2010</td>
</tr>
</tbody>
</table>

§ §
1.0 Introduction

The task of the non-transparent bridge (NTB) is to provide electrical isolation between two subsystems, local and remote, while providing access to a memory window into each subsystem. The NTB is seen as a root complex integrated endpoint (RCIEP) or a PCI Express (PCIe) endpoint, exposing a Type 0 PCIe configuration space. As shown in Figure 1, the RCIEP can be flexibly configured for three different usage models.

The first usage model fits existing customer requirements:

An Intel® Xeon® Processor C5500/C3500 Series local host's secondary side of the NTB is connected to a Intel Xeon processor remote host's secondary side of the NTB, referred to as the back-to-back (B2B) model.

The next two usage models are variations on the classic non-transparent bridge usage model:

- An Intel Xeon processor local host NTB port connected to the Intel Xeon processor remote host's root port (RP) and the Intel Xeon processor remote host's NTB secondary port connected to the local host's root port (symmetric configuration).
- An Intel Xeon processor local host NTB port connected to the Intel Xeon processor or non-Intel Xeon processor remote host's root port (non-symmetric configuration).
The different usage models require the development of three NTB drivers, further referred to as the “NTB driver(s).” This document specifies the implementation of the three NTB drivers and an exposed “client API” to be used by a client driver. The document covers the necessary configuration details for each of the drivers and the usage models:

- **B2B Driver**: NTB Driver for an NTB with its secondary side attached to another NTB’s secondary side. The same driver is used on each Intel Xeon Processor C5500/C3500 Series system in this configuration.
- **NTB Classic Driver**: NTB Driver driving the primary side of the Intel Xeon processor NTB, attached to an RP (an Intel Xeon processor or non-Intel Xeon processor system).
- **Root Port (RP) Classic Driver**: NTB driver running on a remote system and driving the secondary side of the Intel Xeon processor NTB, which is attached to an RP.

A user chooses the NTB configuration in the BIOS configuration screens.

The different usage models require the development of three NTB drivers, further referred to as the “NTB driver(s).” This document specifies the implementation of the three NTB drivers and an exposed “client API” to be used by a client driver. The document covers the necessary configuration details for each of the drivers and the usage models:

- **B2B Driver**: NTB Driver for an NTB with its secondary side attached to another NTB’s secondary side. The same driver is used on each Intel Xeon Processor C5500/C3500 Series system in this configuration.
- **NTB Classic Driver**: NTB Driver driving the primary side of the Intel Xeon processor NTB, attached to an RP (an Intel Xeon processor or non-Intel Xeon processor system).
- **Root Port (RP) Classic Driver**: NTB driver running on a remote system and driving the secondary side of the Intel Xeon processor NTB, which is attached to an RP.

A user chooses the NTB configuration in the BIOS configuration screens.
1.1 Purpose of this Document

This document describes the high-level design features and requirements of the NTB device drivers for the Intel Xeon processor. The NTB drivers are Linux* reference drivers that support several different topologies or usage models involving a local host, which can be an Intel Xeon Processor C5500/C3500 Series system, and a remote host, which can be an Intel Xeon processor or non-Intel Xeon processor system. The NTB drivers provide a basic driver-to-driver interface, or client API, which is exposed to potential client drivers in order to enable use of NTB configuration and memory translation features. The client API is described in detail and example usage is provided.

1.2 Document Scope

This document covers the NTB drivers, all of which are reference drivers. Currently, the document describes only the Linux version of the drivers and the NTB client driver API usage.

1.3 Terminology

Table 1. Terms and Definitions

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address translation</td>
<td>A method for creating a memory window from one subsystem to another connected by the NTB.</td>
</tr>
<tr>
<td>B2B</td>
<td>back to back</td>
</tr>
<tr>
<td>BAR</td>
<td>base address register</td>
</tr>
<tr>
<td>doorbell register</td>
<td>A set of registers enabling each side of the NTB to interrupt the opposite side.</td>
</tr>
<tr>
<td>heartbeat protocol</td>
<td>A method for two or more systems to communicate their state of availability or well-being, typically used in fail-over scenarios.</td>
</tr>
<tr>
<td>IDE</td>
<td>Interactive Development Environment</td>
</tr>
<tr>
<td>ISR</td>
<td>Interrupt Service Routine</td>
</tr>
<tr>
<td>journaling (file system)</td>
<td>Recording modifications to a log or &quot;journal&quot; before writing them to the main file system, diminishing chances for corruption in the event of a system crash.</td>
</tr>
<tr>
<td>memory window</td>
<td>An allotted amount of memory is associated with each 2/3 and 4/5 BAR on each side of the non-transparent bridge. Subsystems on opposite sides of the non-transparent bridge communicate (read and write) via these memory windows. A memory window is &quot;moved&quot; by modifying the translate register where the address of the beginning of the memory window is stored.</td>
</tr>
<tr>
<td>NTB</td>
<td>Non-transparent bridge</td>
</tr>
<tr>
<td>NTB protocols</td>
<td>Protocols associated with the NTB drivers.</td>
</tr>
<tr>
<td>PCIe</td>
<td>PCI Express</td>
</tr>
<tr>
<td>PM</td>
<td>Power Management</td>
</tr>
<tr>
<td>Primary and Secondary sides of NTB</td>
<td>The NTB has both a primary (&quot;local&quot;) and secondary (&quot;remote&quot;) side with primary and secondary PCIe configuration registers and memory-mapped registers. Secondary-side registers are visible to primary side. Primary-side registers are not visible to the secondary side. The primary and secondary sides of the NTB serve as hardware interfaces between two subsystems. The remote system is another system connected to the local system via a backplane.</td>
</tr>
<tr>
<td>RCIEP</td>
<td>Root Complex integrated Endpoint</td>
</tr>
<tr>
<td>RP</td>
<td>Root Port</td>
</tr>
</tbody>
</table>
1.4 Related Documents

