Introduction to OpenAMP Library

An Open Source Standard and APIs for Asymmetric Multiprocessing (AMP) Systems

Authors: Etsam Anjum and Jeffrey Hancock

I. INTRODUCTION

There has been a dramatic increase in the number of embedded heterogeneous hardware platforms to address the diverse requirements of today's electronic devices. These requirements range from supporting the real-time behavior to the provisioning of rich User Interfaces (UIs) and thus need CPUs with different capabilities to optimally handle the task in hand. On the software side, this is accomplished by incorporating dissimilar software environments suitable to the capabilities of the CPUs present in the system. The different software contexts work in conjunction to provide the desired functionality. This collaboration usually entails communication enablement between the software context and the management of system resources. An AMP software architecture is required to develop such systems. The OpenAMP project provides software components to address a variety of AMP systems requirements. One of the important components of the OpenAMP project is the OpenAMP library which deals with the management of system resources and communication between the participating software contexts. The core building blocks of the library are Remoteproc and RPMsg. They provide resource management and communication features respectively.

II. HISTORY AND ORIGIN

The RPMsg and remoteproc infrastructure were originally conceived and committed to the Linux kernel by Texas Instruments (TI). The framework allowed the Linux Operating System (OS) on the master processor to manage the lifecycle and communicate with the remote software context on a remote processor. TI also provided the corresponding remote side implementation for their SYSBIOS operating system. OpenAMP took the course towards providing the standardized and portable implementation for the OS and BareMetal Environment (BME). This was enabled with the inclusion of well abstracted porting layers for the environment (OS/BME) and hardware. The first version was contributed to open source by Mentor Graphics in collaboration with Xilinx. The framework has seen continuous improvement since then and many new features were included, Linux userspace support being the most notable. Other important features support for proxy devices and Linux operation in remote mode. Another distinctive aspect is the ability to run the RPMsg between two Linux processes in the native Linux environment. The RPMsg API is compatible with the upstream Linux.

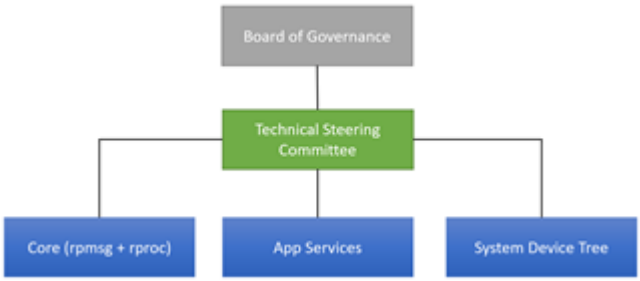
The RPMsg and remoteproc components are managed under OpenAMP library project. The project is actively maintained by the open-source community and it is present on the GitHub at https://github.com/OpenAMP .

The OpenAMP project has evolved continuously and it is not confined to just RPMsg and remoteproc. The project also looks into system resource partitioning problems and development of high level services on top of RPMSG and remoteproc. These topics are covered in different working groups as explained in the next section.

To better address the logistical needs of the project and formalize collaboration, the OpenAMP project was moved to Linaro community projects in September 2019. The details of the project can be found on the project website at https://www.openampproject.org/about/.

III. OPENAMP LINARO COMMUNITY PROJECT

The OpenAMP project is a Linaro Community Project. The organizational structure of the project comprises a Board of governance, Technical Steering Committee (TSC) and Development working groups. The project board of governance oversees strategic & financial decisions, TSC charter, trade shows and face-to-face meetings and other logistics. The TSC deals with the technical matters of projects such as code review process, maintainer selection, coding conventions etc. The technical development is split across three working groups: 1) OpenAMP Core 2) Application services and 3) System Device Tree. OpenAMP core group is responsible for development of RPMsg and remoteproc and it covers the OpenAMP library. The Application services group looks into high level services development on top of the RPMsg such as proxy interface. The System Device Tree group is standardizing the Device Tree bindings for heterogeneous platforms, including the multiple heterogeneous processors and their bus views.



The mailing lists for working groups are available at https://lists.openampproject.org/mailman/listinfo. You can subscribe to the mailing list to get the up to date information.

Please note that the upstream Linux maintains RPMsg and remoteproc kernel drivers which are not discussed in this paper. This paper focuses on technical details of the Open AMP library with occasional references to Linux drivers to clarify the working.

