System Interconnect Software Programming Interface

Notes

By Steve Shih and Alex Chang

Overview

This document describes the programming interface of the system interconnect software components. Specifically, it describes the programming interfaces between the system interconnect software components of the Intel IA32 Root Processor (RP) and the Endpoint Processor (EP) running Linux. EP can be IOP80333 or x86 CPU with non-transparent bridging (NTB) enabled in IDT's 89HPES24NT3 PCIe® switch. Please refer to the PCI Express System Interconnect Software Architecture document (see below) for a description of the system interconnect system architecture and components.

References

89HPES24NT3 User Manual

PCI Express Base Specification Revision 1.0a

Linux source code (linux-2.6.x)

pci.txt under Linux source tree

Enabling Multi-peer Support with a Standard-Based PCI Express Multi-ported Switch. Kwok Kong, IDT White Paper.

PCI Express System Interconnect Software Architecture. Kwok Kong, IDT Application Note 531.

Development Environment

Intel IOP80333 Endpoint Processor

All software development of the system interconnect system is done on Fedora Core 6 i386 systems. The testing was done on an Intel Lindenhurst system with multiple Intel IOP80333 Customer Reference boards as the root processor and the endpoint processors, respectively. The root processor runs Fedora Core 6 i386 Linux distribution. The endpoint processors run RedBoot 2.1 and Linux kernel 2.6.14.3.

i386 Linux kernel 2.6.18.1 is built with:

- GCC 4.1.1
- GNU Binutils 2.17.50.0.3

IOP80333 RedBoot 2.1 is built with:

- GCC 3.3.1
- GNU Binutils 2.14.90

IOP80333 Linux kernel 2.6.14.3 is built with:

- GCC 3.4.5
- GNU Binutils 2.16.1

x86 CPU with NTB

All software development of the system interconnect system is done on Fedora Core 6 i386 systems.

The testing was done on an MSI K8N NEO4 motherboard with AMD Athlon64 3800 CPU as the root processor and the endpoint processors. Both root processor and endpoint processor run Fedora Core 6 i386 Linux distribution.

Notes

I386 Linux kernel 2.6.18.1 is built with:

- GCC 4.1.1

- GNU Binutils 2.17.50.0.3

Source Directory Structure

All system interconnect system software source files are in the mp directory with the following subdirectories:

– include: contains the public header files

– function: contains all the function service source and header files
There is a subdirectory for each of the function service, such as ether for the virtual Ethernet function service.

– message: contains the message frame service source and header files

– device: contains the device specific source and header files

There are 2 subdirectories, called ep and rp, under the device directory. All software modules installed in RP are included under the rp subdirectory. Currently, IDT only supports x86-based RP. Under the ep subdirectory, all software modules installed in the IOP80333 EP are included in the iop333 directory, and all software modules installed in the x86/NTB EP are included in the i386ntb directory

Local Processor

Address Conversion

The following functions are provided by the local processor for converting address types:

– mp_in_phys_to_pci: converts inbound physical address to PCI address
u64 mp_in_phys_to_pci(unsigned long address);
Parameters:
address: the inbound physical address
Returns: the inbound PCI address

– mp_in_pci_to_phys: converts inbound PCI address to physical address
unsigned long mp_in_pci_to_phys(u64 address);
Parameters:
address: the inbound PCI address
Returns: the inbound physical address

– mp_in_virt_to_pci: converts inbound virtual address to PCI address
u64 mp_in_virt_to_pci(void* address);
Parameters:
address: the inbound virtual address
Returns: the inbound PCI address

– mp_in_pci_to_virt: converts inbound PCI address to virtual address
void* mp_in_pci_to_virt(u64 address);
Parameters:
address: the inbound PCI address
Returns: the inbound virtual address

– mp_out_phys_to_pci: converts outbound physical address to PCI address
u64 mp_out_phys_to_pci(unsigned long address);

Notes

Parameters:

address: the outbound physical address

Returns: the outbound PCI address

– mp_out_pci_to_phys: converts outbound PCI address to physical address

unsigned long mp_out_pci_to_phys(u64 address);