Table 2. Related Documents

<table>
<thead>
<tr>
<th>Document Name</th>
<th>Document No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® Xeon® Processor C5500/C3500 Series Non-Transparent Bridge White Paper</td>
<td>323328</td>
</tr>
</tbody>
</table>

1.5 Feature List

This section lists all of the features supported by the NTB drivers.

- Three different NTB drivers: 1) an NTB driver for the B2B configuration 2) an NTB driver running on the local side driving the primary side of the NTB, and 3) an NTB driver running on a remote system driving the secondary side of the Intel Xeon processor NTB, which is attached to a root port.
- Support for up to two client drivers per NTB device. A client requests access to a BAR by registering with the NTB. A single client may own both BARs.
- Read/write for scratchpad registers and doorbell write access to allow clients to define an information exchange protocol between systems.
- Doorbell bit policy definition.
- Client registration and event notification via callback.
- Read/write access for BAR limit size and translation address movement.
- Obtain/release semaphore.
- Snoop level write.
- BAR retrieval for BAR 2/3 and BAR 4/5.
- Get/set NTB link status.
2.0 Hardware and Software Overview

2.1 Interrupt Operation

The NTB contains two 16-bit doorbell registers for communication between the local and remote subsystems or the primary and secondary sides of the non-transparent bridge.

The interrupt scheme for the driver is driven by the doorbell register access. Figure 2 illustrates an NTB with primary (PDB) and secondary (SDB) side doorbells. Writes to the primary side doorbell, via the NTB driver's client API, from the secondary side cause interrupts to occur on the primary side. The NTB driver on the primary side has an ISR that clears the doorbell register and notifies primary side client driver(s) via callback that a doorbell “ring” has occurred.

The driver supports legacy (INTx), MSI and MSIx style interrupts. Because the NTB driver also supports multiple interrupts in an MSIx enabled system, the driver determines the "n" number of interrupts supported during the driver initialization phase.

Figure 2. Interrupts Driven by Doorbell “Rings”

2.2 OS System Support

The drivers were initially developed on Red Hat Enterprise Linux* (RHEL) kernel 2.6.32.7 or later.

2.3 Development Tools Information

The drivers were developed initially using the standard compiler available on Red Hat Linux 5.0 and 5.3. The Eclipse Ganymede* environment was used as an IDE, which provided an editor and a method to invoke the compiler as well as a Doxygen* plug-in.
2.4 Architectural View

The client driver must use the client API of an NTB driver in order to facilitate communication between the local subsystem on the primary side and the remote subsystem on the secondary side. The NTB has a shared semaphore register (SEM) and a shared set of sixteen 32-bit scratchpad registers accessible from both the primary and secondary sides. Also available are primary (PDB) and secondary doorbell (SDB) registers, each 16-bits in length. The client API for accessing these registers from a client driver is rather general and can be used in a myriad of ways with many different client driver scenarios. The way in which a pair of client drivers choose to use the NTB driver’s client API, and, by extension, the NTB registers for subsystem communication, is called the client driver protocol. A sample client driver protocol is defined later in the document.

Figure 3. Relationship of NTB Driver’s Client API and Client Driver Protocol

Communication between the two clients is achieved by handshaking writes to the scratchpad and doorbell registers. In addition, the client driver registers with the NTB driver and supplies a callback, a pointer to a routine in the client driver. A sample communication scenario involving scratchpad register writes is outlined below:

- Obtain the Semaphore: First, the primary client driver (PCD) must obtain the semaphore. The secondary side client driver (SCD) has just released the semaphore making it available. The primary side NTB driver grants the PCD the semaphore.
- Write to the Scratchpad Registers: The PCD writes the message to the scratchpad registers. There are 16 available registers.
- Ring Doorbell: The PCD then "rings" the SDB or writes to it so that the NTB driver on the secondary side is aware that the data is available. A client protocol depends on the usage of the doorbell, which contains 16 bits; however, only 14 bits are available for use by the client. A protocol design involves defining policy of each doorbell bit.
- Call the Client Driver: The SCD client driver is notified by a callback routine. The routine contains the data in the scratchpad registers.
- Ring Doorbell: The SCD then "rings" the PDB or writes to it so that the NTB driver on the primary side may notify the PCD with a callback that the scratchpad write was acknowledged and received.
- Release Semaphore: The semaphore is released by the PCD.

