Linux Device Driver Architecture for PES24NT3/PES12NT3/ PES16NT2/PES8NT2

Application Note AN-705

Notes

By Alex Chang

Overview

This Linux device driver is implemented for the endpoints of IDT inter-domain PCIe switches, PES24NT3/12NT3/16NT2/8NT2 (referred as PES24NT3). This document explains how the driver is designed and implemented to work with the latest released Linux kernels. The driver has been tested with the following Fedora Core releases from Redhat:

Fedora Cores	Kernel releases	32/64 bit OS	32/64 bit BARs
Fedora 3	2.6.9-1.667	32 bit	32 bit
Fedora 6	2.6.18-1.2798	32 and 64 bit	32 bit
Fedora 7	2.6.21-1.3194	32 and 64 bit	32 bit
Fedora 8	2.6.23.1-42	32 and 64 bit	32 bit
Fedora 9	2.6.25-14	32 and 64 bit	32 and 64 bit

Table 1 Supported Linux Platforms

References

89HPES24NT3 User Manual

PCI Express Base Specification Revision 1.0a

Linux source code

pci.txt under Linux source tree

Application Note AN-510, Usage of Non-transparent Bridging with IDT PCI Express NTB Switches by Kwok Kong, January 23, 2007

Limitations

PCIe Spread Spectrum Clock needs to be disabled via system BIOS.

For 64-bit BAR configuration, the driver works only when the system assigns a BAR base address beyond the 4 GB boundary and when the driver can allocate local buffers with associated physical addresses beyond the 4 GB boundary.

Abbreviations

PCIe: PCI Express

NTB: Non-Transparent Bridge

EP: NTB endpoint device

BAR: Base Address Register

IPC: Inter-Processor Communication

DMA: Direct Memory Access

DSID: Device Service ID

EEPROM: Electrically Erasable Programmable Read-Only Memory

Notes

MTU: Maximum Transfer Unit

IOAT: I/O Acceleration Technology

NTB Connections

The switch functional block diagram of a PES24NT3 is illustrated in Figure 1.

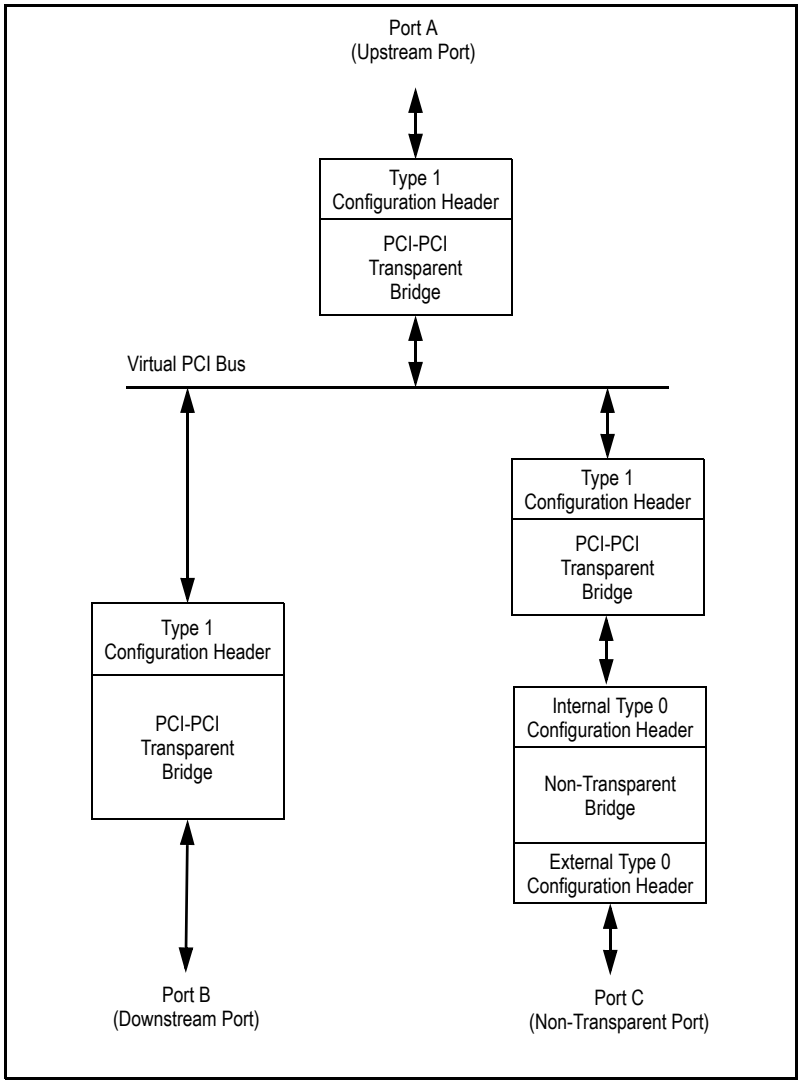

Figure 1 PES24NT3 Functional Block Diagram

There are two PCI endpoints on both internal and external sides of the NTB. These endpoints serve as communication windows between two Root Complexes when Port A and Port C are connected to two separate Root Complexes.

With the Punch-through feature, the PES24NT3 can support two different connections in NTB mode:

- Single NTB Port Connection: Single NTB port connecting two Root Complexes.
- Dual NTB Port Connection: Two NTB ports connecting two Root Complexes.