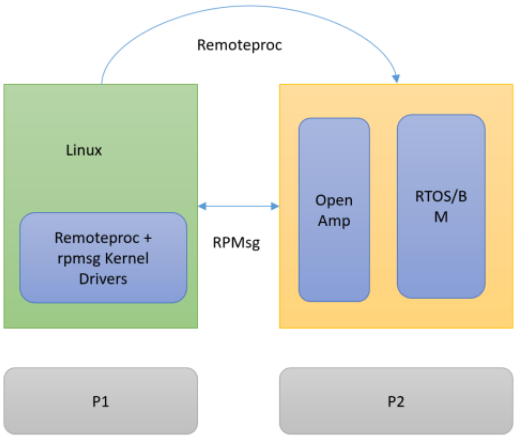
The term OpenAMP is used to refer to the OpenAMP library in this paper from here onwards.

IV. USE CASES AND APPLICATIONS

OpenAMP can be used in different hardware platforms containing homogeneous and heterogeneous CPUs, with a variety of OS combinations in supervised and unsupervised AMP environments.

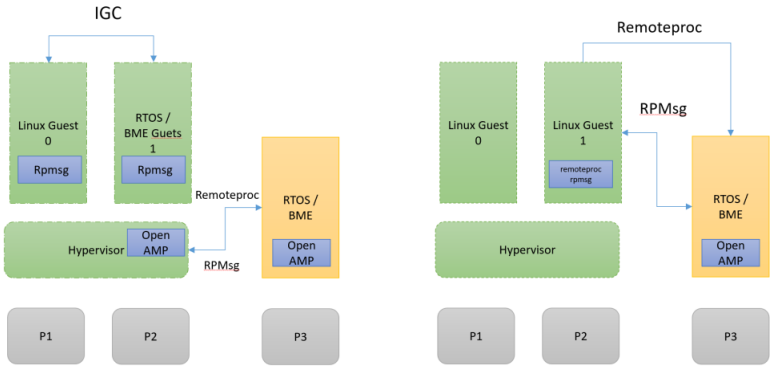
The unsupervised AMP software architecture is desirable in scenarios where the number of software contexts are less or equal to the number of cores and these contexts are expected to be used mostly unmodified but at the same time take advantage of the multiple cores. In these systems, the software runs natively on the processors and therefore does not incur any additional performance overhead. The hardware provided resources are usually leveraged to enforce isolation between the software contexts. For example, on the Xilinx Zynq UltraScale+ MPSoC platform (ZynqMP), the Xilinx Memory Protection Unit (XMPU), and the Xilinx Peripheral Protection Unit (XPPU) are used for this purpose. On the TI OMAP SoC, L3 firewall serves the same purpose. OpenAMP is effectively utilized in the platforms having only homogenous CPUs such as NXP i.MX6 Sabrelite and Xilinx Zynq-7000 as well as platforms having a combination of different CPUs such as Xilinx ZynqMP, NXP i.MX8QXP etc. However, for the latter platforms, some additional drivers are required to route the notification across the cores, such as, IPI block driver for ZynqMP and Messaging Unit (MU) driver for i.MX8. For Unsupervised AMP systems, OpenAMP allows master software to manage lifecycle and offload computation to other CPUs present in the system.

Different combinations of a General Purpose OS (GPOS) and RTOS/BareMetal(BM) can be supported in a master and remote role. For instance, the master context could be a Linux with the kernel space RPMsg & remoteproc support, Linux with OpenAMP in user space or an RTOS/BM with OpenAMP. Similarly a remote could be BM software, or RTOS and even Linux.



The supervised AMP architecture is best suited for the application where isolation of software instances and virtualization of hardware resources are required. In such systems, the hypervisor manages the guest software context

and provides the system isolation and virtualization services to it. OpenAMP can be used to manage the heterogeneous cores from the hypervisor and its guests. For platforms having only the homogenous cores and a hypervisor running on all its cores, the inter-guest communication can be enabled using the RPMsg.



The framework provides a convenient way to migrate the legacy single core application to multi-core SoCs without excessive modification while at the same time leveraging the capabilities of new hardware platforms.

