OpenAMP Framework User Reference

Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: www.mentor.com
SupportNet: supportnet.mentor.com/

Send Feedback on Documentation: supportnet.mentor.com/doc_feedback_form

Table of Contents

List of Figures

List of Tables

Chapter 1
OpenAMP Framework Overview

Open Asymmetric Multi Processing (OpenAMP) Framework provides software components that enable development of software applications for Asymmetric Multiprocessing (AMP) systems.

Abbreviations, Terminology, and Definitions

The following abbreviations and terminology appear throughout the document.

Table 1-1. Abbreviations and Terminology

Abbreviations and Terminology	Definition
OpenAMP Framework	Open Asymmetric Multi Processing Framework
AMP	Asymmetric Multi Processing
LCM	Life Cycle Management
IPC	Inter Processor Communication
RTOS	Real Time Operating System
BM or BME	Bare Metal or Bare Metal Environment
HIL	Hardware Interface Layer
IPI	Inter-Processor Interrupt
Master	The CPU/software context that comes up first and manages other CPUs/software contexts present in the AMP system.
Remote	The CPU/software context that is brought up by the master CPUs/software context present in the AMP system.
Master processor	A Master CPU in a multicore SoC.
Remote processor	A Remote CPU in a multicore SoC.
Master software context	Any software context that can run on a master processor. This software context could be Linux or other OS, RTOS, or bare metal environment based.

Table 1-1. Abbreviations and Terminology (cont.)

Remote software context	Any software context that can run on a remote processor. This software context could be Linux or other OS, RTOS, or bare metal environment based.
Environment or software environment	Refers to the underlying software environment which could be OS, RTOS, or bare metal based.

Overview

An AMP system is characterized by multiple homogeneous and/or heterogeneous processing cores (for example, the Texas Instruments TI OMAP (System on Chips) SoCs have dual ARM Cortex A15, dual ARM Cortex M4, and C64 DSP cores). These cores typically run independent instances of homogeneous and/or heterogeneous software environments, such as Linux[1], RTOS, and Bare Metal that work together to achieve the design goals of the end application. While Symmetric Multiprocessing (SMP) operating systems allow load balancing of application workload across homogeneous processors present in such AMP SoCs, asymmetric multiprocessing design paradigms are required to leverage parallelism from the heterogeneous cores present in the system.

Increasingly, today's multicore applications require heterogeneous processing power. Heterogeneous multicore SoCs often have one or more general purpose CPUs (for example, dual ARM Cortex A9 cores on Xilinx Zynq) with DSPs and/or smaller CPUs and/or soft IP (on SoCs such as Xilinx Zynq APSOC). These specialized CPUs, as compared to the general purpose CPUs, are typicallyt dedicated for demand-driven offload of specialized application functionality to achieve maximum system performance. Systems developed using these types of SoCs, characterized by heterogeneity in both hardware and software, are generally termed as AMP systems.

In AMP systems, it is typical for software running on a master to bring up software/firmware contexts on a remote on a demand-driven basis and communicate with them using IPC mechanisms to offload work during run time. The participating master and remote processors may be homogeneous or heterogeneous in nature.

A *master* is defined as the CPU/software that is booted first and is responsible for managing other CPUs and their software contexts present in an AMP system. A *remote* is defined as the CPU/software context managed by the master software context present.

Components and Capabilities

The OpenAMP Framework implementation provides the necessary API infrastructure required to develop AMP systems.

1. Linux® is a registered trademark of Linus Torvalds in the U.S. and other countries.

The key components and capabilities provided by the OpenAMP Framework include:

- **remoteproc** — This component allows for the Life Cycle Management (LCM) of remote processors from software running on a master processor. The remoteproc API provided by the OpenAMP Framework is compliant with the remoteproc infrastructure present in upstream Linux 3.4.x kernel onward. The Linux remoteproc infrastructure and API was first implemented by Texas Instruments.
- **RPMsg** – The RPMsg API enables Inter Processor Communications (IPC) between independent software contexts running on homogeneous or heterogenous cores present in an AMP system. This API is compliant with the RPMsg bus infrastructure present in upstream Linux 3.4.x kernel onward. The Linux RPMsg bus and API infrastructure was first implemented by Texas Instruments.

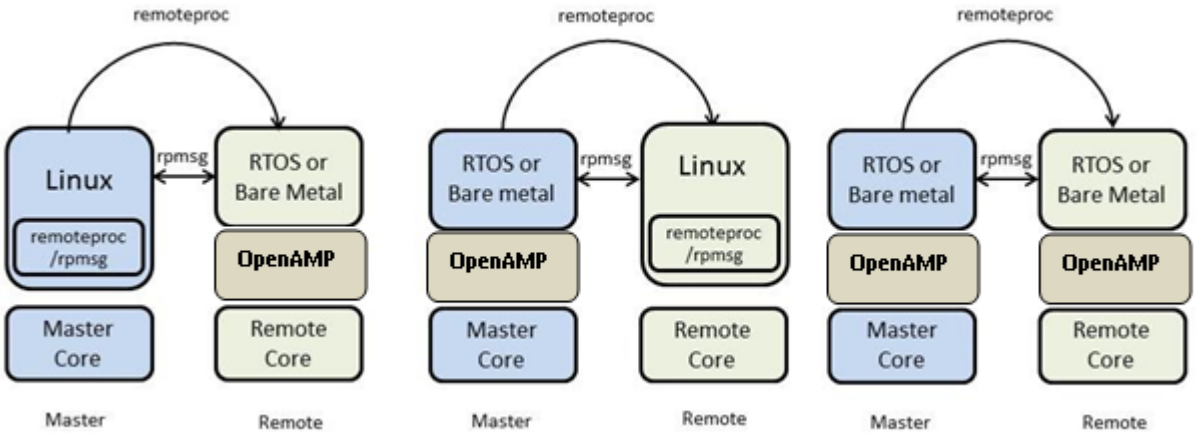
Texas Instruments' remoteproc and RPMsg infrastructure available in the upstream Linux kernel today enable the Linux applications running on a master processor to manage the life cycle of remote processor/firmware and perform IPC with them. However, there is no open-source API/software available that provides similar functionality and interfaces for other possible software contexts (RTOS- or bare metal-based applications) running on the remote processor to communicate with the Linux master. Also, AMP applications may require RTOS- or bare metal-based applications to run on the master processor and be able to manage and communicate with various software environments (RTOS, bare metal, or even Linux) on the remote processor.

The OpenAMP Framework fills these gaps. It provides the required LCM and IPC infrastructure from the RTOS and bare metal environments with the API conformity and functional symmetry available in the upstream Linux kernel. As in upstream Linux, the OpenAMP Framework's remoteproc and RPMsg infrastructure uses virtio as the transport layer/abstraction.

Figure 1-1 shows the various software environments/configurations supported by the OpenAMP Framework. As shown in this illustration, the OpenAMP Framework can be used with RTOS or bare metal contexts on a remote processor to communicate with Linux applications (in kernel space or user space) or other RTOS/bare metal-based applications running on the master processor through the remoteproc and RPMsg components.

The OpenAMP Framework also serves as a stand-alone library that enables RTOS and bare metal applications on a master processor to manage the life cycle of remote processor/firmware and communicate with them using RPMsg.

Figure 1-1. Managing Remote Processes with the OpenAMP framework



In addition to providing a software framework/API for LCM and IPC, the OpenAMP Framework supplies a proxy infrastructure that provides a transparent interface to remote contexts from Linux user space applications running on the master processor. The proxy application hides all the logistics involved in bringing-up the remote software context and its shutdown sequence. In addition, it supports RPMsg-based Remote Procedure Calls (RPCs) from remote context. A retargeting API available from the remote context allows C library system calls such as "_open", "_close", "_read", and "_write" to be forwarded to the proxy application on the master for service.

For more information on this infrastructure and its capabilities, see Figure 5-1 on page 60.

In addition to the core capabilities, the OpenAMP Framework contains abstraction layers (porting layer) for migration to different software environments and new target processors/platforms

Chapter 2
System-Wide Considerations for Using OpenAMP Framework

AMP systems could either be supervised (using a hypervisor to enforce isolation and resource virtualization) or unsupervised (modifying each participating software context to ensure best-effort isolation and cooperative usage of shared resources). With unsupervised AMP systems, there is no strict isolation or supervision of shared resource usage.

Take the following system-wide considerations into account to develop unsupervised AMP systems using the OpenAMP framework:

Note

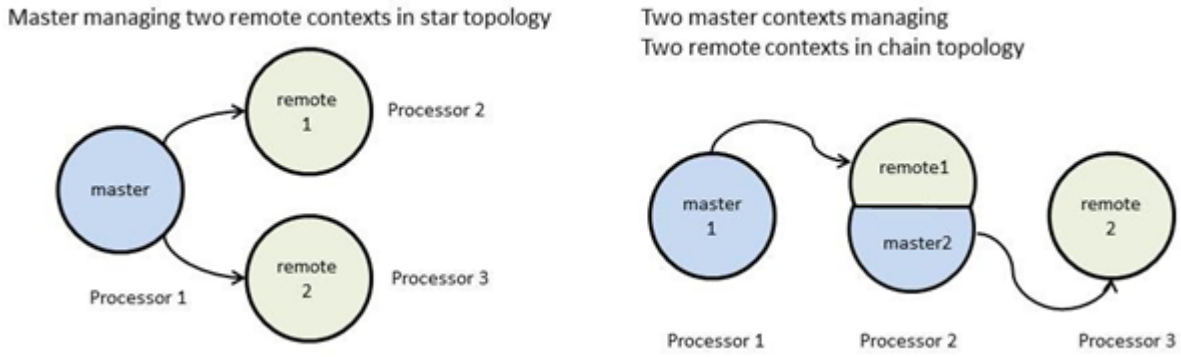
Usage of OpenAMP Framework for supervised AMP systems is not covered in this document.

- Determine system architecture/topology

  The OpenAMP framework implicitly assumes master-slave (remote) system architecture. The topology for the master-slave (remote) architecture should be determined; either star, chain, or a combination. Figure 2-1 shows some simple use cases.

  - Case 1 — A single master software context on processor 1 controlling life cycle and communicating with two independent remote software contexts on processors 2 and 3, in star topology,
  - Case 2 — Master software context 1 on processor 1 brings up remote software context 1 on processor 2. This context acts as master software context 2 for remote software context 2 on processor 3, in chain topology.

Figure 2-1. System Topology Types



- Determine system and IO resource partitioning

  Various OSs, RTOSs, and bare metal environments have their own preferred mechanisms for discovering platform-specific information such as available RAM memory, available peripheral IO resources (their memory-mapped IO region), clocks, interrupt resources, and so forth.

  For example, the Linux kernel uses device trees and bare metal environment typically define platform-specific device information in headers or dedicated data structures that would be compiled into the application.

  To ensure mutually-exclusive usage of unshared system (memory) and IO resources (peripherals) between the participating software environments in an AMP system, you are required to partition the resources so that each software environment is only aware of the resources that are available to it. This would involve, for example, removing unused resource nodes and modifying the available memory definitions from the device tree sources, platform definition files, headers, and so forth, to ensure best-effort partitioning of system resources.