Notes

Single NTB Port Connection

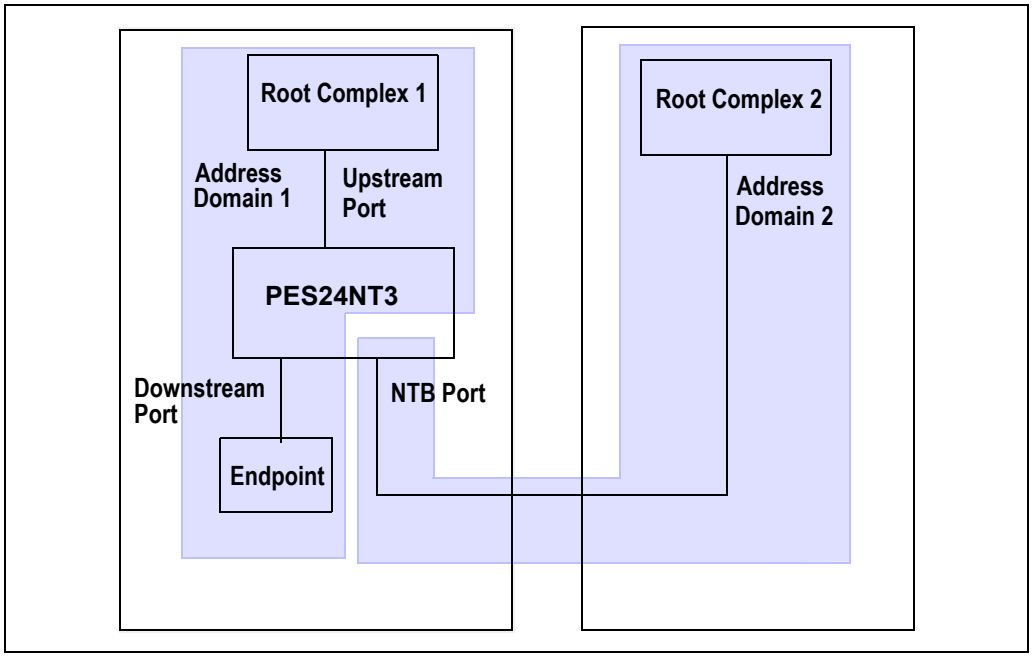

Figure 2 Single NTB Port Connection

With a single NTB port connection, both Root Complexes can communicate with each other through a number of facilities implemented in the NTB endpoints:

- Inter-processor Communication (IPC) registers: Both internal and external endpoints can configure their IPC registers, such as Doorbell registers, to generate interrupts on the other side to draw attention or make requests.
- Base Address Registers (BAR): There are five BARs available for address-routed transactions. BAR[0..3] may each be used to map 32- or 64- bit memory or I/O windows between the internal and external sides of the NTB. BAR4 can only be used to map the configuration space of the NTB endpoint.
- Mapping Table registers: These registers contain valid Bus, Device, and Function number for ID-routed transactions between both sides of the NTB.

Notes

Dual NTB Ports Connection

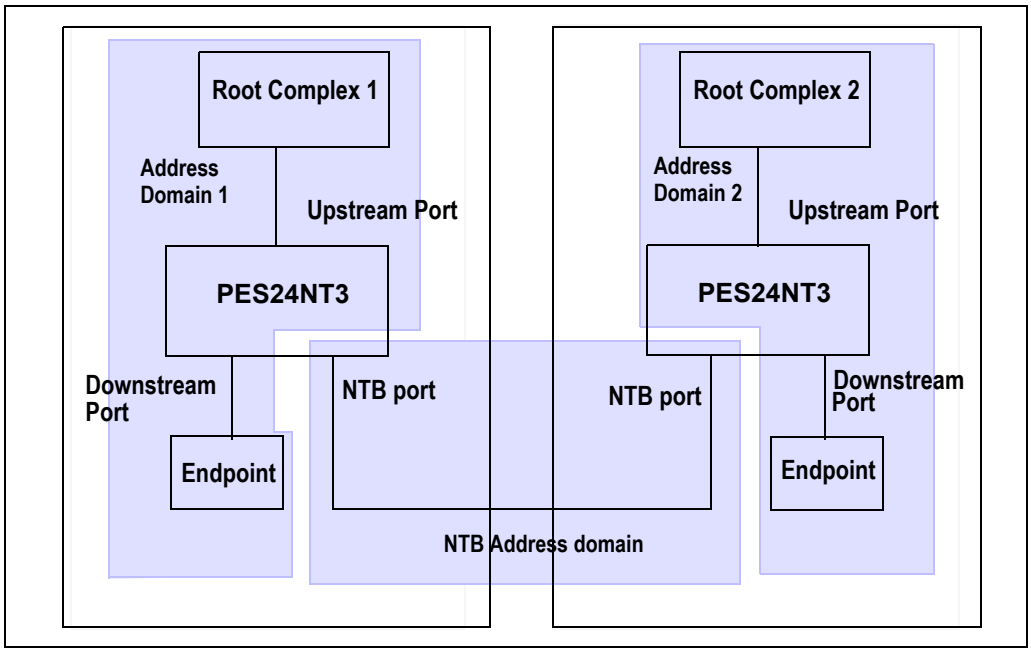

Figure 3 Dual NTB Port Connection

With a dual NTB ports connection, both external endpoints are connected together. In addition to the facilities mentioned in the previous section, the connection requires the Punch-through feature to achieve inter-processor communications. The Punch-through feature ensures both internal endpoints are capable of generating configuration transactions on the external sides of the remote NTB ports, such that the local internal endpoint can communicate with the remote internal endpoint of the other NTB port. Whenever a local internal endpoint intends to interrupt a remote internal endpoint, it programs the remote external endpoint's Outbound Doorbell register via the Punch-through registers. For more details regarding this feature, please refer to Chapter 9 of the PES24NT3 User Manual.

PES24NT3 Device Driver Design

Version 2 of PES24NT3 Linux NTB endpoint device driver is divided into two layers: Function and Base Layers. The Base Layer initializes/de-initializes driver components, interacts with the NTB endpoints, and provides services to Function Layer drivers to complete data transfers. The Function Layer includes a virtual Ethernet driver and a Raw Packet transfer driver (additional details are provided in section Function Layer on page 5).

Driver Architecture

The PES24NT3 NTB endpoint device driver is built as a Linux kernel mode module. The included Function Layer is meant to provide the device driver function to the OS. There are two function services:

- Ethernet Function Service: provides a virtual Ethernet interface that allows Ethernet packets to be exchanged between two hosts connected to the PES24NT3 switch.
- Raw Packet Function Service: directly transfers raw packets from/to PF_PACKET sockets associated with a network interface without any protocol stack processing involved.

Notes

When the driver is loaded, the Base Layer registers with the Linux kernel as an endpoint device driver after successfully allocating the required resources and initializing all necessary sub-modules. The following sub-modules are implemented in the Base Layer driver:

- Init/De-init: module initialization/de-initialization.
- IPC: inter-processor communication.
- Function Service: function service interface.
- Buffer Queuing: software queue management for buffers.
- Tx/Rx: transmit/receive interface.
- DMA: packet transfer via DMA/MTRR.
- MM: memory management interface.
- HPR: hardware platform specific routines.

Figure 4 illustrates the block diagram of both Function and Base Layers implemented in the PES24NT3 device driver. Within the Base Layer, the green sub-modules are designed to interact with the Function Layer drivers while yellow sub-modules are transparent to them. The design details of these blocks and how they interact are discussed in the following sections.