A major feature of the NTB is the ability to configure a memory window for access on opposite sides of the NTB. The NTB achieves this with its multiple sets of 64-bit base address registers (BARs) and translate registers in which it stores the physical
addresses of contiguous memory windows for each side. In addition to the BAR 0/1, which the NTB driver uses for its own PCIe configuration space, the NTB has a BAR 2/3 and a BAR 4/5 on its primary and secondary sides. These BARs correlate to memory windows on each subsystem. The NTB provides address translation registers for both sets of BARs.

As illustrated in Figure 4, a client driver may request PCI BAR regions 2/3, BAR 4/5 or both. The client driver may use the API to store the address of a contiguous memory window that is accessible for BAR writes from the opposite side of the NTB.

In the example below each client has claimed a BAR 2/3. The translate registers are set to contiguous memory locations on each subsystem. A write to BAR 2/3 by the primary side client driver results in data being written to the contiguous memory window on the secondary side.

**Figure 4. Memory Translation Windows**

The architectural details are slightly different for the B2B configuration. Because there are two NTBs involved, the number of registers discussed previously doubles. However, the details are transparent to the client driver.

Complexity arises for software when the NTBs are placed in a B2B configuration. An inaccessible secondary space is created between the secondary sides of the two systems as shown in Figure 5. Each system’s enumeration software stops at the NTB endpoint. The primary host has no knowledge of the secondary system NTB doorbell and scratchpad. The reverse is also true. While data can be moved through this system through the PCIe links, interrupts cannot travel from one system to the other.
The problem is solved by a proxy packet generating mechanism that exists between the secondary sides of the two systems, enabling one system to send a PCIe interrupt packet through the MMIO-inaccessible area into the opposite NTB, effectively giving each host a tunnel into the other system’s MMIO areas, including both the scratchpad protocol area and the interrupt generating doorbell area as shown in Figure 6. The figure illustrates the secondary configuration space path used by drivers located on opposite sides of the bridge for communicating protocol messages.

Upon seeing an assert change in state in the back to back shadow register set in NTBA, NTBA logic creates a posted memory write packet containing the back to back shadow registers in the payload of the posted memory write and sends it across the link. NTBB decodes the posted memory write (PMWR) as its own and updates its local shadow registers. NTBB registers sense a change in the state of B2B shadow register set and generate an upstream interrupt (MSI/MSI-X, or INTx), depending on what interrupt mechanism is selected.
3.0 Non-Transparent Bridge Driver Client API Overview

3.1 Interface

Table 3 provides a general overview of the client API for all of the NTB drivers.

Table 3. NTB Client API

<table>
<thead>
<tr>
<th>API Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client Register</td>
<td>To use the NTB device, the client driver must register with the NTB driver. The client supplies a pointer to a function in the client driver referred to as a “callback routine”, whereby the NTB driver notifies the client driver of certain events. The client asks to be registered to one or more BARs. If successful, the NTB driver returns a type of handle.</td>
</tr>
<tr>
<td>Client Un-register</td>
<td>The client driver un-registers with the NTB driver using the handle obtained during registration.</td>
</tr>
<tr>
<td>Obtain Semaphore</td>
<td>To write to the scratchpad registers, the client driver obtains ownership of the shared semaphore before writing to the scratchpad registers. The NTB driver returns a 0 (successful) or &lt;0 (unsuccessful).</td>
</tr>
<tr>
<td>Release Semaphore</td>
<td>The client driver uses this function to release ownership of the semaphore.</td>
</tr>
<tr>
<td>Write One/All Scratchpad Registers</td>
<td>To write to one or more scratchpad registers, the client driver supplies a 16-field array or structure containing the data values. The NTB driver returns a 0 (successful) or &lt;0 (unsuccessful).</td>
</tr>
<tr>
<td>Read One/All Scratchpad Registers</td>
<td>The NTB driver returns a 16-field array or structure containing the scratchpad data. The client driver should allocate the structure into which the data is returned.</td>
</tr>
<tr>
<td>Write Doorbell Register</td>
<td>To write to an opposite subsystem’s doorbell register, the client driver supplies a 16-bit value. The NTB driver returns a 0 (successful) or &lt;0 (unsuccessful).</td>
</tr>
<tr>
<td>Write Address Translation Value</td>
<td>The client driver supplies a 64-bit physical address for address translation for each BAR. (If a client wants to own two BARs, it must call the API twice.) The address must point to a contiguous memory region. The NTB driver returns a 0 (successful) or &lt;0 (unsuccessful).</td>
</tr>
<tr>
<td>Write/Read Limit Value</td>
<td>The client driver supplies a 64-bit value to serve as the memory mapped region’s size limit. Memory size values are limited to 4KB resolution and may be up to 512 GB. The NTB driver returns a 0 (successful) or &lt;0 (unsuccessful).</td>
</tr>
<tr>
<td>Get/Set Link</td>
<td>This API allows the client to enable and disable the link on the secondary side of the NTB. This API is available only to the client on the Primary side of the NTB.</td>
</tr>
<tr>
<td>Set Snoop Level</td>
<td>This client can set the snoop level per BAR. The client can choose between three snoop levels: 1. All transaction layer packet (TLP) sent as defined by the ATTR field. 2. Force Snoop on all TLPs. 3. Force No-Snoop on all TLPs.</td>
</tr>
</tbody>
</table>
### 3.2 Basic API Flow

As mentioned above, the driver supports two registered clients per side as shown in Figure 7. If two clients exist on the primary side, it is not necessary for two to exist on the secondary side, but it is customary.