Parameters:

address: the outbound PCI address

Returns: the outbound physical address

– mp_out_virt_to_pci: converts outbound virtual address to PCI address

u64 mp_out_virt_to_pci(void* address);

Parameters:

address: the outbound virtual address

Returns: the outbound PCI address

– mp_out_pci_to_virt: converts outbound PCI address to virtual address

void* mp_out_pci_to_virt(u64 address);

Parameters:

address: the outbound PCI address

Returns: the outbound virtual address

DMA Transfer

The local processor also provides the following DMA abstractions:

```
/*
 * DMA transfer direction
 */
typedef enum MP_DMA_DIR {
        MP_DMA_DIR_L2L,
        MP_DMA_DIR_L2P,
        MP_DMA_DIR_P2L,
        MP_DMA_DIR_P2P
} MP_DMA_DIR;
```

- MP_DMA_DIR_L2L: specifies local address space to local address space transfer
- MP_DMA_DIR_L2P: specifies local address space to PCI address space transfer
- MP_DMA_DIR_P2L: specifies PCI address space to local address space transfer
- MP_DMA_DIR_P2P: specifies PCI address space to PCI address space transfer

```
/*
 * DMA fragment
 */
typedef struct mp_dma_frag {
        u64 dst;
        u64 src;
        u32 len;
} mp_dma_frag;
```

- dst: specifies the destination address
- src: specifies the source address

Notes

- len: specifies the length of the data fragment

```
/*
 * DMA termination callback function
 * status:
 *          zero if DMA transfer completed without error
 *          non-zero if DMA transfer terminated with error
 */
typedef void (*mp_dma_cb)(int status, void* cb_data);
```

- mp_dma_cb: specifies DMA callback function prototype

– mp_dma_start: start a DMA transfer

```
int mp_dma_start(MP_DMA_DIR dir, u32 num_frags, mp_dma_frag* frags, mp_dma_cb cb,
        void* cb_data, u32 virt_base, u32 phys_base);
```

Parameters:

dir: specifies the DMA transfer type
num_frags: specifies the number of data fragments for the DMA transfer
frags: array of mp_dma_frag specifies the DMA fragments
cb: specifies the DMA callback function when done
cb_data: specifies the parameter to be passed to the DMA callback function
virt_base: virtual address base of DMA destination
phys_base: physical address base of DMA destination

Returns: zero for success and non-zero for errors

Note that a specific local processor may not support all the DMA transfer types defined above.

Peer Data Structure

Each peer in the system interconnect system is represented by the mp_peer data structure defined below:

```
#define MP_PEER_ID(b,d,f)          ((((b)&0xff)<<8) | (((d)&0x1f)<<3) | ((f)&0x7)
#define MP_PEER_SELF0       x80000000  /* self peer ID */
#define MP_PEER_RP          0       /* the RP peer ID */
#define MP_PEER_BCAST       ~0      /* broadcast peer ID */

/*
 * statistics data structure
 */
typedef struct mp_stats {
        u64 tx_frames;
        u64 tx_bytes;
```

Notes

```
        u64 tx_errors;
        u64 rx_frames;
        u64 rx_bytes;
        u64 rx_errors;
} mp_stats;
```

- tx_frames: number of data frames transmitted
- tx_bytes: number of bytes transmitted
- tx_errors: number of transmit errors
- rx_frames: number of data frames received
- rx_bytes: number of bytes received
- rx_errors: number of receive errors

```
/*
 * definition of peer data structure
 */
typedef struct mp_peer {
        struct list_head list;
        atomic_t ref;
        void* trans;
        u32 type;
        u32 id;
        u32 data_len;
        wait_queue_head_t statsq;
        int status;
        mp_stats stats;
        struct kobject kobj;
        struct work_struct work_kobj_add;
        struct work_struct work_kobj_del;
        struct work_struct work_peer;
        int index;
        int transportLayerUnloading;
} mp_peer;
```

- list: for linking the mp_peer
- ref: for reference counting the mp_peer
- trans: points to the transport service associated with this peer
- type: the transport service ID
- id: the peer ID
- data_len: length of the peer specific data embedded in this peer
- statsq: statistics request wait queue
- status: statistics request wait status
- stats: statistics
- kobj: kobject for the sysfs entries
- work_kobj: work queue for sysfs kobject adding
- work_kobj: work queue for sysfs kobject deleting
- work_peer: work queue for peer notification processing
- index: Peer index, 0 for the RP

Notes

- transportLayerUnloading: Set by the transport layer when it unloads. When set on the RP it prevents removal messages being sent to the other peers.