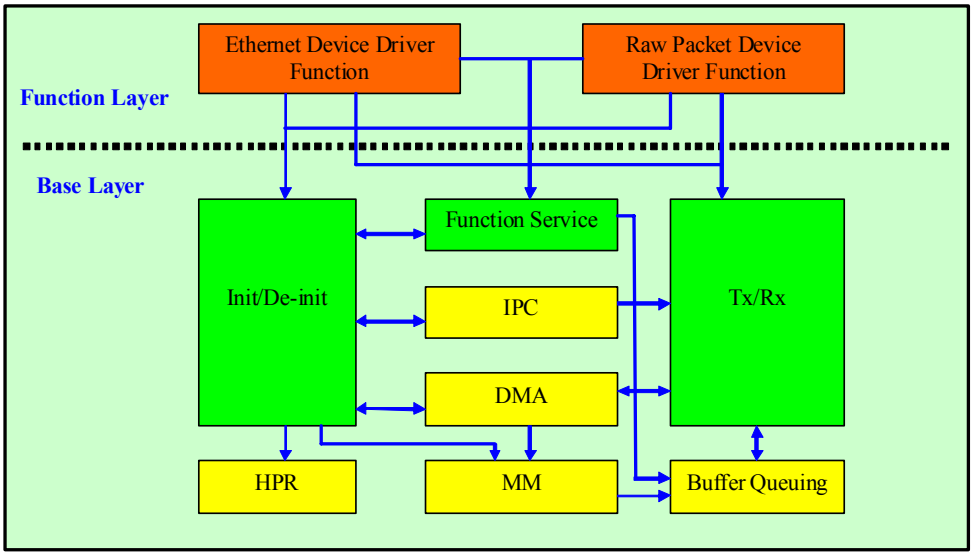

Figure 4 PES24NT3 NTB Endpoint Device Driver Block Diagram

Function Layer

Each function in this layer can be included or excluded independently while compiling the driver. Both Ethernet and Raw Packet device drivers are included by default. However, while compiling the driver, specifying "NONET=1" or "NORAW=1" can exclude the Ethernet or Raw Packet driver respectively. Once included, when the NTB endpoint device driver is loaded with the "insmod" command, it calls their associated initialization routines after all necessary resources are allocated and initialized successfully. When the driver is unloaded with the "rmmod" command, their associated de-initialized routines are called to completely unload the function service driver(s) from the system.

There are four module parameters (shown in Table 2) set up to modify the default sizes of I/O memory and buffer for Ethernet and Raw packet drivers. When the NTB endpoint device driver is being loaded, the module parameters may be overridden only if the specified I/O memory size and buffers size does not exceed the allocated BAR memory size set up via EEPROM. For example, "insmod idt-ntb.ko eth_mem=512 eth_buf=2" requests to have only 512 KB memory for the Ethernet driver, and each transfer buffer is 2 KB in length.

Notes

	I/O Memory Size		Buffer Size	
Ethernet Driver	Module Parameter	Default Size (KB)	Module Parameter	Default Size (KB)
	eth_mem	1024	eth_buf	18
Raw Packet Driver	Module Parameter	Default Size (KB)	Module Parameter	Default Size (KB)
	raw_mem	1024	raw_buf	18

Table 2 Module Parameters for I/O Memory and Buffer Size

Table 2 details these parameters for the Ethernet and Raw Packet drivers. When users try to override the above default sizes, they need to consider the following:

- The default buffer size is used to determine MTU for the Ethernet and Raw Packet driver. By default, the MTU is 18KB - 64 Bytes (assumed max packet header size).
- For some applications which always transfer smaller packet sizes, such as Smartbit applications, the default buffer size can be reduced, resulting in more buffers being made available for data transfers and better throughput.
- The default I/O memory size is set up via EEPOM. If the user attempts to override with a larger size than the default size, the default I/O memory size will be used.
- The limitation of the size of I/O memory is system-dependent.

Ethernet Device Driver Function

After the Base Layer driver is loaded, the Ethernet driver's initialization routine, pes24n3_net_init, is called to start out the necessary initialization process. When unloaded, its module exit routine, pes24n3_net_exit, gets called to remove itself from the system.

Module Initialization

1. Call register_netdev to register as an Ethernet device driver.
2. Call idt_ntb_func_alloc to allocate an idt_ntb_func data structure and initialize its members.
3. Call idt_ntb_func_regiser to register as a function service with Base Layer driver.
4. Call idt_ntb_alloc_bar and idt_ntb_alloc_doorbell to reserve a dedicated BAR memory and a door-bell bit for later data transfer.

Module De-initialization

The module de-initialization process is exactly reversing the steps of module initialization:

1. Call idt_ntb_free_bar and idt_ntb_free_doorbell to release the resources.
2. Call idt_ntb_func_unregiser to remove itself form function service list.
3. Call idt_ntb_func_free to free up function service data structure.
4. Call unregister_netdev to un-register virtual Ethernet device interface.
5. Call free_netdev to de-allocate Ethernet device data structure.

Other Core Functions

net_open: This function is invoked when the Ethernet interface is activated. It calls netif_start_queue to enable packet transfer.

net_stop: This function is invoked when the Ethernet interface is being removed. It calls netif_stop_queue to halt data transfers associated with the interface.

net_tx: This function is called to transmit packets out of the Ethernet interface.

net_rx: This function is called by the Base Layer driver when it receives a packet and identifies it as an Ethernet packet via DSID of idt_ntb_func. Then skb_buff is allocated to save the packet data and call netif_rx to pass up the received packet.

Notes

Raw Packet Device Driver Function

The Raw Packet driver works similarly as the Ethernet driver discussed in the previous section. Both share __net_data structure and some member functions of net_device structure, such as net_open, net_stop, net_tx and net_rx, etc. However, the Raw Packet driver is used to transfer raw packets via PF_PACKET sockets. Applications such as RawX (included in NTB device driver release) create PF_PACKET sockets and exchange raw packets with a network interface directly without involving any protocol stack processing.

The Raw Packet driver sets up the network interface to transfer and receive raw packets. Its module initialization routine, pes24n3_raw_init, works the same as pes24n3_net_init and its module exit routine, pes24n3_raw_exit, does similar cleanups as pes24n3_net_exit.