The framework facilitates the implementation of the fault tolerant system. For example, the framework can enable an RTOS on the safety critical processor, which is the system master and manage the critical system operations, to control the lifecycle of the GPOS (such as Linux) on the application processor. If the GPOS has a failure, the RTOS can simply reboot the GPOS system without impacting the operation of the rest of the system..

V. LIFE CYCLE MANAGEMENT USING REMOTEPROC

The life cycle management capability of OpenAMP is provided by Remoteproc. OpenAMP adheres to master-slave (remote) topology wherein master software controls the life cycle of the remote software and establishes the communication channel.

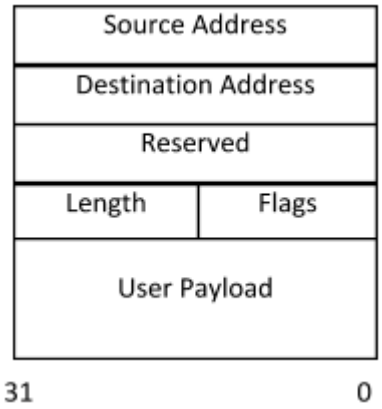
Remoteproc loads the software image for the remote processor. An ELF file of the remote image is required for this operation. Remote lists the required resources in a data structure known as resource table. These resources include memory for the remote image, shared memory for data IO and trace buffers for remote software. An important resource in the resource table is the VirtIO device resource which contains fields for VirtIO ID, device & host features, status, notification ID, number of vrings etc. The status field is used by the master at runtime to share the VirtIO device initialization progress. The VirtIO device resource also encapsulates the information about the vring, which is data structure in the shared memory used for data IO. The resource table is embedded in the special section of the remote ELF file. The resource table is initially populated by the remote requesting the resource from the master. The master software parses the ELF file, obtains the resource table section from it, carves out the resources, loads the program segment into the memory and starts the remote core from the entry point. The mechanics in providing a remote image to the master are not defined by OpenAMP. The remote image can be obtained from the file system (such as when using a Linux master) or it can be embedded into the master image using the inc-bin assembly directive.

The Linux documentation defines a standard set of resources for the resources table. It is also allowed to define the custom/vendor resources with resource handling provided in the hardware specific code.

OpenAMP allows remoteproc usage without the resource table. This is useful when no subsequent communication is required between the master and the remote. Moreover, remoteproc can parse the segments of ELF files without requiring access to complete the ELF file. This provides a convenient way to save memory in a memory constrained systems where it is not possible to bring complete remote image in the main memory.

VI. INTER-PROCESSOR COMMUNICATION USING RPMSG

The communication between the software contexts is provided by the RPMsg component of OpenAMP. RPMsg defines the format of the messages exchanged between the master and the remote and notification sequence. The RPMsg header is attached to each message and it identifies the source, destination endpoints, and size of the payload. The RPMsg header is depicted in the figure below. The Flags field is currently unused.



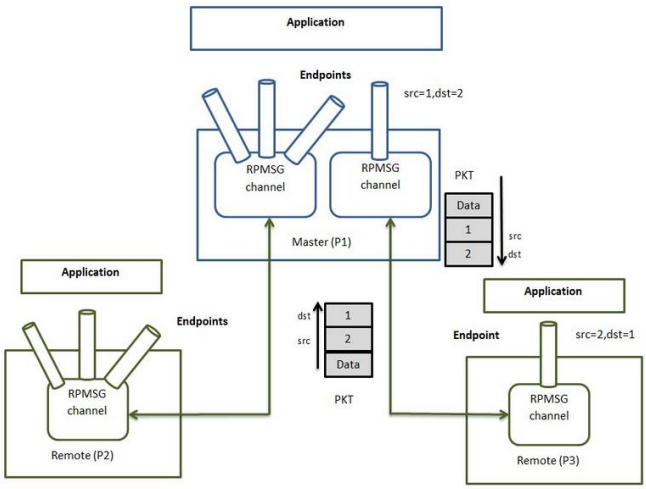
A. Channel

An RPMsg channel is a bidirectional communication link between the master and the remote. The RPMsg channel is identified by the textual name, source and destination address. The RPMsg framework keeps track of the channels using their names. The RPMsg channel is usually dynamically created and the remote advertises its presence to the master by sending the Name Service (NS) announcement containing the name of the channel. It is also possible to have a static channel where both sides know the name of the channel in advance.