In addition to the fields explicitly defined above, each mp_peer embeds the peer specific private data defined and used by its corresponding transport service at the end of the mp_peer data structure.

The following functions are provided to facilitate the use of the mp_peer data structure:

– mp_peer_alloc: creates a new mp_peer

mp_peer mp_peer_alloc(void* trans, u32 type, u32 id, u32 data_len, u32 priv_len);*

Parameters:

trans: points to the transport service associated with this peer
type: the transport service ID
id: the peer ID
data_len: specifies the length of the peer specific data to be embedded
priv_len: specifies the length of the private data to be embedded

Returns: pointer to the newly created mp_peer

– mp_peer_free: releases a mp_peer

void mp_peer_free(mp_peer peer);*

Parameters:

peer: points to the mp_peer to be released

– mp_peer_inc: increments the mp_peer reference count

mp_peer mp_peer_inc(mp_peer* peer);*

Parameters:

peer: points to the mp_peer to increment the reference count

Returns: pointer to the mp_peer if reference count incremented or NULL if failed

– mp_peer_dec: decrements the mp_peer reference count and release mp_peer if reaches zero

void mp_peer_dec(mp_peer peer);*

Parameters:

Notes

peer: points to the mp_peer to decrement the reference count

– mp_peer_data: retrieves the peer specific data embedded in the mp_peer

void mp_peer_data(mp_peer* peer);*

Parameters:
peer: points to the mp_peer

Returns: pointer to the peer specific data

– mp_peer_priv: retrieves the private data embedded in the mp_peer

void mp_peer_priv(mp_peer* peer);*

Parameters:
peer: points to the mp_peer

Returns: pointer to the private data

Frame Data Structure

The unit of data exchange between the peer processors is represented by the mp_frame data structure. Each mp_frame is composed of one or more data fragments and the headers added by the system interconnect system architecture layers as it is passed down to be transferred out of the source system. When the mp_frame is received and passed up on the destination system, the headers are extracted and removed from the mp_frame by the corresponding system interconnect system architecture layers. For broadcast traffic, the mp_frame may be duplicated by the message frame service layer and passed down to one or more transport services.

mp_frag

Each data fragment of the mp_frame is represented by a mp_frag data structure defined below:

```
/*
 * definition of data fragment
 */
typedef struct mp_frag {
        u8* buf;
        u32 len;
} mp_frag;
```

- buf: points to the buffer holding the data fragment
- len: indicates the length of the data fragment

Notes

mp_frame

The mp_frame data structure is defined below:

```
/*
 * definition of data frame
 */
typedef struct mp_frame {
        struct list_head list;
        atomic_t ref;
        u32 frags;
        u32 func_len;
        void (*ds)(struct mp_frame* frame);
        int status;
        u32 flags;
        struct mp_frame* from;
        mp_peer* dst;
        mp_peer* src;
        void* func_priv;
        /* followed by
                array of mp_frag
                transport layer header
                message header
                function header
                private data */
} mp_frame;
```

- list: for queuing the mp_frame
- ref: for reference counting the mp_frame
- frags: indicates the number of data fragments in the mp_frame
- func_len: indicates the length of the function service header
- ds: destructor to be called when ref reaches zero
- status: the status of the mp_frame to be passed to the destructor
- flags: indicate special case handling, such as MP_FRAME_PRIORITY for high priority handling
- from: points to the original frame where this mp_frame is cloned from
- dst: set to the destination mp_peer by the message frame service
- src: set to the source mp_peer by the message frame service

```
/*
 * data frame destructor
 */
typedef void (*mp_frame_ds)(mp_frame* frame);
```

- mp_frame_ds: specifies the mp_frame destructor function prototype

Notes

In addition to the fields explicitly defined above, each mp_frame embeds an array of mp_frag, the message frame service header, the function service header, and the private data at the end of the mp_frame data structure.