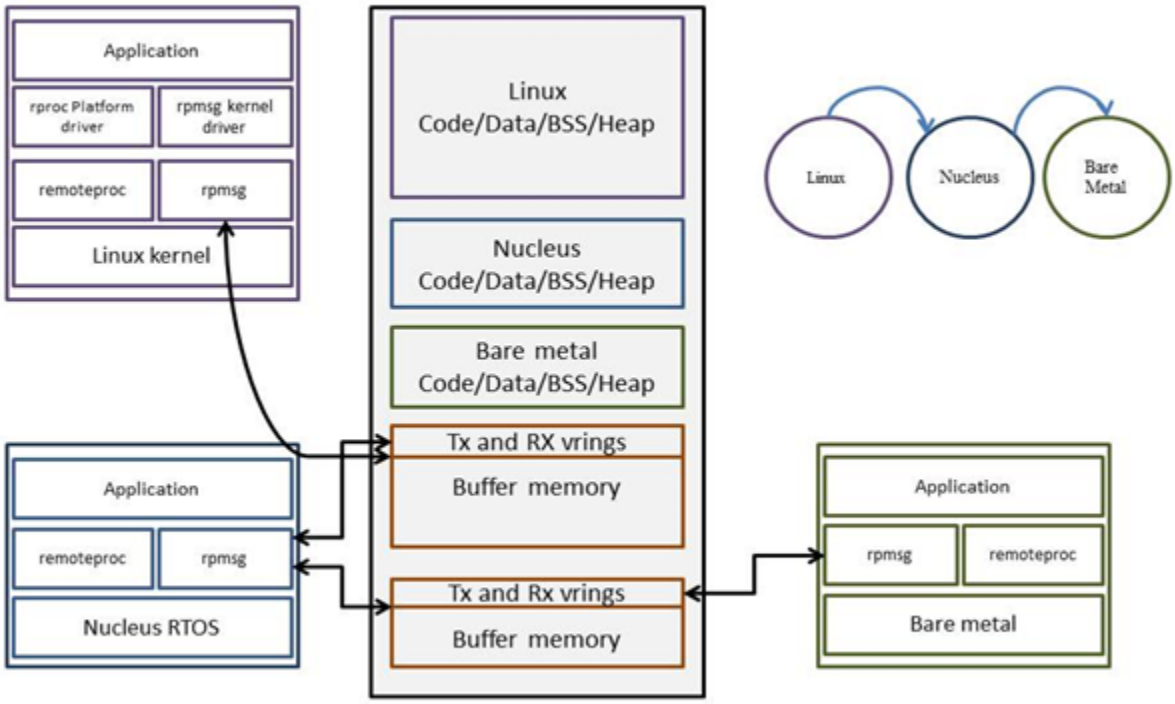
- Determine memory layout

  For the purpose of this description, assume you are using the Zynq SOC used in AMP system architecture with SMP Linux running on the dual Cortex A9 cores, and a RTOS on one instance of Microblaze soft core, and bare metal on another instance of Microblaze soft core in the fabric.

  To develop an AMP system using the OpenAMP Framework, it is important to determine the memory regions that would be owned and shared between each of the participating software environments in the AMP system. For example, in a configuration such as this, the memory address ranges owned (for code/data/bss/heap) by each participating OS or bare metal context, and the shared memory regions to be used by IPC mechanisms (virtio rings and memory for data buffers) needs to be determined. Memory alignment requirements should be taken into consideration while making this determination.

Figure 2-2 illustrates the memory layout for Linux master/Nucleus RTOS-based remote application, and Nucleus RTOS-based master/bare metal-based remote application in chain configuration. After the memory layout is determined, update the platform specific data accessible using the Hardware Interface Layer (HIL) to reflect the memory layout of choice.

Figure 2-2. Determining the Memory Layout in an AMP System



- Ensure cooperative usage of shared resources between software environments in the AMP system

  For the purpose of this discussion, assume you are using a Linux master/bare metal-based remote system configuration.

  The interrupt controller is typically a shared resource in multicore SoCs. It is general practice for OSs to reset and initialize (clear and disable all interrupts) the interrupt controller during their boot sequence given the general assumption that the OS would own the entire system. This will not work in AMP systems; if an OS in remote software context resets and initializes the interrupt controller, it would catastrophically break the master software contexts run time since the master context could already be using the interrupt controller to manage its interrupt resources. Therefore, remote software environments should be patched such that they cooperatively use the interrupt controller (for example, do not reset/clear/disable all interrupts blindly but initialize only the interrupts that belong to the remote context). Ensure the timer peripheral used by the

remote OS/RTOS context is different from the one used by the master software context so the individual run-times do not interfere with each other.

Chapter 3
The remoteproc Component

The remoteproc APIs provided by the OpenAMP Framework allow software applications running on the master processor to manage the life cycle of a remote processor and its software context. A complete description of the remoteproc workflow and APIs are provided.

Concepts

The remoteproc APIs provide life cycle management of remote processors by performing five essential functions.

- Allow the master software applications to load the code and data sections of the remote firmware image to appropriate locations in memory for in-place execution
- Release the remote processor from reset to start execution of the remote firmware
- Establish RPMsg communication channels for run-time communications with the remote context
- Shut down the remote software context and processor when its services are not needed
- Provide an API for use in the remote application context that allows the remote applications to seamlessly initialize the remoteproc system on the remote side and establish communication channels with the master context

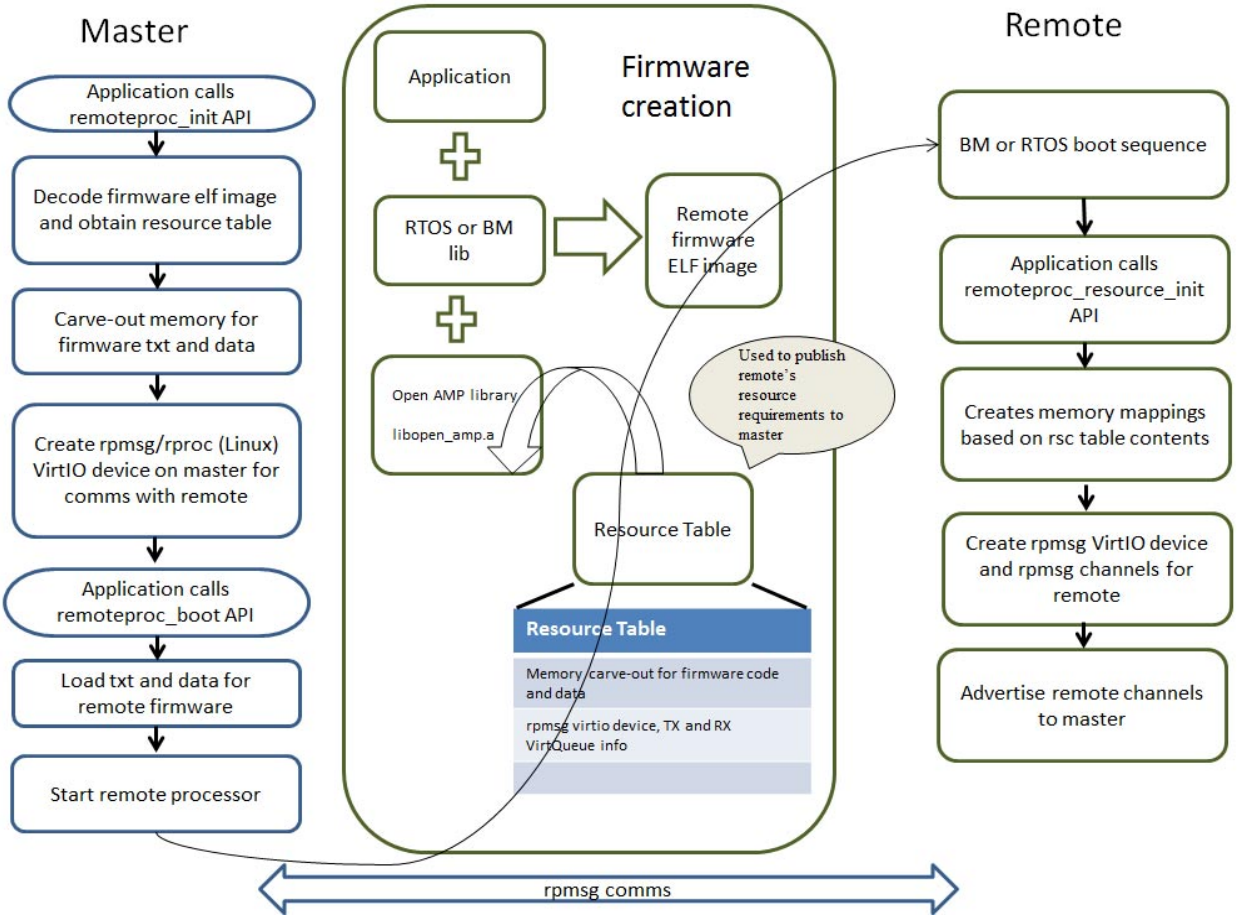
Note

The remoteproc infrastructure provided by the OpenAMP framework is for use with RTOS and bare metal environments only in master or remote configurations. If the AMP use case requires Linux OS as the master, use the upstream Linux remoteproc infrastructure. Refer to Figure 1-1 on page 10.

The remoteproc component currently supports Executable and Linkable Format (ELF) for the remote firmware; however, the framework can be easily extended to support other image formats. The remote firmware image publishes the system resources it requires to remoteproc on the master using a statically linked resource table data structure. The resource table data structure contains entries that define the system resources required by the remote firmware (for example, contiguous memory carve-outs required by remote firmware's code and data sections), and features/functionality supported by the remote firmware (like virtio devices and their configuration information required for RPMsg-based IPC).

The remoteproc APIs on the master processor use the information published through the firmware resource table to allocate appropriate system resources and to create virtio devices for IPC with the remote software context. Figure 3-1 illustrates the resource table usage.

Figure 3-1. remoteproc Conceptual Diagram



When the application on the master calls to the remoteproc_init API, it performs the following:

- Causes remoteproc to fetch the firmware ELF image and decode it
- Obtains the resource table and parses it to handle entries
- Carves out memory for remote firmware before creating virtio devices for communications with remote context

The master application then performs the following actions:

1. Calls the remoteproc_boot API to boot the remote context
2. Locates the code and data sections of the remote firmware image
3. Releases the remote processor to start execution of the remote firmware.

After the remote application is running on the remote processor, the remote application calls the remoteproc_resource_init API to create the virtio/RPMsg devices required for IPC with the master context. Invocation of this API causes remoteproc on the remote context to use the rpmsg name service announcement feature to advertise the rpmsg channels served by the remote application.

The master receives the advertisement messages and performs the following tasks:

1. Invokes the channel created callback registered by the master application
2. Responds to remote context with a name service acknowledgement message

After the acknowledgement is received from master, remoteproc on the remote side invokes the RPMsg channel-created callback registered by the remote application. The RPMsg channel is established at this point. All RPMsg APIs can be used subsequently on both sides for run time communications between the master and remote software contexts.

To shut down the remote processor/firmware, the remoteproc_shutdown API is to be used from the master context. Invoking this API with the desired remoteproc instance handle asynchronously shuts down the remote processor. Using this API directly does not allow for graceful shutdown of remote context.

For gracefully bringing down the remote context, the following steps can be performed:

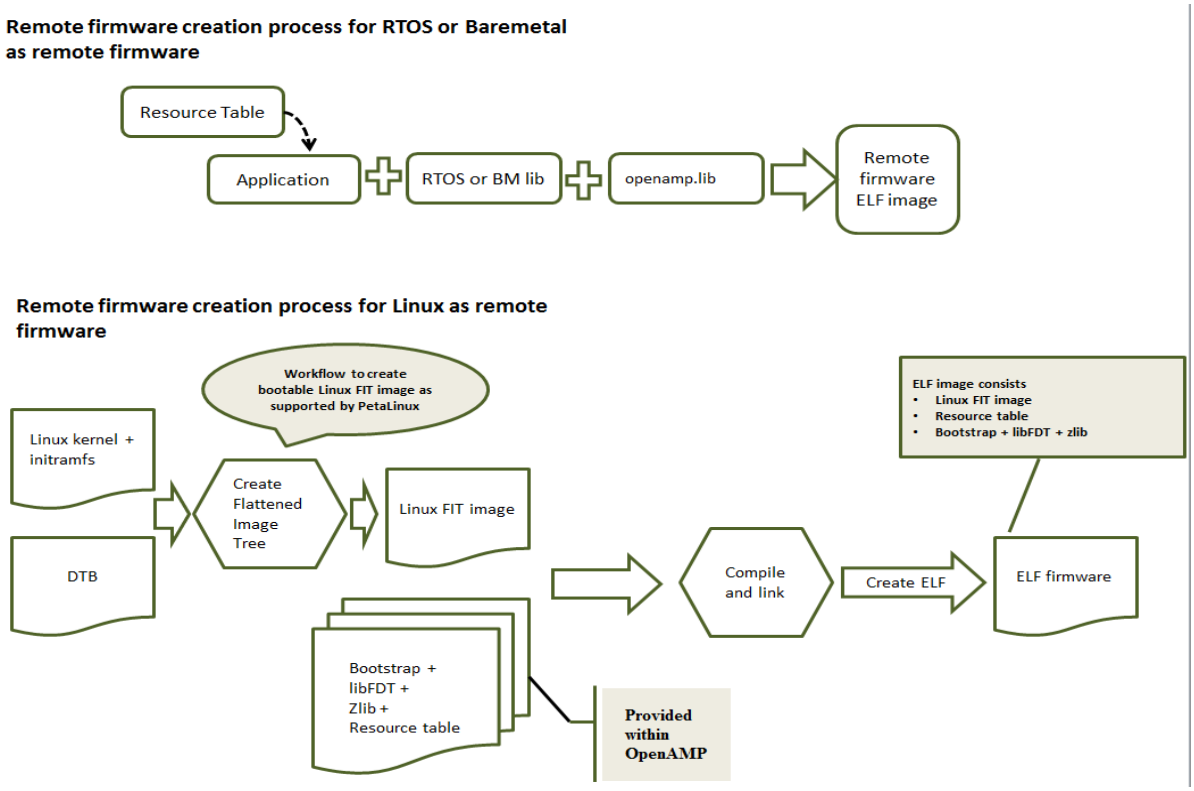
1. The master application sends an application-specific shutdown message to the remote context
2. The remote application cleans up application resources, sends a shutdown acknowledge to master, and invokes remoteproc_resource_deinit API to deinitialize remoteproc on the remote side.
3. On receiving the shutdown acknowledge message, the master application invokes the remoteproc_shutdown API to shut down the remote processor and deinitialize remoteproc using remoteproc_deinit on its side.