B. Endpoint

The RPMsg endpoints provide logical connections on top of the RPMsg channel. Every endpoint has a unique source and destination address. The source address is the local address of the endpoint, i.e. it is the address of the endpoint in the context where it was created. The destination address is the address of the endpoint on the remote context. Each RPMsg Packet (PKT) carries the source and destination endpoint address. The RPMsg endpoint has an associated call back function which is triggered when the data is received on the endpoint. The application consumes that data in the call back.

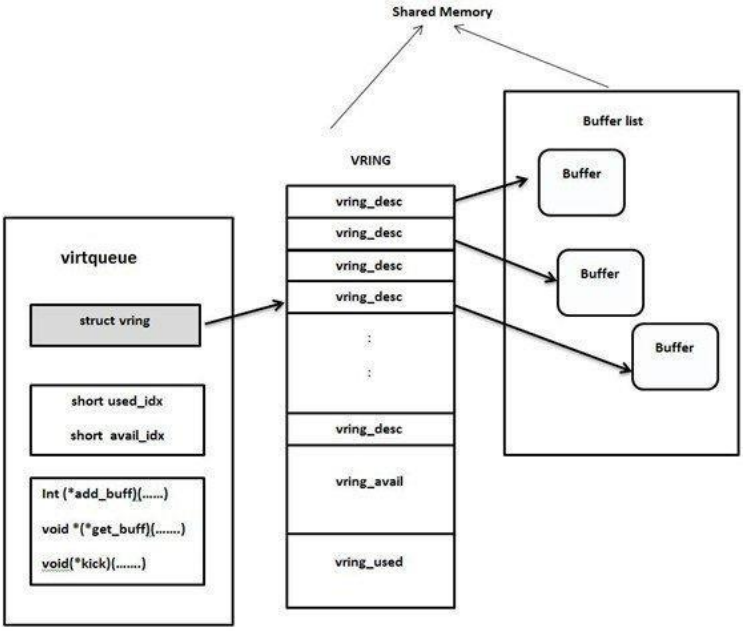
The next figure depicts the concept of channel and endpoint in the RPMsg framework.



C. VirtIO

RPMsg uses VirtIO as a shared memory based transport abstraction. VirtIO has its roots in the lguest hypervisor. It is used as a standard IO virtualization mechanism, providing virtual device configuration and data exchange between the guest driver (front end) and the virtual device in the hypervisor (backend). VirtIO defines a representation of virtual devices in the shared memory and also defines APIs to read/write to device memory. In OpenAMP, the resource table mimics device space in virtualized environment. VirtIO also defines a communication abstraction known as virtqueues. This is used to transfer data between guest and host. RPMsg uses the same abstraction to exchange data between the master and the remote. Internally, virtqueues maintain a ring buffer, known as vring. The vring resides in the shared memory and contains ring of buffer descriptors. The buffer descriptors contain the pointers to buffers

exchange between the master and the remote and read/write permissions etc. The VirtIO data structures are depicted in the figure below.



VII. OpenAMP Code Organization

OpenAMP code is organized into three components:

D. OpenAMP (Library)

OpenAMP library contains code for remoteproc and RPMsg components. The user-facing APIs are part of the OpenAMP library. The OpenAMP library is OS and hardware agnostic and therefore does not contain porting layers for either, OS or hardware.

E. Libmetal

Libmetal provides the interfaces for OS services needed by OpenAMP. This includes: locking primitives, interrupts, cache maintenance operations, DMA memory, time, sleep etc. The portable components are organized in separate files and are needed to be implemented for each OS. Libmetal also provides OS independent utilities such as logger, linked list, bitmap operations etc.

Libmetal maintains the concept of bus and device abstraction which enables OpenAMP usage from the Linux user space. Linux User Space IO (UIO) system is used for memory space access and notifications.

The libmetal code is maintained in a separate repository in the OpenAMP GitHub project.

F. Application

The applications contain the demo code to showcase the usage of the OpenAMP APIs. The hardware porting layer is maintained along with the applications and provides the implementation of the remoteproc operations such as CPU start/stop, memory mappings and notification generation. Applications are part of the OpenAMP library.

VIII. OpenAMP APIs

OpenAMP provides a comprehensive set of APIs to manage remote processor and establish communication with the software running on a remote processor. This section provides an overview of the important OpenAMP APIs.