The following functions are provided to facilitate the use of the mp_frame data structure:

– mp_frame_alloc: creates a new mp_frame

mp_frame mp_frame_alloc(u32 frags, u32 func_len, u32 priv_len, mp_frame_ds ds);*

Parameters:

frags: specifies the number of data fragments to be embedded
func_len: specifies the length of the function service header to be embedded
priv_len: specifies the length of the private data to be embedded
ds: the destructor to be called

Returns: pointer to the newly created mp_frame

– mp_frame_clone: clones an existing mp_frame

mp_frame mp_frame_clone(mp_frame* frame, u32 priv_len, mp_frame_ds ds);*

Parameters:

frame: points to the existing mp_frame to be cloned
priv_len: specifies the length of the private data to be embedded
ds: the destructor to be called

Returns: pointer to the newly cloned mp_frame

– mp_frame_free: releases a mp_frame

void mp_frame_free(mp_frame frame);*

Parameters:

frame: points to the mp_frame to be released

– mp_frame_inc: increments the mp_frame reference count

mp_frame mp_frame_inc(mp_frame* frame);*

Parameters:

frame: points to the mp_frame to increment the reference count

Notes

Returns: pointer to the mp_frame if reference count incremented or NULL if failed

– mp_frame_dec: decrements the mp_frame reference count and release mp_frame if reaches zero

void mp_frame_dec(mp_frame frame);*

Parameters:

frame: points to the mp_frame to decrement the reference count

– mp_frame_dst_set: get a reference of the peer specified and set the destination for the mp_frame to it

mp_peer mp_frame_dst_set(mp_frame* frame, mp_peer* peer);*

Parameters:

frame: points to the mp_frame

peer: points to the mp_peer

Returns: pointer to the mp_peer if success or NULL if failed

– mp_frame_src_set: get a reference of the peer specified and set the source for the mp_frame to it

mp_peer mp_frame_src_set(mp_frame* frame, mp_peer* peer);*

Parameters:

frame: points to the mp_frame

peer: points to the mp_peer

Returns: pointer to the mp_peer if success or NULL if failed

– mp_frame_frag: retrieves the next mp_frag embedded in the mp_frame or the first mp_frag if frag parameter is NULL

mp_frag mp_frame_frag(mp_frame* frame, mp_frag* frag);*

Parameters:

frame: points to the mp_frame

frag: points to a mp_frag embedded

Returns: pointer to the next mp_frag

– mp_frame_msg: retrieves the message frame service header embedded in the mp_frame

Notes

void mp_frame_msg(mp_frame* frame);*

Parameters:

frame: points to the mp_frame

Returns: pointer to the message frame service header

– mp_frame_func: retrieves the function service header embedded in the mp_frame

void mp_frame_func(mp_frame* frame);*

Parameters:

frame: points to the mp_frame

Returns: pointer to the function service header

– mp_frame_priv: retrieves the private data embedded in the mp_frame

void mp_frame_priv(mp_frame* frame);*

Parameters:

frame: points to the mp_frame

Returns: pointer to the private data

– mp_frame_msg_len: retrieves the length of the message frame service header

u32 mp_frame_msg_len(void);

Returns: the length of the message frame service header

– mp_frame_func_len: retrieves the length of the function service header embedded in the mp_frame

u32 mp_frame_func_len(mp_frame frame);*

Parameters:

frame: points to the mp_frame

Returns: the length of the function service header

Notes

– mp_frame_hdr_len: retrieves the length of the message frame and function header embedded in the mp_frame

u32 mp_frame_hdr_len(mp_frame frame);*

Parameters:
frame: points to the mp_frame

Returns: the length of the message frame and function headers

– mp_frame_data_len: retrieves the length of the data embedded in the mp_frame

u32 mp_frame_data_len(mp_frame frame);*

Parameters:
frame: points to the mp_frame

Returns: the length of the data

– mp_frame_len: retrieves the length of the data and headers embedded in the mp_frame

u32 mp_frame_len(mp_frame frame);*

Parameters:
frame: points to the mp_frame

Returns: the length of the data and headers

Transport Service

Each transport service is represented by the mp_trans data structure defined below:

```
/*
 * multi peer transport service
 */
typedef struct mp_trans {
        struct list_head list;
        atomic_t ref;
        u32 id;
        mp_stats stats;
        mp_peer* (*peer_add)(u32 peer, void* data);
        void (*peer_del)(mp_peer* peer);
        int (*frame_send)(mp_frame* frame);
```