Creation and Boot of Remote Firmware Using remoteproc

You can create and boot remote firmware for Linux, RTOS, and bare metal-based remote applications using remoteproc. The following procedure provides general steps for creating and executing remote firmware on a supported platform.

Figure 3-2 illustrates the remote firmware creation process.

Figure 3-2. The Remote Firmware Creation Process



Defining the Resource Table and Creating the Remote ELF Image

Creating a remote image through remoteproc begins by defining the resource table and creating the remote ELF image.

Procedure

1. Define the resource table structure in the application. The resource table must minimally contain carve-out and VirtIO device information for IPC.

As an example, please refer to the resource table defined in the bare metal remote echo test application at *<open_amp>/apps/tests/remote/baremetal/echo_test/rsc_table.c*. The resource table contains entries for memory carve-out and virtio device resources. The memory carve-out entry contains info like firmware ELF image start address and size. The virtio device resource contains virtio device features, vring addresses, size, and alignment information. The resource table data structure is placed in the resource table section of remote firmware ELF image using compiler directives.

2. After defining the resource table and creating the OpenAMP Framework library, link the remote application with the RTOS or bare metal library and the OpenAMP Framework library to create a remote firmware ELF image capable of in-place execution from its pre-determined memory region. (The pre-determined memory region is determined according to guidelines provided by section.)

3. For remote Linux, step 1 describes modifications to be made to the resource table. Figure 3-2 on page 18 shows the high level steps involved in creation of the remote Linux firmware image. The flow leverages supported Petalinux workflows to create a Linux FIT image that encapsulates the Linux kernel image, Device Tree Blob (DTB), and initramfs.

   The user applications and kernel drivers required on the remote Linux context could be built into the initramfs using supported Petalinux workflows or moved to the remote root file system as needed after boot. The FIT image is linked along with a boot strap package provided within the OpenAMP Framework. The bootstrap implements the functionality required to decode the FIT image (using libfdt), uncompress the Linux kernel image (using zlib) and locate the kernel image, initramfs, and DTB in RAM. It can also set up the ARM general purpose registers with arguments to boot Linux, and transfer control to the Linux entry point.

Making Remote Firmware Accessible to the Master

After creating the remote firmware's ELF image, you need to make it accessible to remoteproc in the master context.

Procedure

1. If the RTOS- or bare metal-based master software context has a file system, place this firmware ELF image in the file system.

2. Implement the get_firmware API in *config.c* (in the *OPENAMP/porting/config/* directory) to fetch the remote firmware image by name from the file system.

3. For AMP use cases with Linux as master, place the firmware application in the root file system for use by Linux remoteproc platform drivers.

In the OpenAMP Framework reference port to Zynq ZC702EVK, the bare metal library used by the master software applications do not include a file system. Therefore, the remote image is packaged along with the master ELF image. The remote ELF image is converted to an object file using “objcpy” available in the “GCC bin-utils”. This object file is further linked with the master ELF image.

The remoteproc component on the master uses the start and end symbols from the remote object files to get the remote ELF image base and size. Since the logistics used by the master to obtain a remote firmware image is deployment specific, the config_get_firmware API in *config.c* of the *OPENAMP/porting/config/* directory implements all the logistics described in this procedure to enable the OpenAMP Framework remoteproc on the master to obtain the remote firmware image.

You can now use the remoteproc APIs.

remoteproc API Usage

The following sections assumes a simple application for description.

- The application software on the master processor uses remoteproc to load and execute a remote application (remote firmware) on the remote processor
- After the remote application is running, an rpmsg channel is established between the remote and master applications
- The RPMsg APIs are used for IPC

This simple example should serve as a reference for most typical use cases – where the master software context would bring up the remote context, establish communication channel with it, and start the IPC to offload the computation to the remote context.

Using remoteproc APIs From the Master Software Context

Using OpenAMP framework on a software application on the master processor involves bringing up a remote software context and communicating with it. The following steps describe the general procedure.

Procedure

1. Initialize the remoteproc using the remoteproc_init API and provide callback functions for rpmsg channel creation, rpmsg channel destruction, and rpmsg rx callbacks.
2. Boot the image using remoteproc_boot API.
3. Provide implementation of rpmsg_channel_created, rpmsg_channel_destroyed, and rpmsg_rx_cb functions. The functions are listed as follows:

```
void rpmsg_channel_created (struct rpmsg_channel *rp_chnl)
void rpmsg_channel_destroyed(struct rpmsg_channel *rp_chnl)
void rpmsg_rx_cb(struct rpmsg_channel *rp_chnl, void *data, int
                  len, void * priv, unsigned long src)
```

When using the RPMSG framework from the master software context, the application calls remoteproc_init API to bring up the remote software context. When the remote application boots up on the remote processor, a call to remoteproc_resource_init API initializes remoteproc and rpmsg on the remote side, triggering the RPMSG framework on the remote side to send the rpmsg name service message to advertise itself to the master context.

This announcement causes the RPMSG framework on the master to call the rpmsg channel-created callback registered during initialization. The application is notified from the channel-created callback (using methods that make send for the environment) of the availability of a live rpmsg channel to remote context. The application is free to use rpmsg APIs for IPC with remote context from this point onward.

Examples

The code snippet that follows showcases a sample master application that brings up a remote application that echoes all incoming messages back to the sender.

Example 3-1. Master Application With Echoes to Sender

```
#include "open_amp.h"
/* Application provided callbacks */
void rpmsg_channel_created(struct rpmsg_channel *rp_chnl );
void rpmsg_channel_deleted(struct rpmsg_channel *rp_chnl );
void rpmsg_rx_cb_t(struct rpmsg_channel *rp_chnl , void *data ,
                    int len , void * priv , unsigned long src );

/* Globals */
char remote_fw_name[]= "remote_firmware";

int main(int argc, void *argv) {

    struct remote_proc *proc;
    int    idx ,i,ret;

    /* Initialize Remoteproc */
    ret = remoteproc_init((void *) remote_fw_name, rpmsg_channel_created,
            rpmsg_channel_deleted, rpmsg_rx_cb_t, &proc);

    /* Boot remote firmware */
    if(!ret && (proc))
       ret = remoteproc_boot(proc);

    if(!ret)
    {
        /* Block waiting for invocation of rpmsg channel created callback.
         */
       /* In RTOS environments control should block here
          on a blocking primitive, for example, semaphore, which would be
```

```
            released by the rpmsg channel created callback */
          /* In bare metal environments control should block here
             For example, by spinning on a flag to be released by the rpmsg
             channel  created callback */
           wait()/* This is pseudo-code */

          /* rpmsg APIs can be used for IPC from this point onward */

       }

      /* To shut down the remote processor
         asynchronously and de-initialize the system */
      remoteproc_shutdown(proc);
      remoteproc_deinit(proc);
}

void rpmsg_channel_created(struct rpmsg_channel *rp_chnl) {

      /* Release the context blocked on rpmsg channel creation callback
invocation */
}

void rpmsg_channel_deleted(struct rpmsg_channel *rp_chnl) {

      /* Clean up application resources used by rpmsg */
}

void rpmsg_rx_cb_t(struct rpmsg_channel *rp_chnl, void *data, int len,
                   void * priv, unsigned long src) {

      /* Copy received data to application buffer */
      /* Release the context blocked on rpmsg rx callback invocation */
}
```

Using remoteproc APIs From Remote Software Context

Software applications using the OpenAMP framework on a remote processor must initialize the remoteproc and establish communications with the master software context. The steps that follow describe the general procedure.

Procedure

1. Initialize the remoteproc using the remoteproc_resource_init API and provide callback functions for RPMsg channel creation, RPMsg channel destruction, and RPMsg rx callbacks.

   ```
   int remoteproc_resource_init(struct rsc_table_info *rsc_info,
                                rpmsg_chnl_cb_t channel_created,
                                rpmsg_chnl_cb_t channel_destroyed,
                                rpmsg_rx_cb_t default_cb,
                                struct remote_proc** rproc_handle)
   ```

2. Provide implementation of RPMsg channel created, RPMsg channel destroyed, and RPMsg rx callback functions. The functions are listed as follows:

```
void rpmsg_channel_created (struct rpmsg_channel *rp_chnl)
void rpmsg_channel_deleted(struct rpmsg_channel *rp_chnl)
void rpmsg_rx_cb_t(struct rpmsg_channel *rp_chnl, void *data, int
                   len, void * pric, unsigned long src)
```

When the remote application calls the remoteproc_resource_init API, the remoteproc and rpmsg components are initialized on the remote side. On receiving a name service acknowledgement message from the master, the OpenAMP framework on the remote application calls the rpmsg_chnl_cb_t callback. From the channel_created callback, the remote application is notified of the availability of a live RPMsg channel to the master context. The remote application is free to start communications with the master context from this point onward.

Examples

The code snippet below shows a simple echo remote application the uses the OpenAMP framework APIs to echo back data received from the master.

Example 3-2. Remote Application with Echoes to Master

```
#include "open_amp.h"

/* Internal functions */
static void rpmsg_channel_created(struct rpmsg_channel *rp_chnl);
static void rpmsg_channel_deleted(struct rpmsg_channel *rp_chnl);
static void rpmsg_rx_cb_t(struct rpmsg_channel *, void *, int, void *,
unsigned long);

/* Globals */
static struct remote_proc *proc = NULL;
static struct rsc_table_info rsc_info;
extern const struct remote_resource_table resources;

/* Application entry point */
int main() {

    int ret;
    rsc_info.rsc_tab = (struct resource_table *)&resources;
    rsc_info.size = sizeof(resources);

    /* Application specific initialization
    .
    .
    .*/

    /* Initialize remoteproc on the remote side */
    ret = remoteproc_resource_init(&rsc_info, rpmsg_channel_created,
                                   rpmsg_channel_deleted, rpmsg_rx_cb_t,
                                   &proc);
    if (ret) {
        printf("Error during initialization\r\n");
    }

    /* Block waiting for invocation of rpmsg channel created callback. */
```

```
    /* In RTOS environments control should block here on a blocking
       primitive, for example, semaphore, which would be released by the
       rpmsg channel created callback */
    /* In bare metal environments control should block here
       For example, by spinning on a flag to be released by the rpmsg
       channel created callback */
        wait()/* This is pseudo-code */

    /* rpmsg APIs can be used for IPC from this point onward */

    return 0;
}

static void rpmsg_channel_created(struct rpmsg_channel *rp_chnl)
{

    /* New channel created, save its handle for subsequent reference */
}

static void rpmsg_channel_deleted(struct rpmsg_channel *rp_chnl)
{

   /* perform any clean up required */
}

static void rpmsg_rx_cb_t(struct rpmsg_channel *rp_chnl, void *data,
                          int len,
                          void * priv, unsigned long src)
{
   /* Echo data back to master*/
    rpmsg_send(rp_chnl, data, len);
}
```

remoteproc API Functions

The OpenAMP framework provides the following functions for using the remoteproc API.