#### Figure 7. Sample API Flow Diagram

<table>
<thead>
<tr>
<th>API Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledge Power Management Event (PM)</td>
<td>The client uses this API to acknowledge a PM event and to signal readiness for the idle state.</td>
</tr>
<tr>
<td>Add/Reset/Get Policy</td>
<td>To define a protocol, the client driver must set the policy for the doorbell register. The doorbell policy consists of the bit usage of the 14 available doorbell bits.</td>
</tr>
<tr>
<td>Get Number of Devices</td>
<td>This API returns the number of NTB devices present in the system.</td>
</tr>
<tr>
<td>Get BAR Address</td>
<td>The client may retrieve the physical address of a BAR.</td>
</tr>
<tr>
<td>Get Next Available Bus/Device/Function (BDF)</td>
<td>Because more than one NTB device may exist in an Intel® Xeon® Processor C5500/C3500 Series system, the client must specify which device and bar is being requested by supplying a bus/device/function parameter and a BAR parameter. An API is available to help the client determine the next available Bus/Device/Function.</td>
</tr>
<tr>
<td>Get Total Number of Unused BDFs</td>
<td>The client uses this API to discover how many BDF/BAR combinations are available.</td>
</tr>
<tr>
<td>Read/Write Write Cache Control Register (WCCNTRL) Bit</td>
<td>Writing to a bit in the WCCNTRL register forces a snapshot flush of the entire IIO write cache. Reading a zero from this bit indicates that the flush operation is complete.</td>
</tr>
<tr>
<td>Read/Write Remote Translate Address</td>
<td>This API reads/writes the remote side's translate address from the primary side.</td>
</tr>
<tr>
<td>Read/Write Remote Doorbell Mask</td>
<td>This API reads/writes the remote side's doorbell mask register from the primary side.</td>
</tr>
<tr>
<td>Read/Write Remote Limit</td>
<td>This API reads/writes the remote side's limit register from the primary side.</td>
</tr>
<tr>
<td>Read/Write Remote BAR</td>
<td>This API reads/writes the remote side's BAR from the primary side.</td>
</tr>
</tbody>
</table>

**Table 3. NTB Client API (Continued)**

![Sample API Flow Diagram](image-url)
Every client must register with the NTB driver to use it for operations. However, it is somewhat artificial to outline a basic flow for the NTB client API, as client drivers determine protocols involving the doorbell and scratchpad writes that defines communication. Communication between subsystems in terms of heartbeat protocol and data writing varies widely.

As an example, Table 4 summarizes the basic API flow and usage by a client driver for setting up or moving a memory window.

### Table 4. Steps for Address Translation Negotiation

<table>
<thead>
<tr>
<th>Step</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Register</td>
<td>Client driver registers requesting one bar. To request both bars the client must register twice with the NTB driver.</td>
</tr>
<tr>
<td>2. Negotiate or quiesce</td>
<td>Client driver uses doorbell register to inform opposite side of NTB that it is about to set the memory window. Client waits for acknowledgement from other client. NTB driver informs a client through callback mechanism.</td>
</tr>
<tr>
<td>3. Memory allocation</td>
<td>Client driver allocates memory for the address translation.</td>
</tr>
<tr>
<td>4. Address Translation</td>
<td>Client driver writes physical address translation value.</td>
</tr>
<tr>
<td>5. Limit register</td>
<td>Client writes size limit.</td>
</tr>
</tbody>
</table>

### 3.3 Dependencies

One or two client drivers are supported per NTB device. The client drivers call the NTB Client API, which is exposed to potential clients. Client drivers are likely driven by applications (see Figure 8).

### Figure 8. Dependencies

![Dependencies Diagram](image)

### 3.4 Key Assumptions

Below are several key points to remember:

- A maximum of two clients NTB device are allowed.
- The registration API returns a unique handle, which a client saves uses as a parameter for subsequent client API calls.
- A client supplies the NTB driver with physical addresses only.
- The client driver should implement a callback routine whereby it may receive messages from the NTB driver including the contents of the doorbell and scratchpad registers.
- A client driver writes to the scratchpad register after obtaining a semaphore.
3.5 Error Handling

Errors are logged using the standard Linux logging prints to /var/log/messages.
4.0 Sample Client Protocol Overview

4.1 Interface

Table 5 describes example client driver protocols.

