RPMSG RTOS Layer User's Guide

Freescale Semiconductor, Inc.

Document Number: RPMSGRTOSLAYERUG
Rev. 0.1
Nov 2015

Contents

Chapter 1
RPMsg Component

The Remote Processor Messaging (RPMsg) is a virtio-based messaging bus that allows Inter Processor Communications (IPC) between independent software contexts running on homogeneous or heterogeneous cores present in an Asymmetric Multi Processing (AMP) system. The RPMsg component source code has been recently published as a part of the Open Asymmetric Multi Processing (OpenAMP) Framework. The RPMsg API is compliant with the RPMsg bus infrastructure present in upstream Linux 3.4.x kernel onward.

This document describes the extension of the RPMsg API designed and implemented by Freescale. It discusses the motivation for these changes in the RPMsg as well as the advantages of the extension. This document also serves as the API reference, covering all newly added API functions that can be use in an RPMsg-based application.

The RPMsg extension is based on the OpenAMP repository code. See `https://github.com/OpenAMP/open-amp.git` / SHA1 ID 44b5f3c0a6458f3cf80. The documentation for the legacy RPMsg core code can be found in the *docs* folder of this repository.

1.1 Motivation to create RTOS aware API

The original RPMsg API is based on processing the transmitted data (messages) in the interrupt context via a registered receive callback, which is called when a message is received. The inconvenience of this approach is that either all the processing of received data must be done in the interrupt context, or that the message must be copied in a temporary application buffer for later processing. Both usages of the available API are not compatible with the concept of a Real-Time Operating System (RTOS), since the interrupt always preempts the running task, whatever its current priority, and this interruption can occur at a random date and can take a random amount of time to execute. This can introduce additional jitter in the real-time system timing. Additionally, the practice in application development using an RTOS is to have multiple independent sequential contexts. It is more natural and convenient to have a blocking sequential API, which was not available in the original RPMsg API. A good example of a blocking API is a socket interface or any POSIX-like interface. Therefore, the natural trend is to provide this a kind of interface to the application programmer. To summarize, the advantages of the RTOS-aware extension of RPMsg API are the following:

- No data processing in the interrupt context
- Blocking receive API
- Zero-copy send and receive API
- Receive with timeout provided by RTOS
- Compatibility with Linux OS upstream kept

1.2 Implementation

The Freescale contribution to the RPMsg consists of two additional layers that are created above the origin base RPMsg layer.

- The RPMsg Extension layer allows users to allocate and release virtio tx buffers, as well as implements the zero-copy send functionality. The RPMsg Extension layer API is intended to be used in Bare Metal applications.
- The RPMsg RTOS layer addresses RTOS-based application needs discussed above (handling received data outside the interrupt context, blocking receive API implementation, zero-copy mechanisms). See RPMsg RTOS layer API. This RTOS aware RPMsg API layer is split into multiple C modules. The module *rpmsg_rtos.c/.h* contains a generic implementation, which does not depend on the used RTOS nor on the used platform. In */porting/<device>/platform.c/.h* and *platform_info.c*, there are platform (SoC) dependent functions. In */porting/env/<rtos name>/rpmsg_porting.c/.h*, the RTOS abstraction is implemented using functions from the *platform.h* to make connection with the hardware. However, the *rpmsg_porting.c/.h* module itself is hardware-independent.