G. Remoteproc APIs

The important remoteproc APIs are tabulated below:

Index	API	Description
1	struct remoteproc *remoteproc_init(struct remoteproc *rproc, struct remoteproc_ops *ops, void *priv);	Master & Remote API -Creates and initializes remoteproc instance.
2	int remoteproc_remove(struct remoteproc *rproc);	Master & Remote API -Deletes remoteproc instance
3	int remoteproc_config(struct remoteproc *rproc, void *data);	Master API -This function initializes the remoteproc resources required for loading and executing the image.
4	int remoteproc_start(struct remoteproc *rproc);	Master API -This function starts the remote processor from the image entry point which is already present in the memory.
5	int remoteproc_stop(struct remoteproc *rproc);	Master API -This function stops the remote processor but its resources are not released
6	int remoteproc_load(struct remoteproc *rproc, const char *path, void *store, struct image_store_ops *store_ops, void **img_info);	Master API -This function fetches ELF segments using the remoteproc image store operations and load them to memory.
7	int remoteproc_shutdown(struct remoteproc *rproc);	Master API -This function shutdowns the remote processor and free its resources.
8	Int remoteproc_create_virtio(struct remoteproc *rproc, int vdev_id, unsigned int role, void (*rst_cb)(struct virtio_device *vdev))	Master & Remote API -Creates virtio device instance.

H. RPMsg APIs

The RPMsg communication APIs are listed in the table below.

Index	API	Description
1	int rpmsg_send(struct rpmsgg_endpoint *ept, const void *data, int len)	This function sends data to remote endpoint. In case there are no TX buffers available, the function will block for 15 seconds.
2	static inline int rpmsg_sendto(struct rpmsg_endpoint *ept, const void *data, int len, uint32_t dst)	This function sends data to endpoint address given in the parameter. In case there are no TX buffers available, the function will block for 15 seconds.
3	static inline int rpmsg_send_offchannel_raw(struct rpmsg_endpoint *ept, uint32_t src, uint32_t dst, const void *data, int len)	This function sends data to remote endpoint, with the address given in the parameter. The src address is also provided in the arguments. In case there are no TX buffers available, the function will block for 15 seconds.
4	static inline int rpmsg_trysend(struct rpmsg_endpoint *ept, const void *data, int len)	This function sends data to remote endpoint. In case there are no TX buffers available, the function will immediately return error.
5	static inline int rpmsg_trysendto(struct rpmsg_endpoint *ept, const void *data, int len, uint32_t dst)	This function sends data to endpoint address given in the parameter In case there are no TX buffers available, the function will immediately return error.
6	static inline unsigned int is_rpmsg_ept_ready(struct rpmsg_endpoint *ept)	Check if the rpmsg endpoint is ready to send data.
7	int rpmsg_create_ept(struct rpmsg_endpoint *ept, struct rpmsg_device *rdev, const char *name, uint32_t src, uint32_t dest, rpmsg_ept_cb cb, rpmsg_ns_unbind_cb ns_unbind_cb);	Creates rpmsg endpoint and initializes it with given parameters.
8	void rpmsg_destroy_ept(struct rpmsg_endpoint *ept);	It destroys the rpmsg endpoint and triggers the destroy endpoint callback if it is provided

I. Pseudo Code

The pseudo code illustrates the usage of the key remoteproc and RPMsg APIs from master and remote contexts.