Table 5. Example Protocols

<table>
<thead>
<tr>
<th>API Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heartbeat</td>
<td>The client drivers communicate the state of well-being or availability of hardware through a heartbeat protocol.</td>
</tr>
<tr>
<td>Encrypt/Decrypt Data Movement</td>
<td>The client drivers exchange encrypted data at the locations pointed to by the BAR translate registers.</td>
</tr>
<tr>
<td>Quiesce/Negotiation</td>
<td>The client drivers negotiate movement of the BAR memory windows.</td>
</tr>
</tbody>
</table>

4.1.1 Doorbell Definition

The doorbell register is the primary means of communication between subsystems and is used to indicate both the health of hardware and of various client driver states. The doorbell registers on the primary and secondary sides contain 14 bits for generic usage. Table 6 defines example usage of the doorbells in the reference design.

Table 6. Doorbell Bits

<table>
<thead>
<tr>
<th>Bit and Code</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 HEARTBEAT</td>
<td>Heartbeat for client</td>
</tr>
<tr>
<td>1 SC_DATA_AVAIL_23</td>
<td>Data Available in Scratchpad for client associated with BAR 2/3</td>
</tr>
<tr>
<td>2 SC_DATA_ACK_23</td>
<td>Acknowledgment/Data Read from Scratchpad</td>
</tr>
<tr>
<td>3 SC_DATA_AVAIL_45</td>
<td>Data Available in Scratchpad for client associated with BAR 4/5</td>
</tr>
<tr>
<td>4 SC_DATA_ACK_45</td>
<td>Acknowledgment/Data Read from Scratchpad</td>
</tr>
<tr>
<td>5 MW_MOVE_REQ_23</td>
<td>Memory Window Movement Request for client associated with BAR 2/3</td>
</tr>
<tr>
<td>6 MW_MOVE_ACK_23</td>
<td>Acknowledgement of Window Movement Request</td>
</tr>
<tr>
<td>7 MW_MOVE_COMP_23</td>
<td>Memory Window Move Complete</td>
</tr>
<tr>
<td>8 MW_MOVE_REQ_45</td>
<td>Memory Window Movement Request for client associated with BAR 4/5</td>
</tr>
<tr>
<td>9 MW_MOVE_ACK_45</td>
<td>Acknowledgement of Window Movement Request</td>
</tr>
<tr>
<td>10 MW_MOVE_COMP_45</td>
<td>Memory Window Move Complete</td>
</tr>
<tr>
<td>11 EVENT_NOTIFICATION</td>
<td>Power management event notification (other side must ready itself for the event to take place)</td>
</tr>
<tr>
<td>12 EVENT_ACKNOWLEDGEMENT</td>
<td>Acknowledgement of receipt (implies readiness for the idle state)</td>
</tr>
<tr>
<td>13</td>
<td>UNDEFINED</td>
</tr>
<tr>
<td>14</td>
<td>RESERVED</td>
</tr>
<tr>
<td>15</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
4.1.2 Scratchpad Registers

A single set of sixteen 32-bit scratchpad registers is also available for use. Access is shared between primary and secondary sides and is controlled by obtaining a semaphore. The sample below defines a structure for example usage of the scratchpad registers in a reference design.

**Note:** Some fields require two 32-bit registers to accommodate the 64-bit addresses that are possible in the Intel® Xeon® Processor C5500/C3500 Series.

```c
struct ntb_scratchpad {
    int32_t client_id;
    int32_t size_upper;
    int32_t size_lower;
    int32_t packet_no_upper;
    int32_t packet_no_lower;
    int32_t packet_size_upper;
    int32_t packet_size_lower;
    int32_t offset_size_upper;
    int32_t offset_size_lower;
    int32_t encryption;
    int32_t reserved_one;
    int32_t reserved_two;
    int32_t reserved_three;
    int32_t reserved_four;
    int32_t reserved_five;
    int32_t reserved_six;
};
```

### Table 7. Description of Structure for Scratchpad Registers

<table>
<thead>
<tr>
<th>DWORD and Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 CLIENT_ID</td>
<td>Client Driver ID (returned by the NTB driver upon registration). Up to four different client drivers may be registered with the NTB driver; hence, the client ID requirement.</td>
</tr>
<tr>
<td>1 SIZE_UPPER</td>
<td>The total size of the data block. Two registers, size upper and size lower, are needed to accommodate extremely large windows (up to 16 GB).</td>
</tr>
<tr>
<td>2 SIZE_LOWER</td>
<td></td>
</tr>
<tr>
<td>3 PACKET_NO_UPPER</td>
<td>Total number of Packets. The contiguous memory window on a system can be broken into serial chunks or an array of “packets.” If this field is 0 or 1, it is assumed that the data is not broken into packets.</td>
</tr>
<tr>
<td>4 PACKET_NO_LOWER</td>
<td></td>
</tr>
<tr>
<td>5 PACKET_SIZE_UPPER</td>
<td>Size of each packet.</td>
</tr>
<tr>
<td>6 PACKET_SIZE_LOWER</td>
<td></td>
</tr>
<tr>
<td>7 OFFSET_SIZE_UPPER</td>
<td>Offset into the window. This may be set to 0 to indicate the start of the memory window.</td>
</tr>
<tr>
<td>8 OFFSET_SIZE_LOWER</td>
<td></td>
</tr>
<tr>
<td>9 ENCRYPTION</td>
<td>Encryption indicator. This field indicates whether the data at the memory window is encrypted.</td>
</tr>
</tbody>
</table>

4.2 Heartbeat

As mentioned above, the driver supports a maximum of four registered clients per side. If two clients exist on the primary side, two corresponding clients may exist on the secondary side. However, a single set of clients communicate the state of well-being or
availability of the hardware through writes to the doorbell register. This is referred to as the heartbeat protocol (please see Table 8). The heartbeat mechanism is depicted in Figure 9.