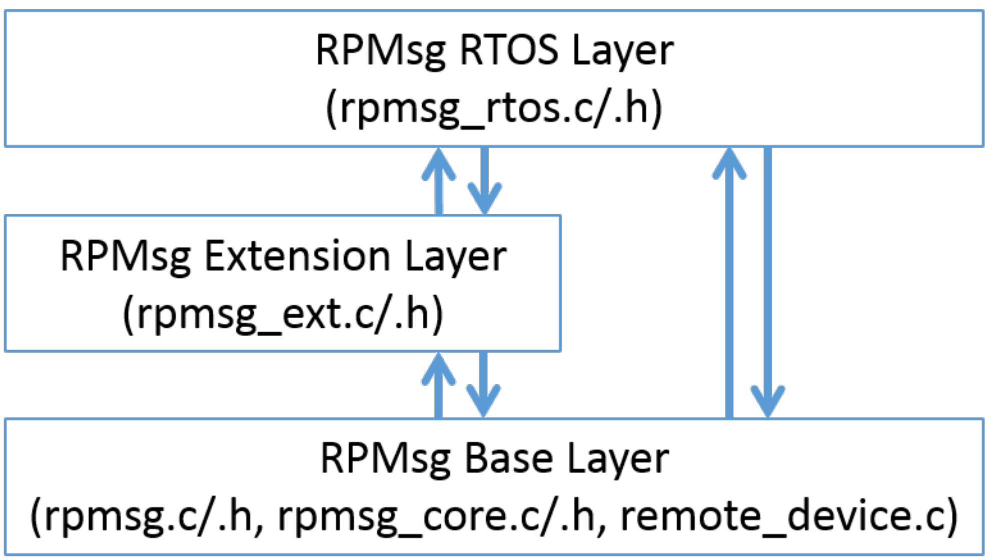

Figure 1.2.1: RPMsg layers

1.3 Usage

To access the RPMsg RTOS layer API, it is necessary to include the *rpmsg_rtos.h* file in the application C module (i.e., main.c). After the RTOS startup, the user should call the rpmsg_rtos_init() function to initialize the RPMsg and to synchronize with the opposite side (other core). After this, application endpoints can be created in any RTOS threads by calling rpmsg_rtos_create_ept(). Consequently, the rpmsg_rtos_send() function is used to send data from an endpoint to a remote endpoint, whose address is specified in the function call. The rpmsg_rtos_recv() is then used to receive data on an endpoint or to wait for data to be received with a certain timeout (or the timeout can be set to wait forever).

If the application is low on memory or needs to be more memory efficient and faster, the no-copy mechanism can be used. The RPMsg RTOS layer implements no-copy mechanisms for both sending and receiving operations. These methods require specifics that have to be considered when used in an application.

no-copy-send mechanism: This mechanism allows sending messages without the cost for copying data from the application buffer to the RPMsg/virtio buffer in the shared memory. The sequence of no-copy sending steps to be performed is as follows:

- Call the rpmsg_rtos_alloc_tx_buffer() function to get the virtio buffer and provide the buffer pointer to the application.
- Fill the data to be sent into the pre-allocated virtio buffer. Ensure that the filled data does not exceed the buffer size (provided as the rpmsg_rtos_alloc_tx_buffer() *size* output parameter).
- Call the rpmsg_rtos_send_nocopy() function to send the message to the destination endpoint. Consider the cache functionality and the virtio buffer alignment. See the rpmsg_rtos_send_nocopy() function description below.

no-copy-receive mechanism: This mechanism allows reading messages without the cost for copying data from the virtio buffer in the shared memory to the application buffer. The sequence of no-copy receiving steps to be performed is as follows:

- Call the rpmsg_rtos_recv_nocopy() function to get the virtio buffer pointer to the received data.
- Read received data directly from the shared memory.
- Call the rpmsg_rtos_recv_nocopy_free() function to release the virtio buffer and to make it available for the next data transfer.

Figure 1.3.1: RPMsg send/receive mechanisms

When deinitializating the RPMsg communication, the master side calls the rpmsg_rtos_deinit() function that deinitializes all on the master side, and also triggers the Name Service (NS) destroy callback on the remote side, which destroys the default channel and the default endpoint. From that time onwards, any call of send or receive API on the remote side returns an error. It is up to the user application to gracefully stop the RPMsg, i.e., to destroy all application-created endpoints (rpmsg_rtos_destroy_ept()) first, then destroy the RPMsg component (rpmsg_rtos_deinit()).