Notes

*int (*frame_sync)(mp_frame* frame, u32 frags, mp_frag* buffers, mp_dma_cb cb, void* cb_data);*

} mp_trans;

- list: for linking the mp_trans
- ref: for reference counting the mp_trans
- id: identifies the transport service
- stats: statistics

- peer_add: notifies the transport service to add a new peer of its type

mp_peer (*peer_add)(u32 peer, void* data);*

Parameters:

p*eer: the peer ID*
data: points to the peer specific data associated with the peer

Returns: pointer to the newly added mp_peer

- peer_del: notifies the transport service to remove a peer of its type

*void (*peer_del)(mp_peer* peer);*

Parameters:

peer: points to the mp_peer

- frame_send: to send the mp_frame to a peer

*int (*frame_send)(mp_frame* frame);*

Parameters:

frame: points to the mp_frame

Returns: zero for success and non-zero for errors

- frame_sync: to synchronize the data fragments in the mp_frame

*int (*frame_sync)(mp_frame* frame, u32 frags, mp_frag* buffers, mp_dma_cb cb, void* cb_data);*

Parameters:

frame: points to the mp_frame for synchronizing the data from
frags: specifies the number of elements in the buffers array
buffers: points to an array of mp_frag for synchronizing the data to
cb: specifies the DMA callback function when done

Notes

cb_data: specifies the parameter to be passed to the DMA callback

Returns: zero for success and non-zero for errors

In addition to the fields explicitly defined above, the mp_trans may embed a transport service specific private data at the end of the mp_trans structure.

The following functions are provided to facilitate the use of the mp_trans data structure:

– mp_trans_alloc: creates a new mp_trans

mp_trans mp_trans_alloc(u32 id, u32 priv_len);*

Parameters:
id: the tra*nsport service ID*
*priv_len: spe*cifies the length of the private data to be embedded

Returns: pointer to the newly created mp_trans

– mp_trans_free: releases a mp_trans

void mp_trans_free(mp_trans trans);*

Parameters:
trans: points to the mp_trans to be released

– mp_trans_inc: increments the mp_trans reference count

mp_trans mp_trans_inc(mp_trans* trans);*

Parameters:
trans: points to the mp_trans to increment the reference count

Returns: pointer to the mp_trans if reference count incremented or NULL if failed

– mp_trans_dec: decrements the mp_trans reference count and release mp_trans if reaches zero

void mp_trans_dec(mp_trans trans);*

Parameters:
trans: points to the mp_trans to decrement the reference count

Notes

– mp_trans_priv: retrieves the private data embedded in the mp_trans

void mp_trans_priv(mp_trans* trans);*

Parameters:
trans: points to the mp_trans

Returns: pointer to the private data

Function Service

Each function service is represented by the mp_func data structure defined below:

```
/*
 * multi peer function service
 */
typedef struct mp_func {
        struct list_head list;
        atomic_t ref;
        u32 id;
        void (*peer_add)(mp_peer* peer);
        void (*peer_del)(mp_peer* peer);
        int (*frame_receive)(mp_frame* frame);
} mp_func;
```

- list: for linking the mp_func
- ref: for reference counting the mp_func
- id: identifies the function service
-
- peer_add: notifies the function service of a new peer

*void (*peer_add)(mp_peer* peer);*

Parameters:
peer: pointers to mp_peer

- peer_del: notifies the function service of a peer removal

*void (*peer_del)(mp_peer* peer);*

Parameters:
peer: points to the mp_peer

- frame_receive: to receive the mp_frame

Notes

*int (*frame_receive)(mp_frame* frame);*

Parameters:

frame: points to the mp_frame

Returns: zero for success and non-zero for errors

In addition to the fields explicitly defined above, the mp_func may embed a function service specific private data at the end of the mp_func structure.