Base Layer

The major components of the NTB endpoint device driver are implemented in the Base Layer driver. As illustrated in Figure 4, the Base Layer driver is divided into several sub-modules. The detail design of each sub-module is elaborated in the following sub-sections.

Init/De-init

This sub-module covers all the driver/device initialization and de-initialization related tasks:

Module initialization routine: pes24n3_ntb_init.

Module de-initialization routine: pes24n3_ntb_exit.

All PCI driver required routines: pes24n3_ntb_probe, pes24n3_ntb_remove, pes24n3_ntb_resume, etc.

Device initialization routine: pes24n3_dev_init.

Device de-initialization routine: pes24n3_dev_exit.

Pes24n3_ntb_init

This function gets called right after the NTB endpoint driver is loaded. It calls pci_register_driver with a populated pci_driver data structure to register itself with the system.

Pes24n3_ntb_exit

This function gets called when the driver is removed from the system. It calls pci_unregister_driver with pci_driver data structure to un-register itself from the system.

PCI Driver Routines

The PCI driver routines are specified and populated in the pci_driver data structure. Also, all the supported NTB endpoint devices and IDT vendor IDs are disclosed in pci_driver.

pes24n3_ntb_probe: This function is called by the kernel after it has identified the presence of a device whose vendor/device IDs match any entry of the disclosed ID table of pci_driver. All device-related initialization starts from here.

pes24n3_ntb_remove: This function is called when the driver is removed from the system. All the device-related de-initialization is handled here.

pes24n3_dev_init

This function is called by pes24n3_ntb_probe to start all the device-related initialization:

Call pci_enable_device to enable the PCI device.

Call pci_set_master to enable PCI bus mastering.

Call pci_request_regions to request regions.

Call ioremap_nocache to map PCI configuration space onto BAR4.

Check to see whether it's single or dual NTB port connection.

Call pes24n3_plat_init to complete platform specific tasks.

Populate mapping table entries.

Set up interrupt and its ISR.

Notes

Call pes24n3_mem_init to initialize BAR related memory.

Call pes24n3_net_init if the Ethernet driver is included.

Call pes24n3_raw_init if the Raw Packet driver is included.

Call idt_ntb_ipc_init to initialize IPC related tasks.

Enable interrupt and get ready to communicate with the other endpoint of NTB.

Now the main control shift to IPC sub-module and all activities are interrupt-driven.

pes24n3_dev_exit

This function is called by pes24n3_ntb_remove to complete all the device related de-initialization:

Call pes24n3_net_exit if the Ethernet driver was initialized.

Call pes24n3_raw_exit if the Raw Packet driver was initialized.

Call pes24n3_ipc_exit to notify the other endpoint with IPC_CMD_DOWN, disable interrupt and free up resources.

Call pes24n3_mem_exit to release memory resources.

Call iounmap to un-map BAR4.

Call pci_release_regions.

Call pci_disable_device.

Function Service

This sub-module provides major APIs to the Function Layer drivers. When the Function Layer driver initializes, it calls idt_ntb_func_alloc to allocate its own idt_ntb_func data structure. After populating the structure, it calls idt_ntb_func_register to register itself with the Base Layer driver. When the Function Layer driver is unloaded, it calls idt_ntb_func_free and idt_ntb_func_unregister to remove itself from the function service list of the Base Layer driver.

Other available APIs are also provided:

- int idt_ntb_alloc_bar(idt_ntb_ep* ep, idt_ntb_func* func, unsigned bar, unsigned outbound)

  Any Function Layer driver can reserve which BAR will serve as the incoming or outgoing data transfer window via parameter outbound after registering with the Base Layer driver. If the specified bar is being used, the Base Layer driver will assign another available BAR instead. When the Function Layer driver sends out IPC_CMD_HELLO to another endpoint, it indicates the assigned BAR in bit[0..15] of MSG1 to tell the other endpoint using the BAR to send data to itself.
- int idt_ntb_free_bar(idt_ntb_ep* ep, idt_ntb_func* func, unsigned outbound)

  The Function Layer driver needs to call this function to release the reserved BAR when the driver is removed from the system. Then the Base Layer driver can assign it to other Function Layer drivers upon request.
- int idt_ntb_alloc_doorbell(idt_ntb_ep* ep, idt_ntb_func* func, unsigned db_bit)

  The Function Layer driver can reserve an INDBELL bit for its incoming data transfer by calling this function. If the specified db_bit is not available, an available bit will be granted by the Base Layer driver. When the Function Layer driver sends out IPC_CMD_HELLO to the other endpoint, it indicates the assigned bit in bit[16..31] of MSG1 to tell the other endpoint using the bit to interrupt itself.
- int idt_ntb_free_doorbell(idt_ntb_ep* ep, idt_ntb_func* func)

  The Function Layer driver needs to call this function to release the reserved INDBELL bit when the driver is removed from the system. Then the Base Layer driver can assign it to other Function Layer drivers upon request.
- idt_ntb_frame* idt_ntb_frame_alloc(u32 frags, u32 priv_len, idt_ntb_frame_ds ds)

  The Function Layer driver needs to call this function to allocate a packet frame for data transfer. Usually, it needs one fragment and zero private data length. The destruction routine, ds, needs to be specified. This routine gets called to free skb buffer by calling dev_kfree_skb after the transfer is completed.

Notes

The Base Layer driver also calls this function to allocate a packet frame when receiving a packet from the other endpoint. The priv_len is used to indicate the size of structure idt_ntb_rx instead.

– void idt_ntb_frame_free(idt_ntb_frame* frame)

When receiving a packet from the other endpoint, the Function Layer driver first passes the skb up by calling netif_rx, and then the driver uses this function to free the packet frame allocated by the Base Layer driver. Again, this function also gets called by the Base Layer driver to free the packet frame after finishing the data transfer by ringing OUTDBELL bit to notify the other endpoint of the just sent packet.

Tx/Rx

This sub-module contains the core functions required for transfer/receive operations. These functions are responsible for handling data frame transfer per requests from the Function Service sub-module and the pending received data frame in PostQueue.

Transferring Packets

The Function Layer driver calls idt_ntb_frame_send to transfer data after it successfully allocates memory for idt_ntb_frame data structure and populates its data members, such as the location and length of the source buffer.

If the return value of idt_ntb_frame_send is negative, the packet is dropped and the memory of idt_ntb_frame is freed. If it's positive, the Function Layer driver needs to notify the application to temporarily halt a data transfer and to set up a timer to transfer the current packet at a later time by calling netif_wake_queue. Otherwise, the transfer is completed.