- remoteproc_init
- remoteproc_deinit
- remoteproc_boot
- remoteproc_shutdown
- remoteproc_resource_init
- remoteproc_resource_deinit

remoteproc_init

Target Files:

- Prototype definition — *open_amp/remoteproc/remoteproc.h*
- Function definition — *open_amp/remoteproc/remoteproc.c*

The remoteproc_init API is meant to be used on the master processor. It is a non-blocking call that returns a status and handle to remoteproc instance on successful execution.

Usage

```
int remoteproc_init(char             *fw_name,
                    rpmsg_chnl_cb_t  channel_created,
                    rpmsg_chnl_cb_t  channel_destroyed,
                    rpmsg_rx_cb_t    default_cb,
                    struct           remote_proc** rproc_handle);
```

Arguments

- fw_name

  IN direction — The name of the firmware to load
- channel_created

  IN direction — The RPMsg channel creation callback
- channel_destroyed

  IN direction — The RPMsg channel deletion callback
- default_cb

  IN direction — The default rx callback for the RPMsg channel
- rproc_handle

  OUT direction — The pointer to a new remoteproc instance

Return Values

- RPROC_SUCCESS

  The initialization completed successfully.
- RPROC_ERR_NO_RSC_TABLE

  The resource table is not present in the ELF file.
- RPROC_ERR_CPU_ID

  The CPU does not exist for the given firmware name.
- RPROC_ERR_NO_MEM

  An out-of-memory error occurred.

Description

This call performs the following operations:

- Consults HIL and identifies the firmware definition
- Populates the platform data structure with relevant HIL parameters
- Parses firmware and obtains resource table
- Reserves memory required for firmware based on resource table entries

Related Topics

remoteproc API Functions

remoteproc_deinit

Target Files:

- Prototype definition — *open_amp/remoteproc/remoteproc.h*
- Function definition — *open_amp/remoteproc/remoteproc.c*

This API is meant to be used on the master processor. It is a non-blocking call that returns a status.

Usage

```
int remoteproc_deinit (struct remote_proc *rproc);
```

Arguments

- rproc

  IN direction — The pointer to a remoteproc instance to deinitialize

Return Values

- RPROC_SUCCESS

  The deinitialization completed successfully.

Description

This API performs the following operations:

- Free up memory reserved for firmware
- Reclaim memory allocated for hw_proc and remote_proc data structures

Related Topics

remoteproc API Functions

remoteproc_boot

Target Files:

- Prototype definition — *open_amp/remoteproc/remoteproc.h*
- Function definition — *open_amp/remoteproc/remoteproc.c*

The API is meant to be used on the master processor. It is a non-blocking call that returns a status.

Usage

```
int remoteproc_boot(struct remote_proc *rproc);
```

Arguments

- rproc

  IN direction — The pointer to the remoteproc instance to boot

Return Values

- RPROC_SUCCESS

  The initialization completed successfully.

- RPROC_ERR_PARAM

  An invalid parameter was passed.

- RPROC_ERR_LOADER

  An error occurred while loading the ELF image.

Description

This call performs the following operations:

- Copies the firmware image's text and data sections to memory carved out for firmware sections by remoteproc_init
- Releases the remote processor from reset
- Creates the rpmsg_virtio_device for the master node

Related Topics

remoteproc API Functions

remoteproc_shutdown

Target Files:

- Prototype definition — *open_amp/remoteproc/remoteproc.h*
- Function definition — *open_amp/remoteproc/remoteproc.c*

This API is meant to be used on the master processor to shut down the remote processor. It is a non-blocking call that returns a status.

Usage

```
int remoteproc_shutdown(struct remote_proc *rproc);
```

Arguments

- rproc

  IN direction — The pointer to the remoteproc instance to shut down

Return Values

- RPROC_SUCCESS

  The shutdown completed successfully.

Description

This call performs the following operations:

- Powers down the remote processor
- Reclaims resources

Related Topics

remoteproc API Functions

remoteproc_resource_init

Target Files:

- Prototype definition — *open_amp/remoteproc/remoteproc.h*
- Function definition — *open_amp/remoteproc/remoteproc.c*

This API is meant to be used on the remote processor. It is a non-blocking call that returns a status.

Usage

```
int remoteproc_resource_init(struct rsc_table_info *rsc_info,
                             rpmsg_chnl_cb_t channel_created,
                             rpmsg_chnl_cb_t channel_destroyed,
                             rpmsg_rx_cb_t   default_cb,
                             struct          remote_proc** rproc_handle);
```

Arguments

- rsc_info

  IN direction - pointer to resource table info control block
- channel_created

  IN direction — The RPMsg channel creation callback
- channel_destroyed

  IN direction — The RPMsg channel deletion callback
- default_cb

  IN direction — The default rx callback for the RPMsg channel
- rproc_handle

  OUT direction — The pointer to a new remoteproc instance

Return Values

- RPROC_SUCCESS

  The initialization completed successfully.
- RPROC_ERR_NO_RSC_TABLE

  The resource table is not present in the ELF file.
- RPROC_ERR_CPU_ID

  An invalid CPU ID was received from the HIL.
- RPROC_ERR_NO_MEM

  An out-of-memory error occurred.
- RPROC_ERR_PARAM

An invalid parameter was passed.

Description

This call performs the following operations:

- Consults HIL and identifies the CPU ID from the platform definition
- Populates the platform data structure with relevant HIL parameters
- Creates and initializes the RPMsg remote device on the remote side
- Parses firmware and obtains the resource table to make requested MMIO mappings (optional step)
- Creates, initializes, and provides the rproc handle to the application

Related Topics

remoteproc API Functions

remoteproc_resource_deinit

Target Files:

- Prototype definition — *open_amp/remoteproc/remoteproc.h*
- Function definition — *open_amp/remoteproc/remoteproc.c*

This API is meant to be used on the remote processor for deinitializing the remoteproc resources used by the remote firmware. It is a non-blocking call that returns a status.

Usage

```
int remoteproc_deinit(struct remote_proc *rproc);
```

Arguments

- Rproc

  IN direction — The pointer to the remoteproc instance to deinitialize

Return Values

- RPROC_SUCCESS

  The deinitialization completed successfully.

Description

This call performs the following operations:

- Reclaims memory allocated for hw_proc, rproc, and remote_device data structures

Related Topics

remoteproc API Functions

remoteproc Configurable Options

Some remoteproc parameters allow you to configure certain options through their corresponding header files.

RPROC_BOOT_DELAY

File: *open_amp\remoteproc\remoteproc.h*

This is the time in milliseconds defined for remotproc on the master to wait to allow the remote context to boot and initialize remoteproc. This parameter is required to make sure that the RPMsg is initialized on the remote application, otherwise the notification from the master is lost and the remote does not send the name service announcement.

Related Topics

remoteproc API Functions

Chapter 4
The RPMsg Component

The RPmsg APIs provided by the OpenAMP Framework allow RTOS, or bare metal-based applications/drivers running on master and/or remote processors, to perform IPC in an AMP configuration. The RPMsg component only implements the end-user facing APIs and defines the protocol (message header format) component for inter processor communications. The OpenAMP Framework implements a VirtIO-based transport abstraction on which RPMsg performs shared memory based IPC. For VirtIO details please refer to “Virtio Concepts and RPMsg Usage” on page 71.

RPMsg Channel

Every remote core in RPMsg component is represented by RPMsg device that provides a communication channel between master and remote, hence RPMsg devices are also known as channels.

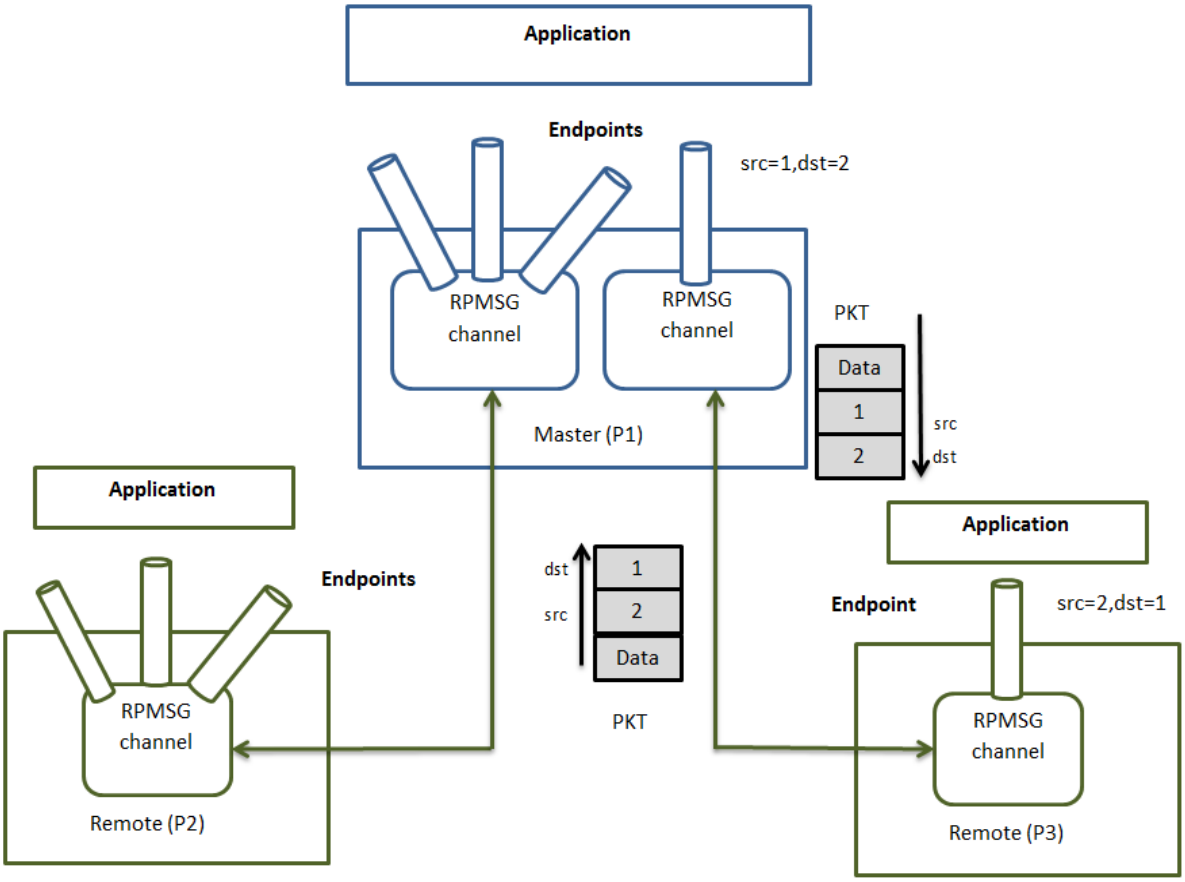
RPMsg channel is identified by the textual name and local (source) and destination address. The RPMsg framework keeps track of channels using their names.

RPMsg Endpoint

RPMsg endpoints provide logical connections on top of RPMsg channel. It allows the user to bind multiple rx callbacks on the same channel.

Every RPMsg endpoint has a unique src address and associated call back function. When an application creates an endpoint with the local address, all the further inbound messages with the destination address equal to local address of endpoint are routed to that callback function. Every channel has a default endpoint which enables applications to communicate without even creating new endpoints.

Figure 4-1. RPMsg Endpoints



RPMsg Header

Every RPMsg transfer starts with the RPMsg header, which contains addresses of Source and Destination endpoints and payload information.