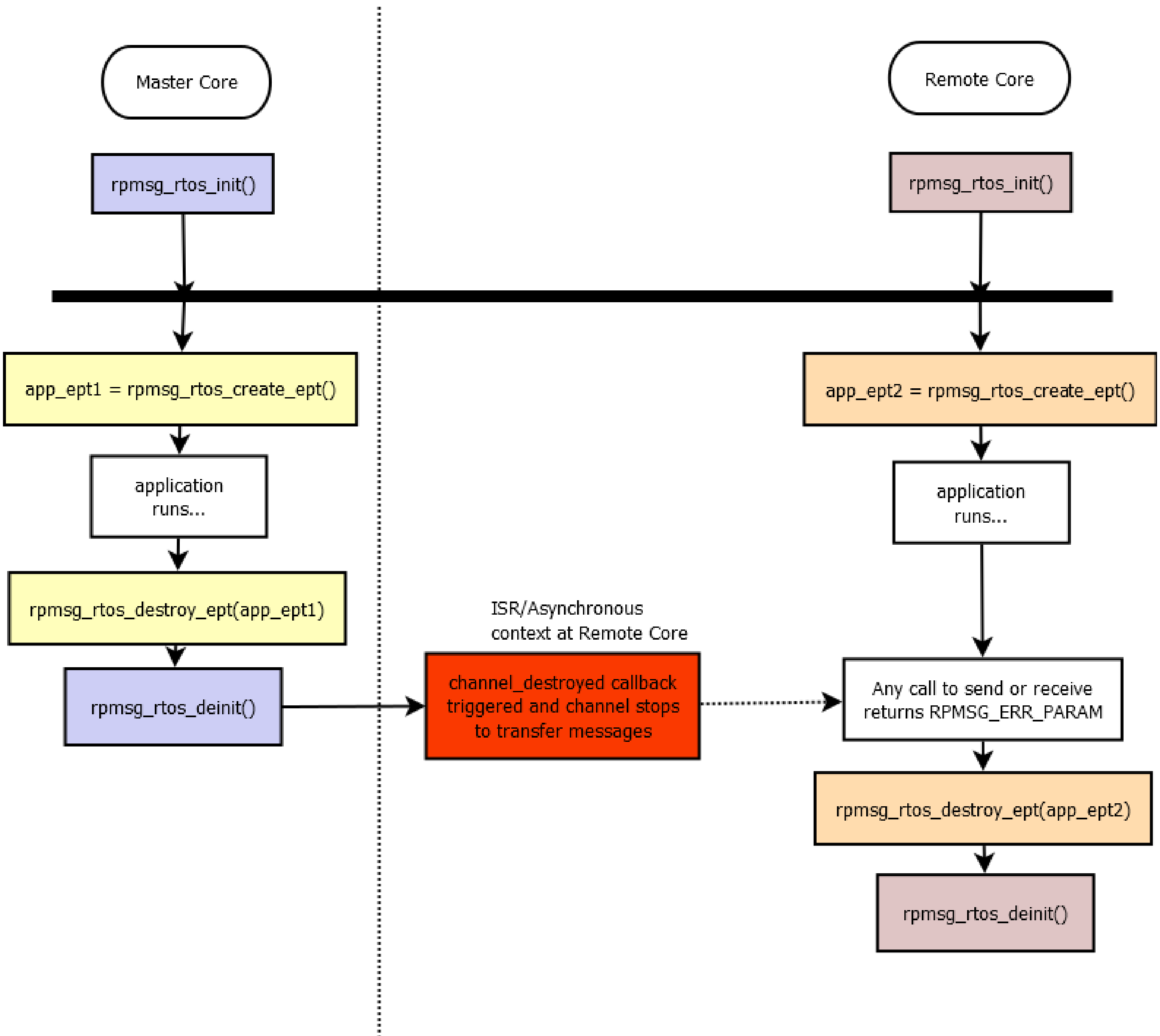

Figure 1.3.2: RPMsg init and deinit process in RTOS environment

1.4 RPMsg porting sub-layers

The RPMsg porting layers have been also modified and consolidated in order to

- Strictly separate platform-related (multicore device) and environment-related (Bare Metal, RTOS) layers.
- Update the environment layer API by functions requested by the RTOS layer. The following *env* functions have been introduced:
  - *int env_create_queue(void∗∗ queue, int length, int element_size)*
  - *void env_delete_queue(void∗ queue)*
  - *int env_put_queue(void∗ queue, void∗ msg, int timeout_ms)*
  - *int env_get_queue(void∗ queue, void∗ msg, int timeout_ms)*