A callback function, tx_cb, is passed while calling the data transfer function, idt_ntb_dma_L2P. After data transfer completes, tx_cb is called to insert the data packet to the PostQueue of the remote endpoint, ring the OUTDBELL bit to interrupt the endpoint, and free the memory of idt_ntb_frame data structure.

Receiving Packets

When the current IPC state is STATE_OK and the local endpoint is interrupted via INDBELL bit[1..31], it indicates the remote endpoint is transferring a packet over. The current thread calls idt_ntb_frame_receive to determine if there is any pending receiving buffer in local PostQueue.

If any pending buffer is found, call rx to set up idt_ntb_frame structure for the received packet.

Call the receiving callback function of Function Layer driver to pass the idt_ntb_frame structure up. The Function Layer driver needs to request a destination buffer, save it into idt_ntb_frame structure, and call rx_frame_sync with a callback function to start the data transfer via DMA or memory copy.

After data transfer completes, the callback function gets called to free the memory of idt_ntb_frame and release the destination buffer.

IPC

When IPC sub-module is initialized, the following tasks need to be completed:

- Initialize tasklet routine, idt_ntb_tasklet_proc.
- Initialize resources, such as spin locks, timer, incoming/outgoing ipc list, etc.
- Register interrupt handler, idt_ntb_ipc_isr.
- Enable interrupt.
- Schedule first tasklet to kick off IPC.

When de-initialized, it does the following:

- Call ipc_down to notify the other endpoint with IPC_CMD_DOWN.
- Disable interrupt.
- De-initialize the above resources.
- Un-register interrupt via free_irq.
- De-initialize tasklet routine.

Notes

More details regarding IPC protocols and usage of IPC registers can be found in section IPC Protocols and Initialization on page 13.

Buffer Queuing

This sub-module serves as a buffer queuing manager that is transparent to the Function Layer drivers. The basic idea is, during initialization, the local CPU adds the addresses of its local receive buffers to the local freeQ, which is initialized as remote freeQ by the remote CPU when it receives IPC_CMD_HELLO, indicating where the freeQ is. Figure 5 details the layout of the initialized queue structure in memory.

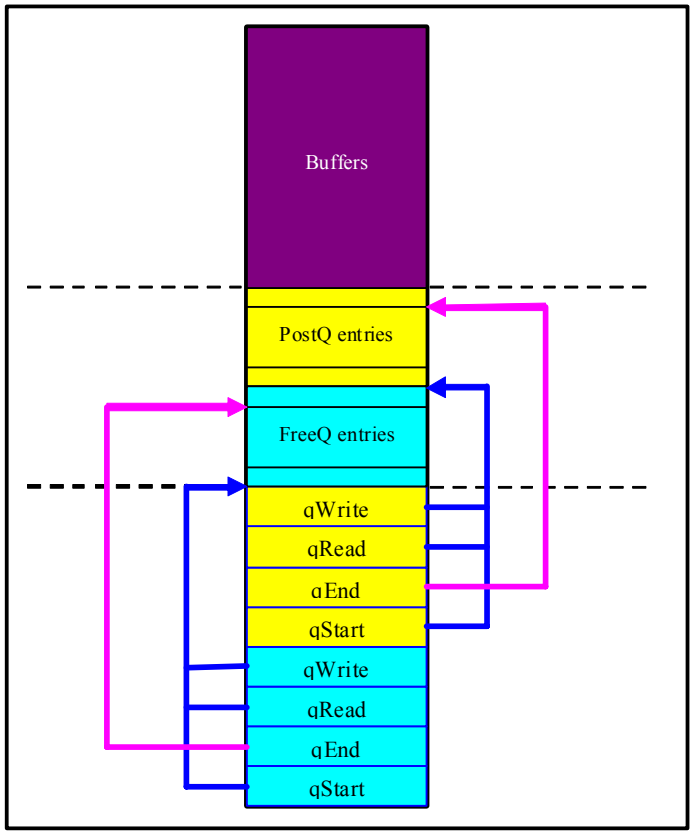

Figure 5 Queue Structure in Memory

When a remote endpoint needs to send a packet to a local endpoint, the following steps are required:

- The remote endpoint reads the buffer address and removes buffer from the remote FreeQ associated with the local endpoint for transferring data.
- Transfer the data into the buffer.
- After completion, the remote endpoint adds the buffer address to the local PostQ of the local endpoint.
- The remote endpoint rings the appropriate doorbell bit to interrupt the local endpoint.
- The local endpoint then reads the pending buffer address from its local PostQ to retrieve the data.
- After processing the data, the local endpoint adds the buffer back to local FreeQ.

Notes

Figure 6 depicts the above scenario.

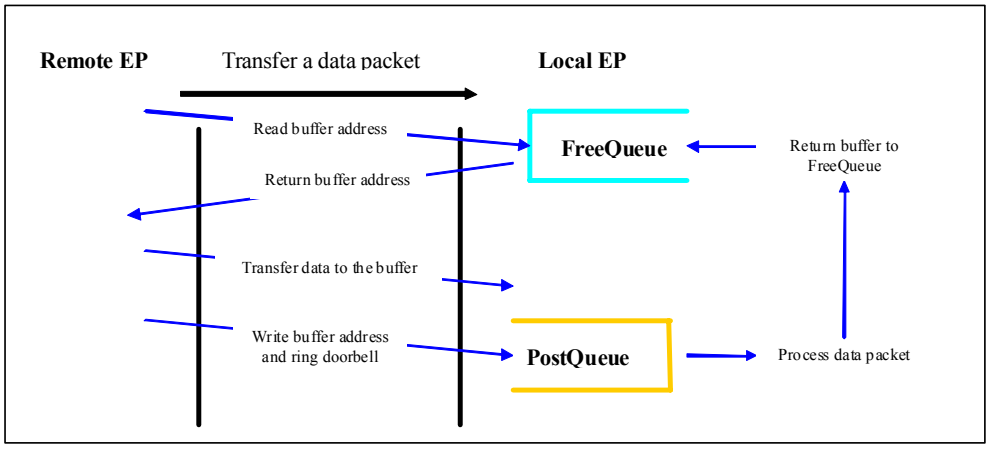

Figure 6 Data Transfer with Software Queuing

DMA

This sub-module provides the interface between the IOAT DMA driver and Tx/Rx when DMA transfer is available. If DMA is not available, packets are transferred via Write-combining MTRR regions set up by the MM sub-module. A compiling flag called IOATDMA is set up for enabling the interface to the IOAT DMA driver. By default, IOATDMA is 1 and it can be disabled with "make IOATDMA=0".