The following functions are provided to facilitate the use of the mp_func data structure:

- mp_func_alloc: creates a new mp_func

  mp_func mp_func_alloc(u32 id, u32 priv_len);*

  Parameters:

  id: the function service ID

  priv_len: specifies the length of the private data to be embedded

  Returns: pointer to the newly created mp_func

- mp_func_free: releases a mp_func

  void mp_func_free(mp_func func);*

  Parameters:

  func: points to the mp_func to be released

- mp_func_inc: increments the mp_func reference count

  mp_func mp_func_inc(mp_func* func);*

  Parameters:

  func: points to the mp_func to increment the reference count

  Returns: pointer to the mp_func if reference count incremented or NULL if failed

- mp_func_dec: decrements the mp_func reference count and release mp_func if reaches zero

  void mp_func_dec(mp_func func);*

Notes

Parameters:

func: points to the mp_func to decrement the reference count

- mp_func_priv: retrieves the private data embedded in the mp_func

void mp_func_priv(mp_func* func);*

Parameters:

func: points to the mp_func

Returns: pointer to the private data

Message Frame Service

Message Frame Header

The message frame service prepends its header to each mp_frame sent and extracts its header from each mp_frame received. The message frame service header is defined as follows:

```
/*
 * multi peer message frame service header
 */
typedef struct mp_msg {
        u32 dst;
        u32 src;
        u32 len;
        u32 func;
} mp_msg;
```

- dst: the destination peer ID
- src: the source peer ID
- len: the length of the function service header and data
- func: the function service ID

The message frame header data structure above is in little-endian format.

Transport Service Management

Transport service management provides the following functions:

– mp_trans_register: to register transport service

int mp_trans_register(mp_trans trans);*

Parameters:

trans: points to the mp_trans

Notes

Returns: zero for success and non-zero for errors

– mp_trans_unregister: to unregister transport service

int mp_trans_unregister(mp_trans trans);*

Parameters:
trans: points to the mp_trans

Returns: zero for success and non-zero for errors

– mp_trans_next: retrieve the next transport service or the first one if trans parameter is NULL

mp_trans mp_trans_next(mp_trans* trans);*

Parameters:
trans: points to the mp_trans

Returns: pointer to the mp_trans

– mp_trans_get: retrieve the transport service by ID

mp_trans mp_trans_get(u32 id);*

Parameters:
id: the transport service ID

Returns: pointer to the mp_trans

Function Service Management

Function service management provides the following functions:

– mp_func_register: to register function service

int mp_func_register(mp_func func);*

Parameters:
func: points to the mp_func

Returns: zero for success and non-zero for errors

Notes

– mp_func_unregister: to unregister function service

int mp_func_unregister(mp_func func);*

Parameters:
func: points to the mp_func

Returns: zero for success and non-zero for errors

– mp_func_next: retrieve the next function service or the first one if func parameter is NULL

mp_func mp_func_next(mp_func* func);*

Parameters:
func: points to the mp_func

Returns: pointer to the mp_func

– mp_func_get: retrieve the function service by ID

mp_func mp_func_get(u32 id);*

Parameters:
id: the function service ID

Returns: pointer to the mp_func

Peer Management

Peer management provides the following functions:

– mp_peer_add: to add a new peer

int mp_peer_add(mp_peer peer);*

Parameters:
peer: points to the mp_peer

Returns: zero for success and non-zero for errors

– mp_peer_del: to remove an existing peer

Notes

int mp_peer_del(mp_peer peer);*

Parameters:
peer: points to the mp_peer

Returns: zero for success and non-zero for errors

– mp_peer_next: retrieve the next peer or the first one if peer parameter is NULL

mp_peer mp_peer_next(mp_peer* peer);*

Parameters:
peer: points to the mp_peer

Returns: pointer to the mp_peer

– mp_peer_get: retrieve a peer by ID

mp_peer mp_peer_get(u32 id);*

Parameters:
id: the peer ID

Returns: pointer to the mp_peer

– mp_self_add: adds self

int mp_self_add(mp_peer self);*

Parameters:
self: points to the mp_peer

Returns: zero for success and non-zero for failure

– mp_self_del: removes self

int mp_self_del(mp_peer self);*

Parameters:
self: points to the mp_peer

Notes

Returns: zero for success and non-zero for failure

– mp_self_get: retrieves the mp_peer data structure for itself

mp_peer mp_self_get(void);*

Returns: pointer to the mp_peer for itself

Frame Transfer

Frame transfer provides the following functions:

– mp_frame_send: to send a frame

int mp_frame_send(mp_frame frame, u32 dst, u32 func);*

Parameters:
Frame: points to the mp_frame to be sent
dst: specifies the destination peer ID
func: specifies the function ID

Returns: zero for success and non-zero for errors

– mp_frame_receive: to receive a frame

int mp_frame_receive(mp_frame frame);*

Parameters:
frame: points to the mp_frame received

Returns: zero for success and non-zero for errors

– mp_frame_sync: to synchronize data in a frame

int mp_frame_sync(mp_frame frame, u32 frags, mp_frag* buffers, mp_dma_cb cb, void* cb_data);*

Parameters:
frame: points to the mp_frame from which data is to be transferred
frags: specifies the number of elements in the buffer array
buffers: points to an array of mp_frag to which data is to be transferred
cb: specified the DMA callback function when done
cb_data: specifies the parameter to be passed to the DMA callback function

Notes

Returns: zero for success and non-zero for errors

Endpoint-Specific Transport Service

The transport service defines the direction flags and data fragment format below. They are common to the currently supported endpoints.

```
/*
* Transport direction lags
*/
        #define MP_DMA_DIR_MASK 0x00000003
        #define MP_DMA_DIR_L2L 0x00000000
        #define MP_DMA_DIR_L2P 0x00000001
        #define MP_DMA_DIR_P2L 0x00000002
```

- MP_DMA_DIR_MASK: mask for transfer direction flags
- MP_DMA_DIR_L2L: specifies local address space to local address space transfer
- MP_DMA_DIR_L2P: specifies local address space to PCI address space transfer
- MP_DMA_DIR_P2L: specifies PCI address space to local address space transfer

```
/*
* Transport data fragment
*/
typedef struct mp_frag {
        u8* buf;
        u32 len;
} mp_frag;
```

- buf: pointer to the data fragment
- len: specifies the length of the data fragment

IOP80333 Endpoint

Each Intel IOP80333 message block contains a IOP80333 header defined below:

```
/*
* IOP 80333 header
*/
typedef struct mp_iop_hdr {
        u32 next;
        u32 hdr_len;
        u32 len;
        u32 reserved;
} mp_iop_hdr;
```

- next: points to the address of the next message block
- hdr_len: specifies the total length of the headers in this message block
- len: specifies the length of the data in this message block
- reserved: not used.

Notes

The Intel IOP80333 transport service associates each IOP80333 peer with a private data structure defined below:

```
/*
* IOP 80333 peer
*/
typedef struct iop_peer {
        u64 base;
} iop_peer;
```

- base: specifies the base address of the peer in the PCI address space

All the IOP80333 transport service data structures are in little-endian format.

x86/NTB Endpoint

Each message block contains an x86/NTB header defined below:

```
/*
NTB header
*/
typedef struct mp_x86_hdr {
        u32 buffer_len;
        u32 hdr_len;
        u32 len;
        u32 reserved;
} mp_x86_ hdr;
```

- buffer_len: specifies the total length of the transport buffer in bytes
- hdr_len: specifies the total length of the headers in this message block
- len: specifies the length of the data in this message block
- reserved: not used.

The transport service associates each x86/NTB peer with a private data structure defined below:

```
/*
* x86/NTB peer
*/
typedef struct __mp_x86_peer {
        u32 inb_base;
        u32 reg_base;
        u32 peer_index
} mp_x86_peer;
```

- inb_base: specifies the base address of the peer in the PCI address space
- reg_base: specifies the base register address of the peer in the PCI address space
- peer_index: specifies the peer index assigned by root complex.