Currently, the environment layer is implemented for Bare Metal and FreeRTOS. To support other RTOSes, it is necessary to create (clone) the *rpmsg_porting.c/.h* sub-layer using the desired RTOS API, put this code into the */porting/env/<rtos name>* folder, and to include this path into the list of the project include paths.

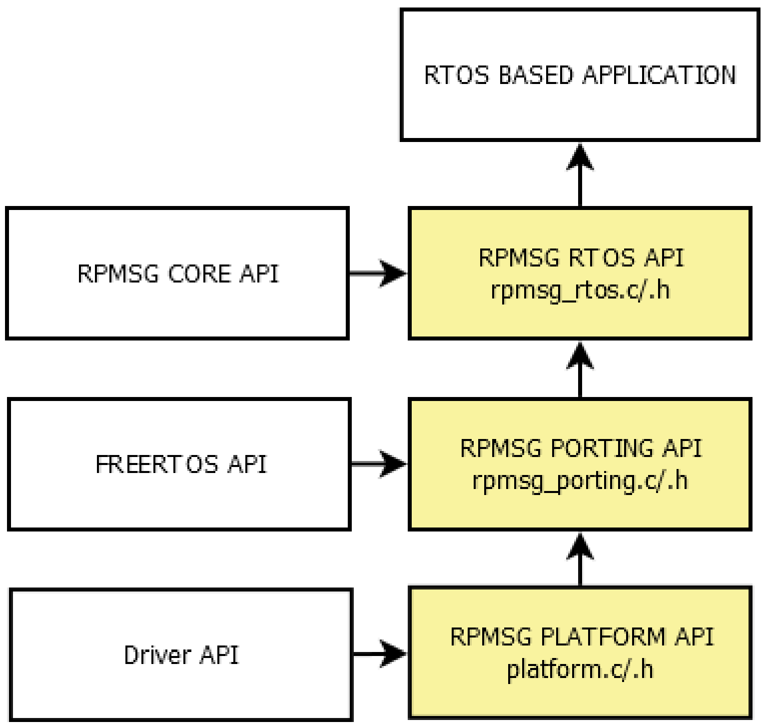

Figure 1.4.1: Rewrite rpmsg_porting.c/.h sub-layer

Chapter 2
RPMsg Extension Layer

2.1 Overview

This part describes the RPMsg extension layer that allows:

- Allocation/release of the virtio tx buffer.
- Zero-copy send functionality.

Functions

- void rpmsg_hold_rx_buffer (struct rpmsg_channel ∗rpdev, void ∗rxbuf)
- void rpmsg_release_rx_buffer (struct rpmsg_channel ∗rpdev, void ∗rxbuf)
- void ∗ rpmsg_alloc_tx_buffer (struct rpmsg_channel ∗rpdev, unsigned long ∗size, int wait)
- int rpmsg_send_offchannel_nocopy (struct rpmsg_channel ∗rpdev, unsigned long src, unsigned long dst, void ∗txbuf, int len)
- static int rpmsg_sendto_nocopy (struct rpmsg_channel ∗rpdev, void ∗txbuf, int len, unsigned long dst)
- static int rpmsg_send_nocopy (struct rpmsg_channel ∗rpdev, void ∗txbuf, int len)

2.2 Function Documentation

2.2.1 void rpmsg_hold_rx_buffer (struct rpmsg_channel ∗ *rpdev,* void ∗ *rxbuf*)

Holds the rx buffer for usage outside the receive callback.