If the IDT-provided IOAT DMA drivers are not loaded first, loading the NTB endpoint driver with IOATDMA=1 results in an error like "Unknown symbol in module" due to the non-existing exported functions. Functions referred by this sub-module with prefix "mp_dma_async_" and "ioat_dma_" are exported from the IOAT DMA drivers. The interaction between the NTB endpoint driver and IOAT DMA drivers can be separated into two parts: Initialization/de-initialization and data transfer.

The chipsets used to test DMA transfers are Intel's Bensley and Stoakley. They are all equipped with an IOAT DMA engine to accelerate data transfer for I/O devices. Please note that the original IOAT DMA drivers from Intel have been modified to work with IDT's NTB endpoint drivers.

Initialization/de-initialization

The initialization routine, mp_dma_register, calls mp_dma_async_client_register with a callback function dma_event to register as a DMA client.

If registration succeeds, the routine then calls mp_dma_async_client_chan_request to request 2 DMA channels, one for local to local memory transfer when receiving packets, and the other for local to PCI I/O memory transfer when transmitting packets.

The callback function, dma_event, needs to be called to receive the granted DMA channels. Asynchronous DMA data transfer is employed only after the requested 2 channels are granted.

If any failure happens above, data transfer will fall back to use MTRR regions.

The de-initialization routine, mp_dma_unregister, calls mp_dma_async_client_unregister when the NTB endpoint driver gets unloaded.

Data Transfer

When transmitting a packet, dma_start_ioatdma gets called to move data from local to PCI I/O memory. When receiving a packet, dma_start_ioatdma gets called to move data from local buffer to the destination buffer provided by Function Layer driver.

Before transferring data, dma_start_ioatdma calls idt_ntb_dma_alloc to allocate memory for idt_ntb_dma data structure and to set up necessary information, such as the callback function for cleaning up and the data structure to pass into the callback function.

Notes

The granted DMA channel 0 is for local to local buffer transfer, while channel 1 is for local to PCI I/O memory transfer.

For local to local transfer, call the exported ioat_dma_memcpy_buf_to_buf_cb function from IOAT DMA drivers to do asynchronous DMA transfer. If the return value is not negative, current thread will move on to engage next transfer. Similarly, the exported ioat_dma_memcpy_buf_to_io_cb function gets called to process local to PCI I/O memory transfer.

When the DMA transfer completes, idt_ntb_dma_free gets called to free up the idt_ntb_dma data structure.

MM (Memory Management)

This sub-module is responsible for initializing local and remote buffers for data transfer. If the given BAR is not set up via EEPROM, it won't be initialized. Its initialization routine, pes24n3_mem_init, is called by pes24n3_dev_init while the de-initialization routine, pes24n3_mem_exit, is called by pes24n3_dev_exit to release the memory resources. Its major functions include:

- int local_mem_init(idt_ntb_ep* ep, unsigned map, unsigned long size)
  Allocates local memory for an incoming data buffer. The local memory allocation would be skipped if the specified BAR of the other endpoint is not set up with an assigned physical address. The valid map values are from 0 to 3 for 32-bit BARs and 0 or 2 for 64-bit BARs.
- local_mem_exit(idt_ntb_ep* ep, unsigned map)
  Free the previously allocated local memory.
- int remote_mem_init(idt_ntb_ep* ep, unsigned map, unsigned long size)
  Re-map the specified BAR memory region into virtual memory space with ioremap_noncache. The memory region serves as an out-going buffer memory to the other endpoint. This region will be added to Write-combining MTRR region to accelerate data transfer.
- void remote_mem_exit(idt_ntb_ep* ep, unsigned map)
  Before un-mapping the previously mapped BAR memory region with iounmap, it removes this region from MTRR region.

HPR (Hardware Platform Routines)

This sub-module includes all known hardware platform-specific routines that must be executed when the NTB endpoint driver is loaded. Its main function, pes24n3_plat_init, handles Mapping Table setup and some NTB endpoint register setup for certain motherboards or chipsets. In addition, for x86 systems, the memory allocation (pes24n3_local_malloc) and de-allocation (pes24n3_local_mfree) APIs are also included.

Users may add any hardware-specific implementation which is required by them to make the driver work with their hardware. For example, if their system has special entries to add into Mapping Table of the NTB endpoint, they should append them in this sub-module.

Notes

IPC Protocols and Initialization

IPC Protocols

To setup memory mappings and coordinate data transfers between NTB endpoints, the device drivers running on both sides of NTB must adhere to the following protocol:

- After module initialization processes successfully complete in the Base Layer driver, the driver needs to start the IPC initialization by sending the IPC_CMD_START command.
- The Base Layer driver manages all the allocated memories physically located on its side of NTB.
- To access a memory region physically residing on the other side of the NTB, the Base Layer driver must request a buffer from the memory region associated with the other side of NTB using an IPC_CMD_MAP command.
- The Base Layer driver disables all message interrupt generations. The OUTMSG registers are used for sending requests and replies to the other side of NTB. In the case of a request, OUTMSG0 is used for identifying the desired service, while OUTMSG1, OUTMSG2, and OUTMSG3 are used for passing parameters associated with the request. In the case of a reply, OUTMSG0 is used for identifying the request associated with this reply, while OUTMSG1, OUTMSG2, and OUTMSG3 are used for returning results associated with the reply. The usage and interpretation of the OUTMSG1, OUTMSG2, and OUTMSG3 depend on whether the content of the OUTMSG0 is a request or a reply.
- The Base Layer driver triggers doorbell interrupt generation. The OUTDBELL registers are used for generating interrupts on the other side of NTB to signal new events. Bits 0 and 1 of OUTDBELL are defined as VALID and DONE bits, respectively. The driver sets the VALID bit in the OUTDBELL register to signal a new request or reply to the other side of NTB after updating the OUTMSG registers. The driver sets the DONE bit in the OUTDBELL register to acknowledge the reception of a request or reply from the other side of NTB after reading the INMSG registers. Other bits in the Doorbell registers may be reserved and used by Function Layer drivers.

The IPC Doorbell register usage is shown in Table 3. The Base Layer driver uses bits 0 and 1 only.