**Figure 9. Heartbeat Mechanism**

![Heartbeat Mechanism Diagram](image)

An interrupt is generated when a client writes to the doorbell register. The NTB driver determines that the message is for a particular client (the client handling heartbeat messages) and sends a callback to that client, indicating its corresponding client on the other side of the NTB is “alive.”

The client driver contains a timing mechanism that fires off the send heartbeat messages periodically. If the client does not receive a heartbeat message from the corresponding client on the other side the NTB, the client takes corrective action.

**Table 8. Heartbeat Protocol**

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send heartbeat</td>
<td>A function that sends heartbeat messages.</td>
</tr>
</tbody>
</table>

### 4.3 Data Transfer with Encryption/Decryption

Figure 10 shows two clients exchanging encrypted data through the protocol.

The client driver, with a handle of 0x00010001, uses a protocol to write encrypted data into the memory window on the other subsystem. The corresponding client driver with a handle of 0x00010002 on subsystem two acknowledges receipt and decrypts the data. It writes the data back to the other side and alerts the first client. Of course, the operation may happen in reverse.

The protocol allows the client drivers the flexibility of writing encrypted or decrypted data to the opposite side, implementing an off-loading scenario.
To write encrypted data, translate addresses for BARs must be configured properly and encrypted data block must be contiguous. The following steps are part of the write encrypted data protocol:

- The client one (with a handle of 0x00010001) writes encrypted data to a mapped memory window on the opposite subsystem.
- The client one is granted the semaphore by the NTB driver so that the scratchpad registers can be written.
- Client one writes relevant information to the scratchpad registers, again, using the client API in the NTB driver.
- Client one writes to doorbell (SC_DATA_AVAIL) via NTB driver client API.
- When the doorbell write occurs, client two (with a handle of 0x00010002) receives a callback with doorbell and scratchpad data contained in the callback. Client two acknowledges receipt by writing to doorbell (SC_DATA_ACK), which causes client one to receive a callback and the semaphore is freed for use. Client two takes action, such as encrypting or decrypting data and writing it back to the other side.
- The data available and data acknowledge process occurs again.

4.4 Negotiation or Quiescing

Client drivers may need to change the memory window locations associated with a BAR. These locations are stored in the BAR translate register and the client driver uses the NTB client API to modify them. However, the client cannot modify these registers safely while data is being written to the BAR memory windows, which implies that a method is required by which they can negotiate a window change.
• The client driver that wants to change the memory location must alert the other subsystem that it intends to change this register by writing to the doorbell register (MW_MOVE_REQ).

• Client driver two receives a callback on the other side. It must gracefully finish activities associated with the window and acknowledge receipt of the request, indicating it is ready for the move (MW_MOVE_ACK).

• Client driver one receives a callback of acknowledgement. After client one has changed the window location using the NTB client API, it should send a completion message to the other client (MW_MOVE_COMP).

### Table 10. Negotiation API

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request a move</td>
<td>A client issues a request to move the window.</td>
</tr>
<tr>
<td>Acknowledge and prepare for move</td>
<td>The client on the opposite side prepares for the move and acknowledge when ready.</td>
</tr>
<tr>
<td>Notification of move completion</td>
<td>The client notifies upon completion.</td>
</tr>
</tbody>
</table>
5.0 Non-Transparent Bridge Driver Details

5.1 Overview

Three distinct drivers are outlined in the Chapter 1.0, “Introduction”:

- B2B Driver
- NTB Classic Driver
- Root Port (RP) Classic Driver

The list above comprises the three NTB driver scenarios that can be configured via a
write PCIe Port Definition register by the BIOS. The write sets the value of the device
ID available to the driver as it goes through its configuration phase.

Upon configuration, the physical and virtual addresses of BAR resources and the
maximum memory window limit apertures are stored in internal structures. The driver
sets up the interrupt scheme based on the system configuration (INTx, MSI, MSIx).
The driver clears the doorbell bit and sets the doorbell mask.

There are three distinct NTB drivers; however, the exposed client API is the same for
each driver. The differences between the three drivers do not matter to the client
driver.

5.2 Dependencies

The NTB drivers do not have any dependencies except those standard to all Linux
drivers. Client drivers depend upon the NTB client API.

5.3 Resource Usage and Power Management

When the NTB driver receives a power management event notification from the
operating system, it must prepare for a power down or suspend to RAM state.
Figure 11 outlines the sequence of event notifications. However, it should be noted that
events may not occur as synchronously as they appear in the figure, which is for
illustration purposes only.
1) The NTB driver receives a power management event notification.