Calling this function prevents the RPMsg receive buffer from being released back to the pool of shmem buffers. This API can only be called at rx callback context (rpmsg_rx_cb_t). With this API, the application doesn't need to copy the message in rx callback. Instead, the rx buffer base address is saved in application context and further processed in application process. After the message is processed, the application can release the rx buffer for future reuse in vring by calling the rpmsg_release_rx_buffer() function.

Parameters

in	*rpdev*	The rpmsg channel
in	*rxbuf*	RX buffer with message payload

See Also

rpmsg_release_rx_buffer

2.2.2 void rpmsg_release_rx_buffer (struct rpmsg_channel ∗ *rpdev,* void ∗ *rxbuf*)

Releases the rx buffer for future reuse in vring.

This API can be called at process context when the message in rx buffer is processed.

Parameters

rpdev	- the rpmsg channel
rxbuf	- rx buffer with message payload

See Also

rpmsg_hold_rx_buffer

2.2.3 void∗ rpmsg_alloc_tx_buffer (struct rpmsg_channel ∗ *rpdev,* unsigned long ∗ *size,* int *wait*)

Allocates the tx buffer for message payload.

This API can only be called at process context to get the tx buffer in vring. By this way, the application can directly put its message into the vring tx buffer without copy from an application buffer. It is the application responsibility to correctly fill the allocated tx buffer by data and passing correct parameters to the rpmsg_send_nocopy() or rpmsg_sendto_nocopy() function to perform data no-copy-send mechanism.

Parameters

in	*rpdev*	Pointer to rpmsg channel
in	*size*	Pointer to store tx buffer size
in	*wait*	Boolean, wait or not for buffer to become available

Returns

The tx buffer address on success and NULL on failure

See Also

rpmsg_send_offchannel_nocopy
rpmsg_sendto_nocopy
rpmsg_send_nocopy

2.2.4 int rpmsg_send_offchannel_nocopy (struct rpmsg_channel ∗ *rpdev,* unsigned long *src,* unsigned long *dst,* void ∗ *txbuf,* int *len*)

Sends a message in tx buffer allocated by rpmsg_alloc_tx_buffer() using explicit src/dst addresses.

This function sends txbuf of length len to the remote dst address, and uses src as the source address. The message will be sent to the remote processor which the rpdev channel belongs to. The application has to take the responsibility for:

1. tx buffer allocation (rpmsg_alloc_tx_buffer())
2. filling the data to be sent into the pre-allocated tx buffer
3. not exceeding the buffer size when filling the data
4. data cache coherency

After the rpmsg_send_offchannel_nocopy() function is issued the tx buffer is no more owned by the sending task and must not be touched anymore unless the rpmsg_send_offchannel_nocopy() function fails and returns an error. In that case the application should try to re-issue the rpmsg_send_offchannel_nocopy() again and if it is still not possible to send the message and the application wants to give it up from whatever reasons the rpmsg_release_rx_buffer function could be called, passing the pointer to the tx buffer to be released as a parameter.

Parameters

`in`	*rpdev*	The rpmsg channel
`in`	*src*	Source address
`in`	*dst*	Destination address
`in`	*txbuf*	TX buffer with message filled
`in`	*len*	Length of payload

Returns

0 on success and an appropriate error value on failure

See Also

rpmsg_alloc_tx_buffer
rpmsg_sendto_nocopy
rpmsg_send_nocopy

2.2.5 static int rpmsg_sendto_nocopy (struct rpmsg_channel ∗ *rpdev,* void ∗ *txbuf,* int *len,* unsigned long *dst*) `[static]`

Sends a message in tx buffer allocated by rpmsg_alloc_tx_buffer() across to the remote processor, specify dst.