Field Names	Bits	Definitions
VALID	0	Signal a new request or reply
DONE	1	Acknowledge the reception of a request or reply
Allocated dynamically	2-31	To be assigned to the Function Layer drivers.

Table 3 IPC Doorbell Register Definitions

The IPC MSG0 register definition is shown in Tables 4 and 5.

Field Names	Bits	Definitions
Tag#	0 – 7	Used for matching reply to request
DSID	8 – 15	Driver Service ID. For Base Layer, always specify it as 0. 0 = Base device driver, 1 = Ethernet device driver function, 2 = Raw Packet device driver function. Others are reserved for other function services.

Table 4 IPC MSG0 Register Definitions for Base Layer Driver (Page 1 of 2)

Notes

CMD	16 – 23	Used for identifying the type of requests or replies. 128 = IPC_CMD_START 1 = IPC_CMD_MAP 2 = IPC_CMD_OK 3 = IPC_CMD_DOWN Others are reserved.
MAP	24 – 26	Used to identify which mapping region: 0 = 32-bit mapping associated with BAR0. 1 = 32-bit mapping associated with BAR1. 2 = 32-bit mapping associated with BAR2. 3 = 32-bit mapping associated with BAR3. 0 = 64-bit mapping associated with BAR[0..1] 2 = 64-bit mapping associated with BAR[2..3]. Others are reserved.
REPLY	27	Set to 0 for requests and 1 for replies
STATUS	28 – 31	Return the status in a reply. 0 = IPC_STS_OK, successful completion of the request. 1 = IPC_STS_NR, IPC is not ready. 2 = IPC_STS_MAP, mapping error. 3 = IPC_STS_OOB, out of bound error 4 = IPC_STS_NS, request is not supported. 15 = IPC_STS_UK, unknown error. Others are reserved.

Table 4 IPC MSG0 Register Definitions for Base Layer Driver (Page 2 of 2)

Field Names	Bits	Definitions
Tag#	0 – 7	Used for matching reply to request
DSID	8 – 15	Driver Service ID. This identifies which Function Layer driver owns the IPC message. Currently defined Driver Service ID for Function Layer drivers: 1 = Ethernet device driver function. 2 = Raw Packet device driver Function. Others are reserved for other function services.
CMD	16 – 23	Used to identify the type of requests or replies. 8 = IPC_CMD_HELLO, IPC packet to notify my peer that I am ready to communicate, which Doorbell bit and BAR to use, etc. Others are reserved.
Reserved	24 – 26	N/A
REPLY	27	Set to 0 for requests and 1 for replies
STATUS	28 – 31	Return the status in a reply. 0 = IPC_STS_OK, successful completion of the request. 1 = IPC_STS_NR, IPC is not ready. 4 = IPC_STS_NS, request is not supported. 15 = IPC_STS_UK, unknown error. Others are reserved.

Table 5 IPC MSG0 Register Definitions for Function Layer Drivers

Notes

The use of IPC MSG registers associated with IPC commands is shown in Table 6. The commands IPC_CMD_MAP and IPC_CMD_HELLO carry additional data in Message registers other than MSG0.

CMD field of MSG0		MSG1	MSG2	MSG3
IPC_CMD_START	Request	N/A	N/A	N/A
	Reply	N/A	N/A	N/A
IPC_CMD_MAP	Request	Lower 32 bits of size	Upper 32 bits of size	N/A
	Reply	Lower 32 bits of address	Upper 32 bits of address	N/A
IPC_CMD_OK	Request	N/A	N/A	N/A
	Reply	N/A	N/A	N/A
IPC_CMD_DOWN	Request	N/A	N/A	N/A
	Reply	N/A	N/A	N/A
IPC_CMD_HELLO	Request	Bit[0..15]: BAR[0..3] to use. Bit[16..31]: Door-bell bit to use for interrupt triggering after an entry is added to the Post-Queue.		N/A
	Reply	N/A	N/A	N/A

Table 6 IPC Request and Reply Definitions for Commands

IPC State Diagram

After the Base Layer driver gets loaded and module initialization is completed, the driver needs to initialize the IPC communication channel by issuing an IPC_CMD_START command to the other side of NTB. After an IPC_CMD_START reply is sent from the other side of NTB, IPC_CMD_MAP and IPC_CMD_HELLO commands will follow. If they complete this hand-shaking process without any error, they both are ready to start transferring data for the Function Layer drivers.

The Base Layer driver notifies all registered Function Layer drivers by calling their associated event call-back function provided by the Function Layer driver with NTB_EVENT_UP message. Once the registered Function Layer device driver receives the message, it sends the IPC_CMD_HELLO command, as mentioned in Table 6, to the other NTB endpoint. When the other NTB endpoint receives the IPC_CMD_HELLO request, it will check to see if it provides the same Function Layer service. If yes, it initializes the remote queue associated with the requesting endpoint and replies to the IPC request with IPC_STS_OK. Otherwise, it replies with IPC_STS_NS.

It makes no difference which endpoint has the NTB endpoint device driver loaded first because they synchronize with each other in STATE_INIT by an IPC_CMD_START command. The IPC states are defined in Table 7. After completing all IPC_CMD_MAP exchanges between the NTB endpoints, they synchronize their states in STATE_OK by the IPC_CMD_OK command.

Figure 7 illustrates the IPC state diagram of a NTB endpoint.

Notes

	STATE_DOWN	STATE_INIT	STATE_MAP	STATE_OK	STATE_ERROR
values	0x00000001	0x00000002	0x00000004	0x00001000	others