The RPMsg driver routes the message using the destination address. The header is provided below:

```
struct rpmsg_hdr {
   unsigned long src;
   unsigned long dst;
   unsigned long reserved;
   unsigned short len;
   unsigned short flags;
   unsigned char data[0];
} __attribute__((packed));
```

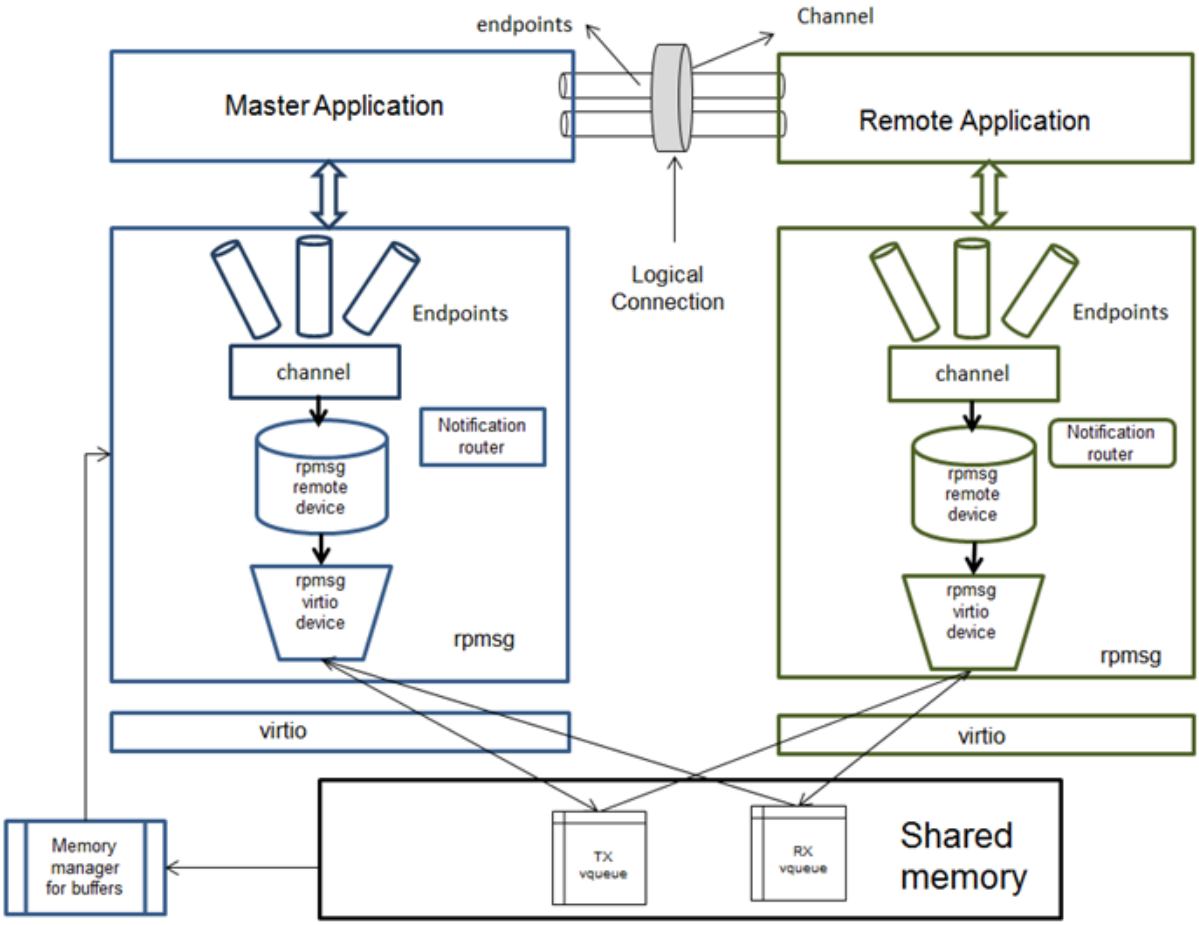
The RPMsg channel and endpoint concept is depicted in Figure 4-1. The example considers a hypothetical multicore system with core IDs P1, P2 and P3. P1 is the master and P2 and P3 are

remotes. The RPMsg packet transmitted from Master contains src address of Remote (P3) endpoint as destination. Whereas the message from remote contains src address of Master in its destination field. The RPMsg stack on respective cores routes the message to application based on the destination address.

OpenAMP Framework RPMsg Driver

Next Figure shows major components present in the OpenAMP Framework RPMsg driver. Communicating cores in the system are represented by the remote device. The remote device encapsulates VirtIO device that provides transport services for RPMsg driver. Moreover, it contains reference to RPMsg channels and endpoints associated with the channels. RPMsg endpoints provide logical connections to remote cores on top of RPMsg channel. A default endpoint is created during initialization and user provided callback is bound to it. remoteproc_init API allows user to bind callbacks to default endpoint. The remoteproc_init API internally uses rpmsg_init function to achieve callback binding. Users are allowed to create additional endpoints for the given channel using rpmsg_create_ept API.

Figure 4-2. RPMsg Driver Components



Applications use APIs provided by the RPMsg driver to communicate with the RPMsg channels. When data is received from the application, RPMsg copies it to internal memory after appending RPMsg header. It then searches the corresponding VirtIO device and places message pointer in virtqueue, followed by a notification. rpmsg_send API is used to send data over default endpoint. rpmsg_sendto, and rpmsg_send_offchannel APIs are used for directing messages to any given application created endpoint.

RPMsg driver generates notifications for applications using callbacks for incoming messages. The notification handling mechanism in RPMsg driver directs the message to applications based on its destination address. There are two types of notifications events: Rx completion and Channel Creation/Deletion events. Rx completion event is generated when data is received from the communication counterpart. The channel creation/deletion events are triggered when Name Service (NS) is received from the remote. Application binds callback to RPMsg driver using rpmsg_create_ept API. A default callback is bound to the RPMsg channel during initialization.

RPMsg API Usage

RPMsg API usage is illustrated using a simple echo application.

The assumptions are as follows:

- The application software on the master processor uses remoteproc to load and execute an echo application (remote firmware) on the remote processor
- After the remote application is running, an rpmsg channel is established between the remote and master applications
- Any message sent by the master application to the remote application using rpmsg APIs is echoed back to the master application

This example should serve as a reference for most typical use cases – where the master software context would bring up the remote context, establish communication channel with it, and start the IPC to offload the computation to the remote context.

RPMsg API Usage From the Master Software Context

After bringing up the remote context, the user application is free to use rpmsg APIs for IPC with remote context from that point onward.

Instructions for bringing up the remote context are present in “remoteproc API Usage” on page 20.

The code snippet in Example 4-1 on page 39 showcases a sample master application that sends messages to a remote, using the rpmsg_send API, then waits for an echo from the remote.

Example 4-1. Master Application With Echoes to Sender

```
#include "open_amp.h"

/* Application provided callbacks */
void rpmsg_channel_created( struct rpmsg_channel *rp_chnl );
void rpmsg_channel_deleted( struct rpmsg_channel *rp_chnl );
void rpmsg_rx_cb_t( struct rpmsg_channel *rp_chnl,
                                          void *data,
                                          int len,
                                          void * priv,
                                          unsigned long src );

/* Globals */
struct rpmsg_channel *app_rp_chnl;
char remote_fw_name []= "remote_firmware";
char s_buff[256];
char r_buff[256];

int main( int argc , void *argv ) {

    struct remote_proc *proc;
    int    idx ,i,ret;

    /* Initialize Remoteproc */
       ret = remoteproc_init((void *) remote_fw_name,
                                      rpmsg_channel_created,
                                      rpmsg_channel_deleted,
                                      rpmsg_rx_cb_t, &proc);

    /* Boot remote firmware */
    if(!ret && (proc))
       ret = remoteproc_boot(proc);

    if(!ret)
    {
        /* Block waiting for invocation of rpmsg channel created callback.
         */

       /* In RTOS environments control should block here
          on a blocking primitive, for example, semaphore, which would be
          released by the rpmsg channel created callback */

       /* In bare metal environments, control should block here
          by spinning on a flag to be released by the rpmsg channel
          created callback */

       /* This is pseudo-code */
          wait()

       /* Setup the buffer with a pattern */
          memset(s_buff, 0xA5, sizeof(s_buff));


       /* Send data to remote side. */
          rpmsg_send(app_rp_chnl, s_buff, sizeof(s_buff));
```

```
            /* Block waiting for invocation of rpmsg rx callback. */
            /* This is pseudo-code */
               wait();

            /* Verify the data received in r_buff */
         }

        /* Shut down the remote processor and de-initialize the system */
           remoteproc_shutdown(proc);
           remoteproc_deinit(proc);
}

void rpmsg_channel_created(struct rpmsg_channel *rp_chnl) {

    app_rp_chnl = rp_chnl;

    /* Release the context blocked on rpmsg channel creation callback
       invocation */
}

void rpmsg_channel_deleted(struct rpmsg_channel *rp_chnl) {

    /* Clean up application resources used by rpmsg */
}

void rpmsg_rx_cb_t(struct rpmsg_channel *rp_chnl,
                                        void *data,
                                        int len,
                                        void * priv,
                                        unsigned long src) {

    /* Copy received data to application buffer */
       memcpy(r_buff,data,len);

    /* Release the context blocked on rpmsg rx callback invocation */
}
```

RPMsg API Usage From Remote Software Context

After the remote image is up and running, the user application waits for the channel creation callback. The remote application is free to start communications with the master context from that point onward.

Instructions for bringing up the remote context are present in “remoteproc API Usage” on page 20.

The code example below shows an echo remote application that uses the RPMsg APIs to echo data received from the master.

```
#include "open_amp.h"

/* Internal functions */
static void rpmsg_channel_created(struct rpmsg_channel *rp_chnl);
static void rpmsg_channel_deleted(struct rpmsg_channel *rp_chnl);
```

```
static void rpmsg_rx_cb_t(struct rpmsg_channel *, void *, int, void *,
unsigned long);

/* Globals */
static struct rpmsg_channel *app_rp_chnl;
static struct remote_proc *proc = NULL;
static struct rsc_table_info rsc_info;
extern const struct remote_resource_table resources;

/* Application entry point */
int main()
{
    int ret;
    rsc_info.rsc_tab = (struct resource_table *)&resources;
    rsc_info.size = sizeof(resources);
    /* Application specific initialization
    .
    .
    .*/

    /* Initialize remoteproc on the remote side */
    ret = remoteproc_resource_init(&rsc_info, rpmsg_channel_created,
                                   rpmsg_channel_deleted, rpmsg_rx_cb_t,
                                   &proc);

    if (ret)
    {
        printf("Error during initialization\r\n");
    }

    /* Wait in infinite loop – echo functionality is provided by callback
       functions*/

    while(1)
    {
        sleep(100);
    }

    return 0;
}

static void rpmsg_channel_created(struct rpmsg_channel *rp_chnl)
{
    /* New channel created, save its handle for subsequent reference */
    app_rp_chnl = rp_chnl;
}

static void rpmsg_channel_deleted(struct rpmsg_channel *rp_chnl)
{
   /* perform any clean up required */
}

static void rpmsg_rx_cb_t(struct rpmsg_channel *rp_chnl, void *data,
                          int len, void * priv, unsigned long src)
{
    /* Echo data back to master*/
    rpmsg_send(rp_chnl, data, len);
}
```

RPMsg API Functions

The RPMSG framework provides following RPMsg APIs for messaging:

- rpmsg_send
- rpmsg_sendto
- rpmsg_send_offchannel
- rpmsg_trysend
- rpmsg_trysendto
- rpmsg_trysendoffchannel
- rpmsg_get_buffer_size
- rpmsg_create_ept
- rpmsg_destroy_ept
- rpmsg_chnl_cb_t
- rpmsg_rx_cb_t

rpmsg_send

Target Files:

- Function definition — *open_amp/rpmsg/rpmsg.h*

This function sends user provided data of specified size to the default endpoint associated with the RPMsg channel. If no TX buffers are available, the API will either block until one becomes available or a timeout of 15 seconds elapses. This function copies the data in its internal buffer so the caller can reclaim the buffer once this call has been returned.

Usage

```
static inline int rpmsg_send(struct rpmsg_channel *rpdev,
                             void                 *data,
                             int                  len);
```

Arguments

- rpdev

  IN direction — The pointer to the RPMsg device

- data

  IN direction — The pointer to the buffer containing data

- len

  IN direction — The size of the data, in bytes, to transmit

Return Values

- RPMSG_SUCCESS

  The operation completed successfully.

- RPMSG_ERR_PARAM

  An invalid parameter was received.

- RPMSG_ERR_DEV_STATE

  The remote device is in an invalid state. The device is not in a ready state yet.

- RPMSG_ERR_NO_MEM

  An out-of-memory error was received.

- RPMSG_ERR_NO_BUFF

  No buffer is present in the virtqueue.

Related Topics

RPMsg API Functions

rpmsg_sendto

Target Files:

- Function definition — *open_amp/rpmsg/rpmsg.h*

This function sends user provided data of specified size to the RPMsg device endpoint with specfied destination address. In case there are no TX buffers available, the API will block until one becomes available, or a timeout of 15 seconds elapses. This function copies the data in its internal buffer so the caller can reclaim the buffer once this call has been returned.