2 a and b) The NTB driver notifies each registered client of the event through a callback. The clients notify the driver via API when they have completed outstanding work and are ready to be brought to an idle state.

3) The NTB driver notifies the secondary side of the event by ringing the secondary side’s doorbell.

4 a and b) The secondary side driver goes through a process similar to 2a/2b.

5) When the secondary side has received an acknowledgement from its client, it rings the doorbell on the primary side. Each side is ready for the primary side to go into a suspend to RAM or power off state.

6) An additional step, not shown above, is the NTB driver’s return to the OS, indicating that power can be safely removed.

For registered clients, the NTB drivers handle power management events by callback notification. Each client receives a callback and the client should handle the event accordingly. If the client is in the middle of a copy activity, the client should either stop or complete the copy activity and send a message to the NTB driver when it is ready.

Upon the suspend to RAM state, the NTB driver saves the value of registers and it saves the client structure (handle and callback pointer) so that they do not have to reregister after power on.
6.0 API Usage

6.1 Introduction

The NTB driver exposes an API to be used by a client driver, another device driver or a purely software driver. The following sections describe the usage of several NTB APIs in a client driver along with source code examples. These sections do not exhaustively describe or illustrate the entire set of NTB APIs. For more details, see function prototypes in the ntb_api.h file in the NTB driver.

6.2 Initialization and Clean Up

A client driver uses the NTB driver’s exposed APIs by creating and initializing an API function table. The functions ntb_get_b2b_api, ntb_get_classic_api or ntb_get_rootport_api are used to fill in function tables for each of the three driver’s APIs. The struct ntb_api_export defines the NTB driver’s function table. The client driver uses the typedef ntb_get_api_t to call the ntb_get_api function.

Sample:

```c
/* Declare a function table. */
struct ntb_api_export ntb_funcs = { };
/* Declare a function pointer of type ntb_get_api_t. */
ntb_get_api_t function_get;
/* Call Linux API/macro to obtain ntb_get_api for the B2B driver */
function_get = symbol_request(ntb_get_b2b_api);
/* Initialize the function table. */
function_get(&ntb_funcs);
```

A call to symbol_get requires a call to symbol_put as the client is unloaded.

Sample:

```c
symbol_put(ntb_get_b2b_api);
```

6.3 Registration/Deregistration

After initializing the function table, the client driver registers with the NTB driver and requests access to a BAR by calling ntb_register_client. An NTB device is associated with each processor on a system; therefore, it is necessary to specify a bus, device, and function (BDF) combination along with the BAR ID in the call to ntb_register_client.
The client should supply a pointer to a callback routine for effective monitoring of NTB doorbell and scratchpad events; however, the register routine will not return an error if none is supplied.

NTB driver provides enumerated types to request access to BAR 2/3 and BAR 4/5.

```c
enum ntb_bar_t {
    NTB_BAR_23    = 0x00010000,
    NTB_BAR_45    = 0x00020000
};
```

The BDF value can be created by bitwise OR-ing the bus value into the first 8 bits of a 16-bit value, the function value into the next 5 bits, and the device value into the last 3 bits.

Sample:

```c
/* Bus = 0, Device = 3, Function = 0 */
unsigned char bus = 0x=00;
unsigned char devfn = 0x18;

uint16_t bdf = bus;
bdf = bdf << SHIFT_8;
bdf = bdf | devfn;
```

The client driver’s callback function contains doorbell and scratchpad register parameters for monitoring events.

Sample:

```c
void ntbc_callback(int16_t doorbell_value, struct scratchpad_registers pad);
```

The ntb_register_client API returns a handle that the client uses to pass to other NTB APIs as a unique identifier. If the registration was successful, a value greater than 0 is returned.

Optionally, the client driver can supply a limit and translate address in the last parameter or use the API calls to set those values.

Sample:

```c
/* Register client and request use of BAR 2/3 associated with PROC 0 */
int32_t ret_val;
ntb_client_handle_t handle = 0;
int16_t bdf = 0x0018;
handle = ntb_funcs.ntb_register_client(
    NTB_BAR_23,
    &ntbc_callback, bdf, NULL);
```
if (handle <= 0) {
    printk("Failure to obtain BAR 2-3 \n");
}

To unregister, the client calls the ntb_unregister_client API. Only one client can own a particular BAR; however, a single client may own both BARs. Clients must unregister to free the BAR for other clients to use.

Sample:

ntb_funcs.ntb_unregister_client(handle_test0);

6.4 Writing to Translate Address and Limit Registers

To write the translate address and limit registers values, the client driver uses the ntb_register_client API during registration or separate APIs designed to write to those registers after registration has occurred. The address supplied to the translate register must be a physical memory address.