Table 7 IPC State Definition

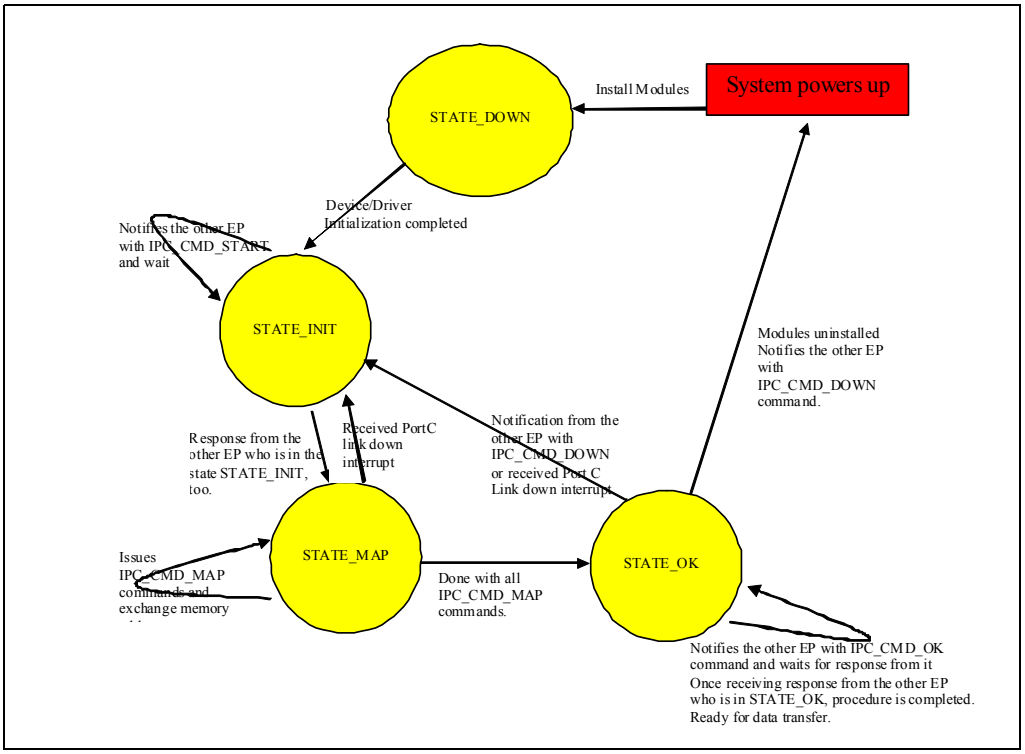

Figure 7 IPC State Diagram

EEPROM Initialization

The following registers need to be initialized through EEPROM for NTB mode in 32-bit (Table 8) and 64-bit BARs (Table 9).

Register Name	Register Offset	Register Value	Comment
PCIE_INTRLINE	0x303C	0x00000100	Use INTA
PCEE_INTRLINE	0x383C	0x00000100	Use INTA
PCIE_NTBCTL	0x3078	0x00000000	Enable opposite side
PCIE_BARSETUP0	0x307C	0x80000140	Memory Space Indicator = Memory Space Address Type = 32-bit addressing Prefetchable = False Size = 0x14 (1 Mbytes) Bar Enable = True

Table 8 32-bit BAR EEPROM Contents for NTB Initialization (Page 1 of 2)

Notes

PCEE_BARSETUP0	0x387C	0x80000140	Memory Space Indicator = Memory Space Address Type = 32-bit addressing Prefetchable = False Size = 0x14 (1 Mbytes) Bar Enable = True
PCIE_BARSETUP1	0x3084	0x80000140	Memory Space Indicator = Memory Space Address Type = 32-bit addressing Prefetchable = False Size = 0x14 (1 Mbytes) Bar Enable = True
PCEE_BARSETUP1	0x3884	0x80000140	Memory Space Indicator = Memory Space Address Type = 32-bit addressing Prefetchable = False Size = 0x14 (1 Mbytes) Bar Enable = True
PCIE_BARSETUP4	0x309C	0x800000c0	Memory Space Indicator = Memory Space Address Type = 32-bit addressing Prefetchable = False Size = 0xC (4 Kbytes) Bar Enable = True
PCEE_BARSETUP4	0x389C	0x800000c0	Memory Space Indicator = Memory Space Address Type = 32-bit addressing Prefetchable = False Size = 0xC (4 Kbytes) Bar Enable = True

Table 8 32-bit BAR EEPROM Contents for NTB Initialization (Page 2 of 2)

Register Name	Register Offset	Register Value	Comment
PCIE_INTRLINE	0x303C	0x00000100	Use INTA
PCEE_INTRLINE	0x383C	0x00000100	Use INTA
PCIE_NTBCTL	0x3078	0x00000000	Enable opposite side
PCIE_BARSETUP0	0x307C	0x8000014c	Memory Space Indicator = Memory Space Address Type = 64-bit addressing Prefetchable = True Size = 0x14 (1 Mbytes) Bar Enable = True
PCEE_BARSETUP0	0x387C	0x8000014c	Memory Space Indicator = Memory Space Address Type = 64-bit addressing Prefetchable = True Size = 0x14 (1 Mbytes) Bar Enable = True

Table 9 64-bit BAR EEPROM Contents for NTB Initialization (Page 1 of 2)

Notes

PCIE_BARSETUP2	0x308C	0x8000014c	Memory Space Indicator = Memory Space Address Type = 64-bit addressing Prefetchable = True Size = 0x14 (1 Mbytes) Bar Enable = True
PCEE_BARSETUP2	0x388C	0x8000014c	Memory Space Indicator = Memory Space Address Type = 64-bit addressing Prefetchable = True Size = 0x14 (1 Mbytes) Bar Enable = True
PCIE_BARSETUP4	0x309C	0x800000cc	Memory Space Indicator = Memory Space Address Type = 32-bit addressing Prefetchable = False Size = 0xC (4 Kbytes) Bar Enable = True
PCEE_BARSETUP4	0x389C	0x800000cc	Memory Space Indicator = Memory Space Address Type = 32-bit addressing Prefetchable = False Size = 0xC (4 Kbytes) Bar Enable = True

Table 9 64-bit BAR EEPROM Contents for NTB Initialization (Page 2 of 2)

IMPORTANT NOTICE AND DISCLAIMER

RENESAS ELECTRONICS CORPORATION AND ITS SUBSIDIARIES (“RENESAS”) PROVIDES TECHNICAL SPECIFICATIONS AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT OF THIRD-PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for developers who are designing with Renesas products. You are solely responsible for (1) selecting the appropriate products for your application, (2) designing, validating, and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. Renesas grants you permission to use these resources only to develop an application that uses Renesas products. Other reproduction or use of these resources is strictly prohibited. No license is granted to any other Renesas intellectual property or to any third-party intellectual property. Renesas disclaims responsibility for, and you will fully indemnify Renesas and its representatives against, any claims, damages, costs, losses, or liabilities arising from your use of these resources. Renesas' products are provided only subject to Renesas' Terms and Conditions of Sale or other applicable terms agreed to in writing. No use of any Renesas resources expands or otherwise alters any applicable warranties or warranty disclaimers for these products.

(Disclaimer Rev.1.01 Jan 2024)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Contact Information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit www.renesas.com/contact-us/.

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.