Usage

```
static inline int rpmsg_sendto(struct rpmsg_channel *rpdev,
                  void                 *data,
                  int                  len,
                  unsigned long        dst);
```

Arguments

- rpdev

  IN direction — The pointer to the RPMsg device

- data

  IN direction — The pointer to the buffer containing data

- len

  IN direction — The size of the data, in bytes, to transmit

- dst

  IN direction — The destination address of the message

Return Values

- RPMSG_SUCCESS

  The operation completed successfully.

- RPMSG_ERR_PARAM

  An invalid parameter was received.

- RPMSG_ERR_DEV_STATE

  The remote device is in an invalid state. The device is not in a ready state yet.

- RPMSG_ERR_NO_MEM

  An out-of-memory error was received.

- RPMSG_ERR_NO_BUFF

  No buffer is present in the virtqueue.

Related Topics

RPMsg API Functions

rpmsg_send_offchannel

Target Files:

- Function definition — *open_amp/rpmsg/rpmsg.h*

This function sends a message using explicit src/dst addresses. In case there are no TX buffers available, the API will block until one becomes available, or a timeout of 15 seconds elapses. This function copies the data in its internal buffer so the caller can reclaim the buffer once this call has been returned.

Usage

```
static inline int rpmsg_send_offchannel(struct rpmsg_channel *rpdev,
                                        unsigned long        src,
                                        unsigned long        dst,
                                        void                 *data,
                                        int                  len);
```

Arguments

- rpdev

  IN direction — The pointer to the RPMsg device

- src

  IN direction — The source address of the message

- dst

  IN direction — The destination address of the message

- data

  IN direction — The pointer to the buffer containing data

- len

  IN direction — The size of the data, in bytes, to transmit

Return Values

- RPMSG_SUCCESS

  The operation completed successfully.

- RPMSG_ERR_PARAM

  An invalid parameter was received.

- RPMSG_ERR_DEV_STATE

  The remote device is in an invalid state. The device is not in a ready state yet.

- RPMSG_ERR_NO_MEM

  An out-of-memory error was received.

- RPMSG_ERR_NO_BUFF

  No buffer is present in the virtqueue.

Related Topics

RPMsg API Functions

rpmsg_trysend

Target Files:

- Function definition — *open_amp/rpmsg/rpmsg.h*

This function sends user-provided data of specified size to the default endpoint associated with the RPMsg channel. The src and the dst address are that of the RPMsg channel itself. In case there are no TX buffers available, the API will immediately return with an error code. This function copies the data in its internal buffer so the caller can reclaim the buffer once this function has been returned.

Usage

```
static inline int rpmsg_trysend(struct rpmsg_channel *rpdev,
                                void                 *data,
                                int                  len);
```

Arguments

- rpdev

  IN direction — The pointer to the RPMsg device

- data

  IN direction — The pointer to the buffer containing data

- len

  IN direction — The size of the data, in bytes, to transmit

Return Values

- RPMSG_SUCCESS

  The operation completed successfully.

- RPMSG_ERR_PARAM

  An invalid parameter was received.

- RPMSG_ERR_DEV_STATE

  The remote device is in an invalid state. The device is not in a ready state yet.

- RPMSG_ERR_NO_MEM

  An out-of-memory error was received.

- RPMSG_ERR_NO_BUFF

  No buffer is present in the virtqueue.

Related Topics

RPMsg API Functions

rpmsg_trysendto

Target Files:

- Function definition — *open_amp/rpmsg/rpmsg.h*

This function sends user-provided data of specified size to the RPMsg device endpoint with the specfied destination address. In case there are no TX buffers available, the API will immeditately return with an error code. This function copies the data in its internal buffer so the caller can reclaim the buffer once this call has been returned.

Usage

```
static inline int rpmsg_trysendto(struct rpmsg_channel *rpdev,
                    void                *data,
                    int                 len,
                    unsigned long       dst);
```

Arguments

- rpdev

  IN direction — The pointer to the RPMsg channel

- data

  IN direction — The pointer to the buffer containing data

- len

  IN direction — The size of the data, in bytes, to transmit

- dst

  IN direction — The destination address of the message

Return Values

- RPMSG_SUCCESS

  The operation completed successfully.

- RPMSG_ERR_PARAM

  An invalid parameter was received.

- RPMSG_ERR_DEV_STATE

  The remote device is in an invalid state. The device is not in a ready state yet.

- RPMSG_ERR_NO_MEM

  An out-of-memory error was received.

- RPMSG_ERR_NO_BUFF

  No buffer is present in the virtqueue.

Related Topics

RPMsg API Functions

rpmsg_trysendoffchannel

Target Files:

- Prototype definition — *open_amp/rpmsg/rpmsg.h*
- Function definition — *open_amp/rpmsg/rpmsg.c*

This function sends messages to the default endpoint associated with the RPMsg channel. In case there are no TX buffers available, the API will immediately return with an error code. This function copies the data in its internal buffer so the caller can reclaim the buffer once this call has been returned.

Usage

```
int rpmsg_trysendoffchannel(struct rpmsg_channel *rp_chnl,
                            unsigned long        src,
                            unsigned long        dst,
                            char                 *data,
                            int                  len);
```

Arguments

- rp_chnl

  IN direction — The pointer to the RPMsg channel
- src

  IN direction — The source address of the message
- dst

  IN direction — The destination address of the message
- data

  IN direction — The pointer to the buffer containing data
- len

  IN direction — The size of the data, in bytes, to transmit

Return Values

- RPMSG_SUCCESS

  The operation completed successfully.
- RPMSG_ERR_PARAM

  An invalid parameter was received.
- RPMSG_ERR_DEV_STATE

  The remote device is in an invalid state. The device is not in a ready state yet.
- RPMSG_ERR_NO_MEM

An out-of-memory error was received.

- RPMSG_ERR_NO_BUFF

  No buffer is present in the virtqueue.

Related Topics

RPMsg API Functions

rpmsg_get_buffer_size

Target Files:

- Prototype definition — *open_amp/rpmsg/rpmsg.h*
- Function definition — *open_amp/rpmsg/rpmsg.c*

This function returns the size of the buffer that is available for sending messages.

Usage

```
int rpmsg_get_buffer_size(struct rpmsg_channel *rp_chnl);
```

Arguments

- rp_chnl

  IN direction — The pointer to the RPMsg channel

Return Values

- RPMSG_SUCCESS

  The operation completed successfully.

- RPMSG_ERR_PARAM

  An invalid parameter was passed.

- RPMSG_ERR_DEV_STATE

  The remote device is in an invalid state. The device is not in a ready state yet.

Related Topics

RPMsg API Functions

rpmsg_create_ept

Target Files:

- Prototype definition — *open_amp/rpmsg/rpmsg.h*
- Function definition — *open_amp/rpmsg/rpmsg.c*

This function creates a new endpoint for a given RPMsg channel and returns it to the caller.

Usage

```
struct rpmsg_endpoint *rpmsg_create_ept(struct rpmsg_channel *rp_chnl,
                                        rpmsg_rx_cb_t        cb,
                                        void                 *priv,
                                        unsigned long        addr);
```

Arguments

- rp_chnl

  IN direction — The pointer to the RPMsg channel.

- cb

  IN direction — The rx callback function for the endpoint.

- priv

  IN direction — Any private data; provided as a parameter in the callback.

- addr

  IN direction — The local (src) address of the endpoint. If RPMSG_ADDR_ANY is passed as an address, the RPMsg driver chooses the address itself.

Return Values

- RPMSG_NULL

  This value returns only if an error occurs; otherwise, a valid pointer returns.

Related Topics

RPMsg API Functions

rpmsg_destroy_ept

Target Files:

- Prototype definition — *open_amp/rpmsg/rpmsg.h*
- Function definition — *open_amp/rpmsg/rpmsg.c*

This function deletes the RPMsg endpoint and reclaims resources.

Usage

```
void rpmsg_destroy_ept(struct rpmsg_endpoint *rp_ept);
```

Arguments

- rp_ept

  IN direction — The pointer to the RPMsg endpoint to deinitialize

Return Values

None.

Related Topics

RPMsg API Functions

rpmsg_chnl_cb_t

This is a typdef for channel creation/deletion callback that an application registers with the rpmsg driver during calls to remoteproc_init and remoteproc_resource_init functions.

An example channel creation callback function declaration is provided under Usage.

Usage

```
void rpmsg_chnl_cb_t(struct rpmsg_channel *rp_chnl);
```

Arguments

- rp_chnl

  IN direction — The pointer to the created RPMsg channel

Related Topics

RPMsg API Functions

rpmsg_rx_cb_t

Target Files:

- Function definition — *open_amp/rpmsg/rpmsg_core.h*

This is a typedef for the data rx completion callback function that an application must provide during calls to remoteproc_init and rpmsg_create_ept functions. This callback is invoked by the RPMsg driver when data is received. The application must copy the data to the local buffer before returning this callback fucnction.

An example rx callback function declaration is provided under Usage.

Usage

```
void rpmsg_rx_complete (struct rpmsg_channel *rp_chnl,
                        void                 *data,
                        int                  len,
                        void                 *priv,
                        unsigned long        src);
```

Arguments

- rp_chnl

  IN direction — The pointer to the RPMsg channel on which data is received.

- data

  IN direction — The buffer containing received data.

- len

  IN direction — The size of data received, in bytes

- priv

  IN direction — Any private data provided during endpoint creation

- src

  IN direction — The address of the endpoint from which data is received

Related Topics

RPMsg API Functions

RPMsg Configurable Options

Some RPMsg parameters allow you to configure certain options through their corresponding header files.

RPMSG_BUFFER_SIZE

File : *open_amp/rpmsg/rpmsg_core.h*

Buffer size supported by the RPMsg driver. To transmit data size greater than this value, you will have to split it into buffer size blocks.

RPMSG_MAX_VQ_PER_RDEV

File : *open_amp/rpmsg/rpmsg_core.h*

Maximum virtual queues (“virtqueues”) per remote device. Currently only two virtqueues are supported.

RPMSG_NS_EPT_ADDR

File : *open_amp/rpmsg/rpmsg_core.h*

Address of name service endpoint. For Linux master, the address must be same as that defined by the Linux RPMsg bus driver.

RPMSG_ADDR_BMP_SIZE

File : *open_amp/rpmsg/rpmsg_core.h*

Size of the bitmap array used to keep track of free and used endpoint addresses.

Related Topics

RPMsg API Functions

Chapter 5
Proxy Infrastructure

The OpenAmp Framework provides a proxy infrastructure that provides a transparent interface to RTOS and bare metal-based remote contexts from Linux user space applications running on the master processor. Read the following sections for a description of this infrastructure.

Proxy Infrastructure Overview

The proxy application essentially hides all the logistics involved in bringing-up the remote software context and its shutdown sequence. In addition, it supports RPCs from remote context using system calls such as “_open”, “_close”, “_read”, and “_write”. In remote context, these system calls are retargeted to proxy applications running on the master over rpmsg for service.