Notes

All x86/NTB transport service data structures are in little-endian format.

Example of Transferring a mp_frame

For example, if a function service wants to keep track of the number of success and fail transfers to the RP and all other EPs, it embeds a pointer to its function specific statistics data structure in the mp_frame and defines the mp_frame destructor as follows:

```
typedef struct func_stats {
u32 success;
u32 fail;
} func_stats;

func_stats my_stats[2]; /* index 0 for RP and 1 for EPs */

void func_frame_ds(mp_frame* frame)
{
        func_stats* stat = *(func_stats**)mp_frame_priv(frame);
        mp_frag* frag = mp_frame_frag(frame, NULL);

        /*
         * free data fragment buffers
         */
        while (frag) {
            kfree(frag->buf);
            frag = mp_frame_frag(frame, frag);
    }

        /*
         * update statistics
         */
        if (frame->status) {
            stat->fail++;
        } else {
            stat->success++;
    }
}
```

The procedure to construct a 2 data fragment frame to the RP is as follows:

```
typedef struct func_hdr {
      u32anything;
} func_hdr;

mp_frame* frame = mp_frame_alloc(2, /* 2 data fragments */
    sizeof(func_hdr), /* length of the function header */
```

Notes

```
        sizeof(func_stats*), /* length of the private data */
        func_frame_ds); /* destructor to be called */

    /* setup the 1st data fragment */
    frag = mp_frame_frag(frame, NULL);
    frag->buf = buffer1;
    frag->len = buffer1_len;

    /* setup the 2nd data fragment */
    frag = mp_frame_frag(frame, frag);
    frag->buf = buffer2;
    frag->len = buffer2_len;

    /* setup the function header */
    func_hdr* hdr = mp_frame_hdr(frame);
    hdr->anything = something;

    /* setup private data */
    func_stats** stats = mp_frame_priv(frame);
    *stats = &my_stat[0]; /* index 0 for RP */
```

Once the construction of the mp_frame is completed, it can be sent by calling mp_frame_send. The code fragment for the mp_frame_send function is shown below:

```
int mp_frame_send(mp_frame* frame, u32 dst, u32 func)
{
        int ret;
        mp_msg* msg;
        mp_trans* trans;

        /* build the message frame service header */
        msg = mp_frame_msg(frame);
        msg->dst = dst;
        msg->src = my_id;
        msg->len = mp_frame_func_len(frame) + mp_frame_data_len(frame);
        msg->func = func;

        /* set the source and destination peer */
        mp_frame_src_set(frame, mp_self_get());
        mp_frame_dst_set(frame, mp_peer_get(dst));

        /* pass the frame to the transport service */
        trans = frame->dst->trans;
        ret = trans->frame_send(frame);

        return ret;
}
```

Notes

Once the frame is sent by the transport service, the transport service will call mp_frame_free, which will call the frame->ds callback function when the frame->ref reaches zero.

The transport service on the RP will detect the arrival of the new frame and construct a new mp_frame similar to what the function service did above. It copies the whole message frame header and function service header as raw data directly into the space embedded in the mp_frame structure. After the construction of the mp_frame is completed, the transport service calls mp_frame_receive. The code fragment for the mp_frame_receive function is shown below:

```
int mp_frame_receive(mp_frame* frame)
{
        int ret;
        mp_msg* msg;
        mp_func* func;

        /* get the message frame service header */
        msg = mp_frame_msg(frame);

        /* set the source and destination peer */
        mp_frame_src_set(frame, mp_peer_get(msg->src));
        mp_frame_dst_set(frame, mp_peer_get(msg->dst));

        /* pass the frame to the function service */
        func = mp_func_get(msg->func);
        ret = func->frame_receive(frame);

        return ret;
}
```

The function service would use mp_frame_frag function to determine the size of the buffer required, allocate one or more buffers, construct an array of mp_frag to describe these destination buffers, and call mp_frame_sync to copy data associated with the mp_frame into the newly allocated destination buffers. After mp_frame_sync returns, it should call mp_frame_free to free the mp_frame and let the frame destructor function do the cleanup.

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.