This function sends txbuf of length len to the remote dst address. The message will be sent to the remote processor which the rpdev channel belongs to, using rpdev's source address. The application has to take the responsibility for:

1. tx buffer allocation (rpmsg_alloc_tx_buffer())
2. filling the data to be sent into the pre-allocated tx buffer
3. not exceeding the buffer size when filling the data
4. data cache coherency

After the rpmsg_sendto_nocopy() function is issued the tx buffer is no more owned by the sending task and must not be touched anymore unless the rpmsg_sendto_nocopy() function fails and returns an error. In that case the application should try to re-issue the rpmsg_sendto_nocopy() again and if it is still not possible to send the message and the application wants to give it up from whatever reasons the rpmsg_release_rx-_buffer function could be called, passing the pointer to the tx buffer to be released as a parameter.

Parameters

in	*rpdev*	The rpmsg channel
in	*txbuf*	TX buffer with message filled
in	*len*	Length of payload
in	*dst*	Destination address

Returns

0 on success and an appropriate error value on failure

See Also

rpmsg_alloc_tx_buffer
rpmsg_send_offchannel_nocopy
rpmsg_send_nocopy

2.2.6 static int rpmsg_send_nocopy (struct rpmsg_channel * *rpdev*, void * *txbuf*, int *len*) `[static]`

Sends a message in tx buffer allocated by rpmsg_alloc_tx_buffer() across to the remote processor.

This function sends txbuf of length len on the rpdev channel. The message will be sent to the remote processor which the rpdev channel belongs to, using rpdev's source and destination addresses. The application has to take the responsibility for:

1. tx buffer allocation (rpmsg_alloc_tx_buffer())
2. filling the data to be sent into the pre-allocated tx buffer
3. not exceeding the buffer size when filling the data
4. data cache coherency

After the rpmsg_send_nocopy() function is issued the tx buffer is no more owned by the sending task and must not be touched anymore unless the rpmsg_send_nocopy() function fails and returns an error. In that case the application should try to re-issue the rpmsg_send_nocopy() again and if it is still not possible to send the message and the application wants to give it up from whatever reasons the rpmsg_release_rx_-buffer function could be called, passing the pointer to the tx buffer to be released as a parameter.

Parameters

`in`	*rpdev*	The rpmsg channel
`in`	*txbuf*	TX buffer with message filled
`in`	*len*	Length of payload

Returns

0 on success and an appropriate error value on failure

See Also

rpmsg_alloc_tx_buffer
rpmsg_send_offchannel_nocopy
rpmsg_sendto_nocopy

Chapter 3
RPMsg RTOS Layer

3.1 Overview

This part describes the RPMsg RTOS adaptation layer that allows:

- Handling of received messages outside the interrupt context.
- Implementation of blocking API for the RPMsg receive side.
- Provides zero-copy receive functionality.
- Provides zero-copy send functionality.

Functions

- int rpmsg_rtos_init (int dev_id, struct remote_device **rdev, int role, struct rpmsg_channel **def_chnl)
- void rpmsg_rtos_deinit (struct remote_device *rdev)
- struct rpmsg_endpoint * rpmsg_rtos_create_ept (struct rpmsg_channel *rp_chnl, unsigned long addr)
- void rpmsg_rtos_destroy_ept (struct rpmsg_endpoint *rp_ept)
- int rpmsg_rtos_recv (struct rpmsg_endpoint *ept, void *data, int *len, int maxlen, unsigned long *src, int timeout_ms)
- int rpmsg_rtos_recv_nocopy (struct rpmsg_endpoint *ept, void **data, int *len, unsigned long *src, int timeout_ms)
- int rpmsg_rtos_recv_nocopy_free (struct rpmsg_endpoint *ept, void *data)
- void * rpmsg_rtos_alloc_tx_buffer (struct rpmsg_endpoint *ept, unsigned long *size)
- int rpmsg_rtos_send (struct rpmsg_endpoint *ept, void *data, int len, unsigned long dst)
- int rpmsg_rtos_send_nocopy (struct rpmsg_endpoint *ept, void *txbuf, int len, unsigned long dst)

3.2 Function Documentation

3.2.1 int rpmsg_rtos_init (int *dev_id*, struct remote_device ** *rdev*, int *role*, struct rpmsg_channel ** *def_chnl*)

This function allocates and initializes the rpmsg driver resources for given device ID (cpu id).