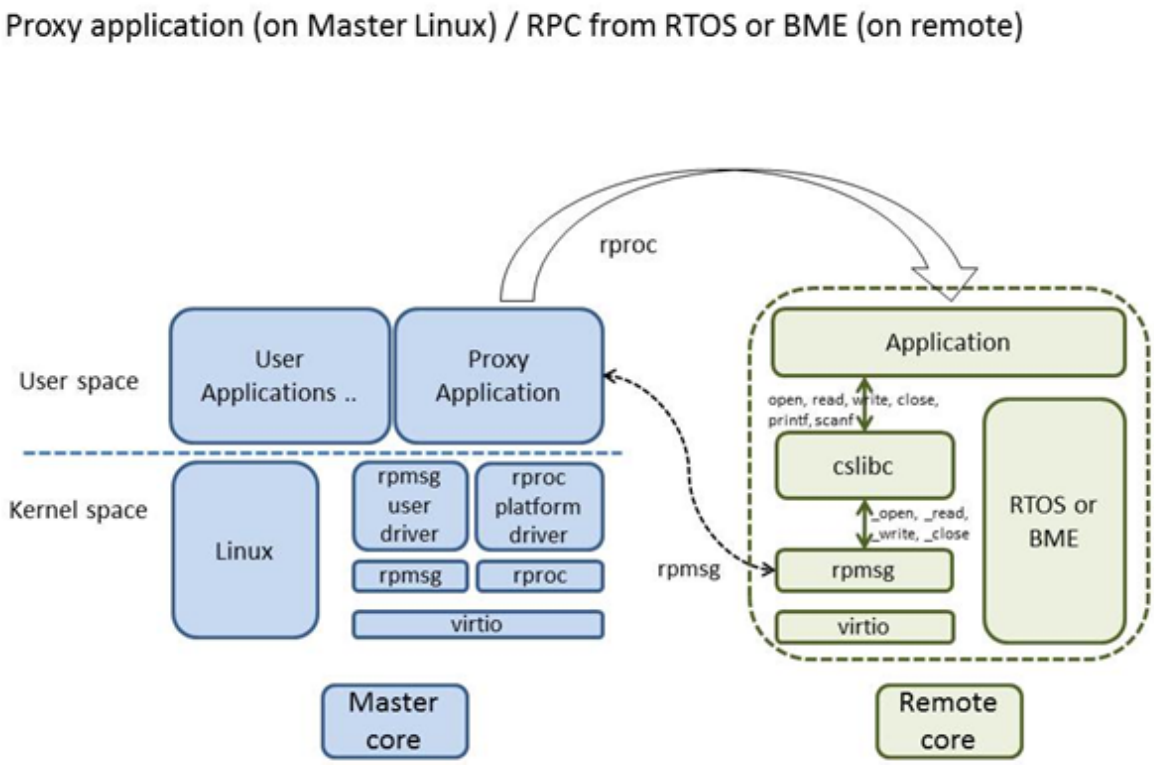
The proxy infrastructure consists of the following:

- A proxy application running as a Linux user space application on the master processor
- An rpmsg-retargeting API available for use from RTOS or bare metal contexts running on the remote processor

The rpmsg-retargeting API allows C library calls from the remote software context to be retargeted over rpmsg-based IPC (as an RPC) to the dedicated proxy application running as a user space Linux application on the master processor. The proxy application handles the remote procedure calls, allowing the remote context to perform useful operations like printf, scanf, and fileIO to STDIO and file handles available on the Linux master. For example, a printf() call from the remote software application context would print the message to the STDOUT (console) of the proxy application running on Linux on the master processor; similarly for scanf, and file IO as well. This capability can be very useful for development and debugging of remote applications.

Figure 5-1 illustrates the proxy infrastructure.

Figure 5-1. The Proxy Infrastructure



Usage of Proxy Infrastructure on Master

On the master Linux OS, the proxy application is to be executed with the path to the remote firmware image.

```
>proxy_app -f <path_to_remote_firmware>
```

For more information on creating a firmware image, see “Creation and Boot of Remote Firmware Using remoteproc” on page 18.

The proxy app performs all logistics required to do the following:

- Load the necessary kernel drivers
- Load and execute the remote firmware application
- Establish rpmsg channel and system endpoint connections with remote context.

After the proxy application is running, system calls such as “_open”, “_close”, “_read”, and “_write” on the remote side are forwarded to the proxy application for service, enabling the

remote context to access the STDIO and file handles available on the master. To terminate the remote context, you can either end the proxy application (SIGTERM), interrupt the application using CTRL + C (SIGINT), or close the console (SIGHUP). This causes the proxy application on master to perform the following:

- Transmit a shutdown request to the remote
- Unload relevant kernel drivers
- Bring the system back to a pristine state

The proxy user space application running on the master requires the rpmsg_proxy_dev_driver kernel driver, which creates the rpmsg_proxy device and exposes the rpmsg proxy services to the proxy application in user space. Along with this, other user developed rpmsg kernel drivers can create application specific character devices to expose rpmsg based IPC services to user space. User space applications access these application-specific rpmsg devices to realize the application IPC needs. The proxy app merely serves as a Linux user space application that hides the logistics of managing the remote firmware from the user and enables debugging of remote applications.

Usage of Proxy Infrastructure on Remote

On the remote side, the proxy infrastructure provides an rpmsg_retarget API. Once the rpmsg channel is established with the master (by invocation of channel creation callback), the application can invoke the rpmsg_retarget_init API. This latter API creates a new rpmsg endpoint (address 127) dedicated to be the system endpoint to forward remote procedure calls to the proxy on the master and process RPC response from the master.

When the RX callback registered with the system endpoint receives a shutdown request from the master, the remote application should invoke the rpmsg_retarget_deinit API, which destroys the system endpoint and shuts down the rpmsg-based retargeting infrastructure used for RPC. The reference implementation provides a sample implementation that showcases the usage of this API.

Chapter 6
OpenAMP Framework Porting Guidelines

The OpenAMP Framework provides abstractions that allow for porting of the OpenAMP Framework to various software environments (operating systems and bare metal environments) and hardware platforms (processors/platforms). The source code for porting components reside in the *OPENAMP/porting/* directory.

Table 6-1 shows the the OpenAMP Framework porting layer information.

Table 6-1. OpenAMP Framework Porting Layers

Directory	Description
OPENAMP/porting/config	System level configuration options
OPENAMP/porting/<platform_name>	Patform porting component
OPENAMP/porting/env	Software environment interface layer that contains abstractions for RTL functions and OS/BM environment features

The high level components, such as RPMSG and Remoteproc, use abstractions provided by the HIL component present in the *OPENAMP/common/hil* directory to access the platform and configuration porting pieces. The environment abstractions are directly used by the high level components.

The description of various files present in the HIL component is provided in the Table 6-2.

Table 6-2. OpenAMP HIL Files

Directory	Description
OPENAMP/common/hil/hil.h	This is generic code that will not require porting. Exposes public interface of HIL to higher level software modules (rpmsg, remoteproc). It also defines interface for platform and config porting components.
OPENAMP/common/hil/hil.c	This is generic code that will not require porting. Implements the HIL APIs that enable higher layers to access the HIL data structures.

The description of various files present in the platform porting component is provided in Table 6-3. Users are expected to provide definitions for porting functions in these files.

Table 6-3. HIL File Changes

File Name	Description
OPENAMP/porting/<platform_name>/platform.h	This is a processor/platform-specific header. You define platform-specific definitions for the new processor/platform in this file.
OPENAMP/porting/<platform_name>/platform.c	This is a processor/platform-specific file in which you are expected to implement the functions defined in the hil_platform_ops function table.
OPENAMP/porting/<platform_name>/platform_info.c	This file consists of APIs that fetch platform-specific information required by the OpenAMP Framework. You are expected to define this information and implement mechanics to obtain this platform specific information based on software environment and hardware platform to be used. The reference implementation puts this file to use to obtain platform-specific information for the ZC702EVK platform, for bare metal-based software environments. For reference, see *<$OPENAMP>/porting/zc702evk/platform_info.c*.

Platform Porting Overview

Platform porting consists of three general steps.

The steps for platform porting are as follows:

1. Implement mechanics for obtaining platform-specific info. This includes the CPU ID and its associated configuration, shared memory regions, IPI, virtio device information, rpmsg channel information, and so forth.
2. Implement platform specific code for enabling and triggering IPIs for rpmsg, defined by hil_platform_ops (*<open_amp>/common/hil/hil.h*).
3. Implement platform specific code for booting and shutting down remote contexts for the remoteproc, defined by hil_platform_ops.

Platform-Specific APIs

Platform information (CPU ID, shared memory, interrupts, and channels information) is obtained during the call to the hil_create_proc API, which invokes platform porting APIs.

The following platform porting APIs are invoked by the hil_create_proc API:

- platform_get_processor_info
- platform_get_processor_for_fw

These APIs are invoked on a per-processor basis to obtain platform-specific information. Ensure you provide implementation of these APIs for each new platform/configuration in the *platform_info.c* file.

platform_get_processor_info

This function accepts a pointer to hil_proc structure and CPU ID as parameters. The successful return from this function should populate all fields of the proc_shm, proc_intr, and proc_chnl control blocks and CPU ID fields of the “hil_proc” data structure. The proc_chnl structure elements are required to be populated only when the OpenAMP Framework is used with remote software contexts. Moreover, for remote contexts, this function is called with the HIL_RSVD_CPU_ID parameter to indicate that the platform information is requested for the master.

Consider an example of platform information provided by bare metal applications. The application defines a hil_proc structure(example_node) and populates only the platform-specific fields leaving rest of fields empty. The platform_get_processor_info function copies the contents of this structure to hil_proc structure passed as a parameter. These nodes must be defined for each core present in the system.

For example, if there is a system with three cores, then for the master context, two such nodes would be defined corresponding to each remote. In the case of the remote, only one node is required (for the master, since each remote can have one master only).

```
struct hil_proc example_node =
{
    /* CPU node for remote context */
    {
        /* CPU ID of master */
        MASTER_CPU_ID,
        /* Shared memory info - Last field is not used currently */
        {
            SHM_ADDR, SHM_SIZE, 0x00
        },
        /* VirtIO device info */
        {
            /* Leave these three fields empty as these are obtained from
            rsc
             * table.
             */
```

```
                0, 0, 0,
                /* Vring info */
               {
                   {
                        /* Provide only vring interrupts info here. Other
                        fields are
                         * obtained from the resource table so leave them
                        empty.
                         */
                        NULL, NULL, 0, 0,
                        {
                            VRING0_IPI_VECT,IPI_PRIORITY,IPI_POLARITY,NULL
                        }
                   },
                   {
                       NULL, NULL, 0, 0,
                       {
                           VRING1_IPI_VECT, IPI_PRIORITY,IPI_POLARITY,NULL
                       }
                   }
               }
           },
           /* Number of RPMSG channels */
           1,
           /* RPMSG channel info */
           {
               {"rpmsg-openamp-demo-channel"}
           },
           /* HIL platform ops table. */
           NULL,
           /* Next three fields are for future use only */
           0,
           0,
           NULL
       }

   int platform_get_processor_info(struct hil_proc *proc , int cpu_id) {
       int idx;
       for(idx = 0; idx < sizeof(proc_table)/sizeof(struct hil_proc); idx++)
   {
           if((cpu_id == HIL_RSVD_CPU_ID) || (proc_table[idx].cpu_id ==
   cpu_id) ) {
               env_memcpy(proc,&proc_table[idx], sizeof(struct hil_proc));
               return 0;
           }
       }
       return -1;
   }
```

The other option is to populate the required structures individually and copy them one by one to the hil_proc structure in the platform_get_processor_info function. The following code illustrates this scenario.

```
#define CPU_ID  1
#define NUM_CHANNELS 1

struct proc_shm ex_shm = {
```

```
            SHM_ADDR, SHM_SIZE, 0x00
        };

    struct proc_chnl ex_chnl = {
            "rpmsg-openamp-demo-channel"};
    };

    struct proc_intr ex_intr = {
            VRING1_IPI_VECT, IPI_PRIORITY, IPI_POLARITY
    };


    int platform_get_processor_info(struct hil_proc *proc , int cpu_id) {
        emv_memcpy(&proc-> sh_buff , &ex_shm , sizeof(struct proc_shm));
        emv_memcpy(&proc-> chnls, &ex_chnl , sizeof(struct proc_chnl));
        emv_memcpy(&proc-> vdev.vring_info[0].intr_info, &ex_intr ,
        sizeof(struct proc_intr));
        emv_memcpy(&proc-> vdev.vring_info[1].intr_info, &ex_intr ,
        sizeof(struct proc_intr));
             return 0;
    }
```

platform_get_processor_for_fw

This function returns the CPU ID for the given firmware name. The platform information is expected to provide the necessary firmware bindings to CPU ID. This implementation is required only when the OpenAMP Framework is used with master software contexts.

APIs to Implement to Provide Platform-Specific Functionality

The hil_platform_ops data structure defines specific functions that you are required to implement. The hil_platform_ops reference is saved in the hil_proc structure in the platform_get_processor_info call. Implementation of these functions should be provided in the platform.c file.

The following platform-specific functions need to be implemented.

enable_interrupt

This function enables APIs for virtio notification and registers the interrupt handler for them.

notify

This function triggers interrupts to let the other core know that there is data available for processing.

get_status

This function is for future use.

set_status

This function is for future use.

boot_cpu

Boot the remote CPU specified by CPU ID at the load address passed in as parameter.

shutdown_cpu

Shuts down the remote CPU specified by CPU ID.

Configuration Porting

The configuration porting component provides system level configuration abstractions such as obtaining firmware for the master and interrupts registery info.