Sample:

/* During client registration */

struct ntb_client_data client_data;

int32_t ret_val;

client_data.limit = LIMIT_VALUE;
client_data.translate_address = physical_address;
ntb_client_handle_t handle = 0;
handle = ntb_funcs.ntb_register_client(
    NTB_BAR_23,
    &ntbc_callback, bdf, &client_data);
if (handle <= 0) {
    printk("Failure to obtain BAR 2-3 \n");
}

Sample:

/* Using the APIs */

uint64_t physical_address = get_physical_address();
uint64_t limit = LIMIT_VALUE;
int32_t ret_translate = 0;
int32_t ret_limit = 0;
ret_translate = ntb_funcs.ntb_write_translate_address_value(handle, physical_address);
ret_limit = ntb_funcs.ntb_write_limit(handle, physical_address);

/* The client should then check API return values to determine if the
 * writes were successful. */

6.5 Writing to Doorbell and Scratchpad Registers

Client drivers on opposite subsystems use the 16-bit doorbell registers to implement communication protocols. Writing to the doorbell register with the API ntb_write_doorbell causes an interrupt to occur in the NTB driver and a subsequent invocation of a client’s callback routine.

Bits 0-13 of the doorbell register are generic, available for use by client drivers. The client driver creators implement a protocol according to application requirements using these bits. Bit 14 is reserved for write cache control events. Bit 15 is reserved for link state change management by the NTB driver. The client driver receives callbacks for these events if a callback routine has been supplied.

Sample:

/* Client Driver on Sub-system One: Clients decide which doorbell bits to use for communication. */
#define NTB_SC_DATA_AVAILABLE_23 0x02
uint16_t doorbell_value = NTB_SC_DATA_AVAILABLE_23;
int32_t ret = 0;
ret = ntb_funcs.ntb_write_doorbell(handle, doorbell_value);
if (ret != 0) {
    printk("Failure to write to doorbell register\n");
}

/* Client Driver on Sub-system Two*/
#define NTB_SC_DATA_ACKNOWLEDGEMENT_23 0x04
uint16_t doorbell_value = NTB_SC_DATA_ACKNOWLEDGEMENT_23;
int32_t ret = 0;
ret = ntb_funcs.ntb_write_doorbell(handle, doorbell_value);
if (ret != 0) {
    printk("Failure to write to doorbell register\n");
}
The scratchpad registers offer another means of cross-subsystem communication for client drivers. A client should obtain a semaphore before attempting to write to scratchpad registers as they are shared between subsystems.

Sample:

```c
do {
    err = ntb_funcs.ntb_obtain_semaphore(handle);
    if (timeout == CRYPTO_TIMEOUT)
        return -EAGAIN;
    else
        timeout++;
} while (err != SUCCESS);
```

After obtaining a semaphore, the client uses the ntb_write_scratch_pad_many or ntb_write_scratch_pad_one APIs to scratchpad registers. Clients write to these registers according to the communication protocol shared between clients on opposite subsystems.

Sample:

```c
struct scratchpad_registers pad;
pad.registers[0] = DATA_SIZE;
pad.registers[1] = DATA_TYPE;
uint16_t doorbell_value = NTB_SC_DATA_AVAILABLE_23;
int32_t ret = 0;
int32_t no_regs = 2;
ntb_funcs.ntb_write_scratch_pad_many(no_regs, &pad, handle);
ret = ntb_funcs.ntb_write_doorbell(handle,
    doorbell_value);
```

### 6.6 Add, Reset, Get Policy

The ntb_add_policy API appoints the bit allocations of the NTB doorbell register for the client drivers based on a client’s protocol. For example, in a given protocol, a client has allocated bit 0 for heartbeat messages.

Sample:

```c
uint16_t heartbeat_bit = 0x0001;
uint16_t client_bits = 0x000E;
ret = ntb_funcs.ntb_add_policy(handle,
    heartbeat_bit, client_bits,
    0,
```
When the doorbell receives a write to bit 1, it will notify the client associated with the BAR 2/3 that it has received a write.

The ntb_reset_policy API resets all of the bits for a particular client, while the ntb_get_policy API returns all of bits.

Sample:
```c
ret = ntb_funcs.ntb_reset_policy(handle);
```

Sample:
```c
uint16_t policy = 0x0000;
policy = ntb_funcs.ntb_get_policy(handle);
```

### 6.7 Get BAR Address

The ntb_get_bar_address API returns a physical address to BAR 2/3 or BAR 4/5, depending on the BAR associated with the given registered client. The client uses this address to write data to memory locations pointed to by the BAR.

Sample:
```c
uint64_t ptr_bar_23;
ptr_bar_23 = ntb_funcs.ntb_get_bar_address(handle, NTB_BAR_23);
```

The client can then map a physical address to a virtual address and write data to the location.

### 6.8 Read/Write to Remote Limit

The ntb_read_remote_limit and ntb_write_remote_limit APIs are available for the B2B and classic NTB configurations. The ntb_get_link_status is called before using the remote limit APIs to refresh stored secondary BAR values for BARs 2/3 and 4/5.

Sample:
```c
uint64_t remote_limit;
ntb_table->ntb_get_link_status(handle);
if(link_status != LINK_UP) {
    remote_limit = ntb_funcs.ntb_read_remote_limit(handle, NTB_BAR_23);
}
```