The successful return from this function leaves fully enabled IPC link. RTOS aware version.

Parameters

in	*dev_id*	Remote device for which driver is to be initialized
out	*rdev*	Pointer to newly created remote device

in	*role*	Role of the other device, Master or Remote
out	*def_chnl*	Pointer to rpmsg channel

Returns

Status of function execution

See Also

rpmsg_rtos_deinit

3.2.2 void rpmsg_rtos_deinit (struct remote_device * *rdev*)

This function frees rpmsg driver resources for given remote device.

RTOS aware version.

Parameters

in	*rdev*	Pointer to device to de-init

See Also

rpmsg_rtos_init

3.2.3 struct rpmsg_endpoint* rpmsg_rtos_create_ept (struct rpmsg_channel * *rp_chnl,* unsigned long *addr*)

This function creates rpmsg endpoint for the rpmsg channel.

RTOS aware version.

Parameters

in	*rp_chnl*	Pointer to rpmsg channel
in	*addr*	Endpoint src address

Returns

Pointer to endpoint control block

See Also

rpmsg_rtos_destroy_ept

3.2.4 void rpmsg_rtos_destroy_ept (struct rpmsg_endpoint * *rp_ept*)

This function deletes rpmsg endpoint and performs cleanup.

RTOS aware version.

Parameters

in	*rp_ept*	Pointer to endpoint to destroy

See Also

rpmsg_rtos_create_ept

3.2.5 int rpmsg_rtos_recv (struct rpmsg_endpoint * *ept,* void * *data,* int * *len,* int *maxlen,* unsigned long * *src,* int *timeout_ms*)

RTOS receive function - blocking version of the received function that can be called from an RTOS task.

The data is copied from the receive buffer into the user supplied buffer.

This is the "receive with copy" version of the RPMsg receive function. This version is simple to use but it requires copying data from shared memory into the user space buffer. The user has no obligation or burden to manage the shared memory buffers.

Parameters

in	*ept*	Pointer to the RPMsg endpoint on which data is received
in	*data*	Pointer to the user buffer the received data are copied to
out	*len*	Pointer to an int variable that will contain the number of bytes actually copied into the buffer
in	*maxlen*	Maximum number of bytes to copy (received buffer size)
out	*src*	Pointer to address of the endpoint from which data is received
in	*timeout_ms*	Timeout, in milliseconds, to wait for a message. A value of 0 means don't wait (non-blocking call). A value of 0xffffffff means wait forever (blocking call).

Returns

Status of function execution

See Also

rpmsg_rtos_recv_nocopy

3.2.6 int rpmsg_rtos_recv_nocopy (struct rpmsg_endpoint ∗ *ept,* void ∗∗ *data,* int ∗ *len,* unsigned long ∗ *src,* int *timeout_ms*)

RTOS receive function - blocking version of the received function that can be called from an RTOS task.

The data is NOT copied into the user-app. buffer.

This is the "zero-copy receive" version of the RPMsg receive function. No data is copied. Only the pointer to the data is returned. This version is fast, but it requires the user to manage buffer allocation. Specifically, the user must decide when a buffer is no longer in use and make the appropriate API call to free it, see rpmsg_rtos_recv_nocopy_free().

Parameters

`in`	*ept*	Pointer to the RPMsg endpoint on which data is received
`out`	*data*	Pointer to the RPMsg buffer of the shared memory where the received data is stored
`out`	*len*	Pointer to an int variable that that will contain the number of valid bytes in the RPMsg buffer
`out`	*src*	Pointer to address of the endpoint from which data is received
`in`	*timeout_ms*	Timeout, in milliseconds, to wait for a message. A value of 0 means don't wait (non-blocking call). A value of 0xffffffff means wait forever (blocking call).