Currently it requires users to implement only the config_get_firmware function for retrieving remote firmware. The function signature is present in the *<open_amp>/porting/config/config.h* file.

Environment Porting

The *env* directory contains the file *env.h*, which declares all the environment-specific APIs required by the OpenAMP Framework. The OpenAMP Framework reference implementation for Zynq ZC702EVK contains environmental API implementations/abstractions for simple bare metal execution environments. This reference implementation for Zynq should serve as a good starting point for enabling other new environments.

Table 6-4 presents the key environment APIs and a brief description their expected functionality/implementation.

Table 6-4. Environment Porting APIs

API	Expected functionality
env_init, env_deinit	Implements the OpenAMP Framework required, environment-specific initialization and deinitialization (for example, in the reference implementation). For bare metal environments: These APIs are stubs that do nothing.
env_allocate_memory, env_deallocate_memory	Implements environment-specific dynamic memory allocation and de-allocation primitives.
env_memset, env_memcpy, env_strlen, env_strcpy, env_strcmp, env_strncpy, env_strncmp, env_print	Implements env-layer mapping for the toolset and provides C library primitives used by the OpenAMP Framework. You can provide your own implementation of these APIs to enable the OpenAMP Framework to be used in embedded environments that do not have toolset-provided C libraries available.
env_map_vatopa, env_map_patova	Implements environment-provided primitives to convert physical address to virtual address and the other way around as well.
env_mb, env_rmb, env_wmb	Implements memory barriers using environment-provided primitives.
env_create_mutex, env_lock_mutex, env_unlock_mutex, env_delete_mutex	Implements protection mechanisms depending on the software environment. In the case of RTOS, this API can use RTOS provided mutex or binary semaphore primitives to provide env-layer abstractions for protecting access to shared resources. In the case of bare metal environments where threading capability is typically not present, this API can disable interrupts globally to protect access to shared resources.

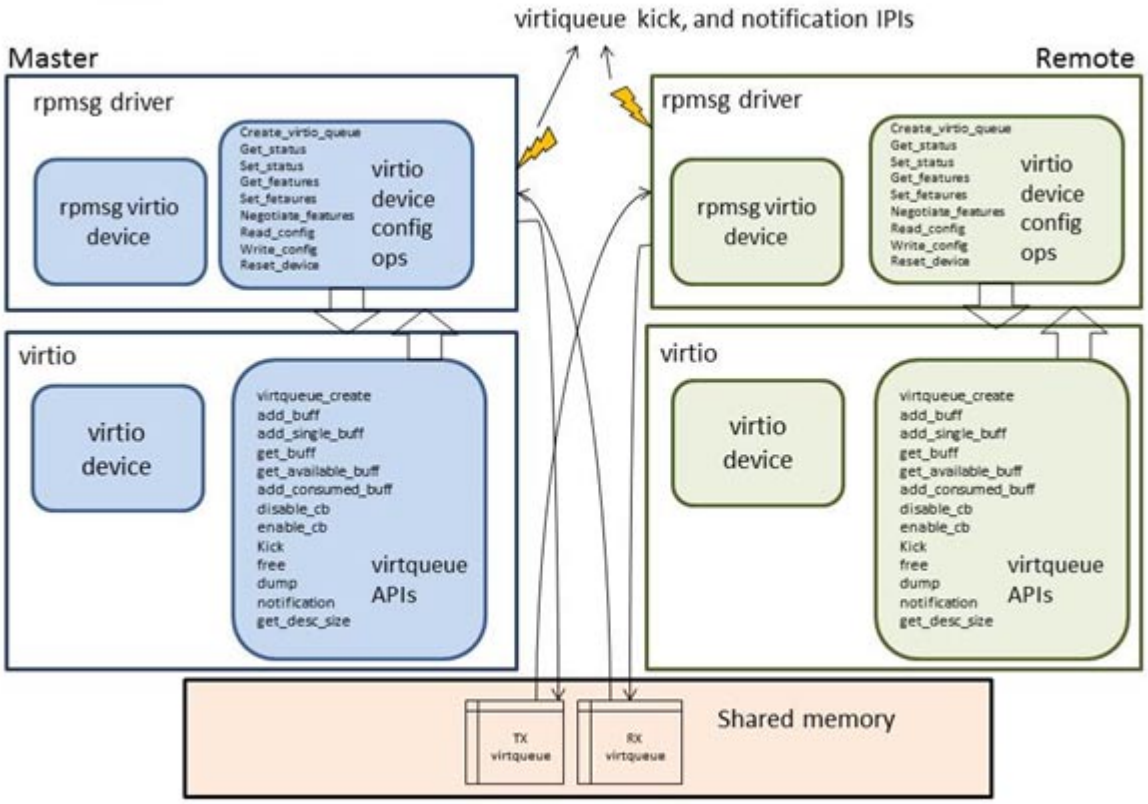
Table 6-4. Environment Porting APIs (cont.)

env_create_sync_lock, env_acquire_sync_lock, env_release_sync_lock, env_delete_sync_lock	Implements synchronization mechanisms depending on the software environment. In the case of RTOS, this API can use RTOS-provided blocking primitives like semaphores to enable synchronization. In the case of bare metal, this API can use atomic spinlocks to enable synchronization.
env_disable_interrupts, env_restore_interrupts	Implements global interrupt enablement and disablement abstractions using environment-provided primitives.
env_sleep_msec	Implements timed sleep abstraction using environment-provided primitives.
env_enable_interrupt, env_disable_interrupt, env_register_isr	Implements environmental abstraction to control processor interrupts on a per interrupt basis using environment-provided primitives.
env_map_memory	Implements environment abstraction to create a MMU tlb entry for a user-specified memory region using environment-provided primitives.

Appendix A
Virtio Concepts and RPMsg Usage

The virtio transport abstraction was originally developed for para-virtualization of Linux-based guests for lguest and KVM hypervisors. It serves as a standardized interface that lguest, KVM, and Mentor Embedded Hypervisors provide for IO virtualization of system resources for the guest operating systems like Linux and Nucleus RTOS. The Linux rpmsg bus driver leverages the virtio implementation in the Linux kernel to enable IPC for Linux in master and remote configurations.

Figure A-1. Virtio Concepts



The RPMsg framework's virtio implementation is adopted from the FreeBSD kernel with the addition of a couple of APIs. The rpmsg component uses virtio-provided interfaces to transmit and receive data with its counterpart. As a transport abstraction, virtio provides two key interfaces to upper level users (illustrated in Figure A-1):

- It provides a “virtio device” abstraction that allows a user driver to instantiate its own instance of a virtio device. It also allows for negotiation of the features and functionality supported by this user device (such as the rpmsg driver) by providing implementations of functions in virtio device config operations.
- It provides a “virtqueue” API that allows user drivers to transmit and receive data with the communicating counterpart using the virtqueue vring infrastructure.

The virtio implementation in the RPMsg framework (and in Linux) consists of the following:

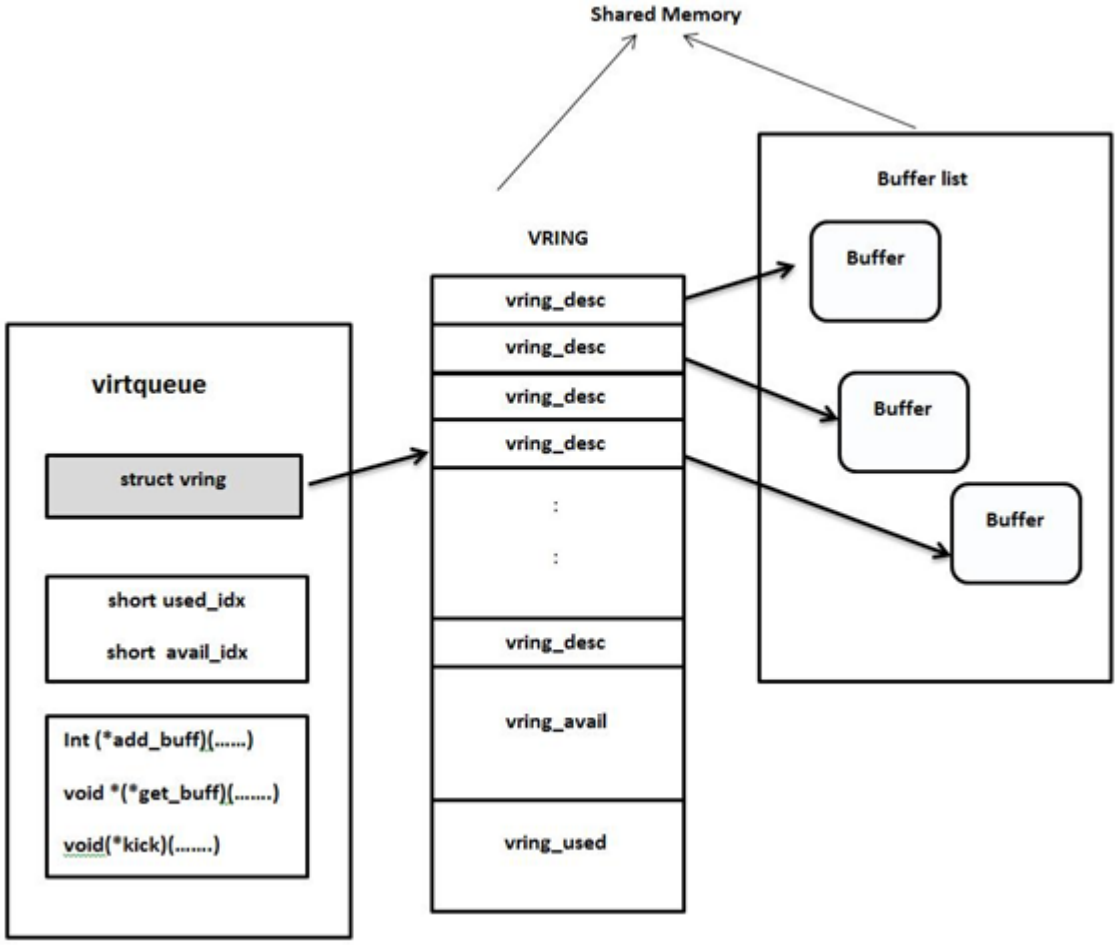
- A buffer management component called “VRING,” which is a ring data structure to manage buffer descriptors located in shared memory (Figure A-2)
- A notification mechanism to notify the communicating counterpart the availability of data to processed in the associated VRING.

Inter-Processor Interrupts (IPIs) are normally used for notifications. The virtqueue is a user abstraction that includes the VRING data structure with some supplemental fields, and APIs to allow user drivers to transmit and receive shared memory buffers. Each rpmsg channel contains two virtqueues associated with it: a tx virtqueue for master to uni-directionally transmit data to remote, and a rx virtqueue for remote to uni-directionally transmit data to master.

During the initialization of rpmsg, the following tasks are performed:

- The master context creates both the TX and RX virtqueues, and initializes the corresponding VRINGs with buffers from shared memory.
- A dedicated shared memory manager component within the RPMsg framework provides fixed-size buffers from a predefined shared memory space defined in HIL.
- The master transmits data to the remote by obtaining buffers referenced by descriptors in the TX virtqueue
- The master populates the virtque buffers with data, and notifies the corresponding remote using notification mechanisms defined in the HIL.

Figure A-2. The Virtqueue and Vring



The remote transmits data to master by obtaining buffers referenced by descriptors in RX virtqueue, populating them with data, and notifying the master using notification mechanisms defined in the HIL. On receiving data, virtio calls the rpmsg driver registered RX call back with reference to data received. The data is further processed by the RPMsg driver for delivery to application registered callbacks.

Third-Party Information

This software application may include zlib version 1.2.5 third-party software, which is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied.

This software application may include libfdt version 17 third-party software, which is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. libfdt version 17 may be subject to the following copyrights:

For the below copyright notice Mentor elects to distribute libfdt under the terms of the BSD license.

libfdt is dual licensed: you can use it either under the terms of the GPL, or the BSD license, at your option.

a) This library is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Alternatively,

b) Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Mentor Graphics BSD License, v1.0

To the extent an Open Source license does not otherwise apply to any component of the Software, the below BSD license shall apply.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of Mentor Graphics Corporation nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL MENTOR GRAPHICS CORPORATION BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.