Returns

Status of function execution

See Also

rpmsg_rtos_recv_nocopy_free
rpmsg_rtos_recv

3.2.7 int rpmsg_rtos_recv_nocopy_free (struct rpmsg_endpoint ∗ *ept,* void ∗ *data*)

This function frees a buffer previously returned by rpmsg_rtos_recv_nocopy().

Once the zero-copy mechanism of receiving data is used, this function has to be called to free a buffer and to make it available for the next data transfer.

Parameters

`in`	*ept*	Pointer to the RPMsg endpoint that has consumed received data
`in`	*data*	Pointer to the RPMsg buffer of the shared memory that has to be freed

Returns

Status of function execution

See Also

rpmsg_rtos_recv_nocopy

3.2.8 void∗ rpmsg_rtos_alloc_tx_buffer (struct rpmsg_endpoint ∗ *ept,* unsigned long ∗ *size*)

Allocates the tx buffer for message payload.

This API can only be called at process context to get the tx buffer in vring. By this way, the application can directly put its message into the vring tx buffer without copy from an application buffer. It is the application responsibility to correctly fill the allocated tx buffer by data and passing correct parameters to the rpmsg_rtos_send_nocopy() function to perform data no-copy-send mechanism.

Parameters

`in`	*ept*	Pointer to the RPMsg endpoint that requests tx buffer allocation
`out`	*size*	Pointer to store tx buffer size

Returns

The tx buffer address on success and NULL on failure

See Also

rpmsg_rtos_send_nocopy

3.2.9 int rpmsg_rtos_send (struct rpmsg_endpoint ∗ *ept,* void ∗ *data,* int *len,* unsigned long *dst*)

Sends a message across to the remote processor.

This function sends data of length len to the remote dst address. In case there are no TX buffers available, the function will block until one becomes available, or a timeout of 15 seconds elapses. When the latter happens, -ERESTARTSYS is returned.

Parameters

in	*ept*	Pointer to the RPMsg endpoint
in	*data*	Pointer to the application buffer containing data to be sent
in	*len*	Size of the data, in bytes, to transmit
in	*dst*	Destination address of the message

Returns

0 on success and an appropriate error value on failure

See Also

rpmsg_rtos_send_nocopy

3.2.10 int rpmsg_rtos_send_nocopy (struct rpmsg_endpoint ∗ *ept,* void ∗ *txbuf,* int *len,* unsigned long *dst*)

Sends a message in tx buffer allocated by rpmsg_rtos_alloc_tx_buffer() to the remote processor.

This function sends txbuf of length len to the remote dst address. The application has to take the responsibility for:

1. tx buffer allocation (rpmsg_rtos_alloc_tx_buffer())
2. filling the data to be sent into the pre-allocated tx buffer
3. not exceeding the buffer size when filling the data
4. data cache coherency

After the rpmsg_rtos_send_nocopy() function is issued the tx buffer is no more owned by the sending task and must not be touched anymore unless the rpmsg_rtos_send_nocopy() function fails and returns an error. In that case the application should try to re-issue the rpmsg_rtos_send_nocopy() again and if it is still not possible to send the message and the application wants to give it up from whatever reasons the rpmsg_rtos_recv_nocopy_free function could be called, passing the pointer to the tx buffer to be released as a parameter.

Parameters

in	*ept*	Pointer to the RPMsg endpoint
in	*txbuf*	Tx buffer with message filled
in	*len*	Size of the data, in bytes, to transmit
in	*dst*	Destination address of the message

Returns

0 on success and an appropriate error value on failure

See Also

rpmsg_rtos_alloc_tx_buffer
rpmsg_rtos_send

Chapter 4
Revision History

This table summarizes revisions to this document.

Revision number	Date	Substantive changes
0	09/2015	Initial release
0.1	11/2015	Update for Extension layer

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: RPMSGRTOSLAYERUG
Rev. 0.1
11/2015