Master

```c
/* Example rmoteproc ops table - implementation is provided by the
application */
struct remoteproc_ops rproc_ops = {
      .init = rproc_init,
      .remove = rproc_remove,
      .start = rproc_start,
      .stop = rproc_stop,
      .shutdown = rproc_shutdown,
      .mmap = rproc_mmap,
};

/* Example image store ops to fetch the remote image content from the memory
and load
 * them to destination memory - implementation is provided by the
application.
 */
struct image_store_ops mem_image_store_ops = {
      .open = mem_image_open,
      .close = mem_image_close,
      .load = mem_image_load,
      .features = SUPPORT_SEEK,
};

/* Endpoint data receive call back function */
static int rpmsg_endpoint_cb(struct rpmsg_endpoint *ept, void *data, size_t
len,
            uint32_t src, void *priv) {
      return RPMSG_SUCCESS;
}

/* Name service announcement call back function for channel creation */
static void rpmsg_name_service_bind_cb(struct rpmsg_device *rdev,
            const char *name, uint32_t dest) {

}

/* Name service announcement call back function for channel deletion */
static void rpmsg_service_unbind(struct rpmsg_endpoint *ept) {
}

int main(void) {

      struct  rpmsg_device * rpmsg_dev;
      struct remoteproc rproc;
      void *store = RMT_IMAGE_MEM_ADDR;
      struct rpmsg_virtio_device *rpmsg_vdev;
      struct rpmsg_endpoint ept;
      struct virtio_device vdev;
      struct metal_io_region *shbuf_io;
      void *shbuf;
      metal_phys_addr_t pa;

      /* Initialize remoteproc instance */
      remoteproc_init(&rproc, rproc_ops, NULL);
```

```
	/* mmap shared memory */
	pa = SHARED_MEM_PA;
	(void *)remoteproc_mmap(&rproc , &pa,
	                        NULL, SHARED_MEM_SIZE,
	                        NORM_NSHARED_NCACHE|PRIV_RW_USER_RW,
	                        &shbuf_io);
	 /* Configure remoteproc to get ready to load executable */
	remoteproc_config(&rproc, NULL);

	/* Load the image */
	remoteproc_load(&rproc, NULL, RMT_IMAGE_MEM_ADDR, &mem_image_store_ops,
	                NULL);

	/* Start the remote processor */
	ret = remoteproc_start(&rproc);

	/* Setup the communication mechanism */
	vdev = remoteproc_create_virtio(&rproc, 0, VIRTIO_DEV_MASTER, NULL);

	/* Only rpmsg virtio master needs to initialize the
	 * shared buffers pool
	 */
	shbuf = metal_io_phys_to_virt(shbuf_io, SHARED_MEM_PA);

	rpmsg_virtio_init_shm_pool(&shpool, shbuf,
	                (SHARED_MEM_SIZE - SHARED_BUF_OFFSET));

	/* Initialize the underlying virtio device */
	ret = rpmsg_init_vdev(rpmsg_vdev, vdev, rpmsg_name_service_bind_cb,
	                shbuf_io, shpool);
	/* Get the rpmsg device */
	rpmsg_dev = rpmsg_virtio_get_rpmsg_device(rpmsg_vdev);

	:
	:
	/* Wait for the name service announcement before endpoint creation  */
	wait();

	/* NS announcement is received, create the endpoint*/
	(void) rpmsg_create_ept(&ept, rdev, RPMSG_SERV_NAME, RPMSG_ADDR_ANY,
dest,
	                rpmsg_endpoint_cb, rpmsg_service_unbind);

	/* Endpoint is created - send data using the rpmsg comm APIs */
	rpmsg_send(&ept, HELLO_MSG, strlen(HELLO_MSG));

	:
	:

	remoteproc_stop(rproc);
	remoteproc_shutdown(rproc);
}
```

Remote

```c
int main(void)
{
	void *rsc_table;
	int rsc_size;
	int ret;
	metal_phys_addr_t pa;
	struct rpmsg_device * rpmsg_dev;
	struct rpmsg_endpoint ept;
	struct remoteproc rproc;
	struct virtio_device vdev;
	struct metal_io_region *shbuf_io;
	void *shbuf;

	/* Get the resource table */
	rsc_table = get_resource_table(rsc_index, &rsc_size);

	/* Initialize remoteproc instance */
	remoteproc_init(&rproc, rproc_ops, NULL);

	/* mmap resource table */
	pa = (metal_phys_addr_t)rsc_table;
	(void *)remoteproc_mmap(ret_rproc , &pa,
	                        NULL, rsc_size,
	                        NORM_NSHARED_NCACHE|PRIV_RW_USER_RW,
	                        &ret_rproc.rsc_io);

	/* mmap shared memory */
	pa = SHARED_MEM_PA;
	(void *)remoteproc_mmap(&rproc , &pa,
	                        NULL, SHARED_MEM_SIZE,
	                        NORM_NSHARED_NCACHE|PRIV_RW_USER_RW,
	                        &shbuf_io);

	/* Pass resource table to remoteproc for parsing */
	remoteproc_set_rsc_table(&rproc , rsc_table, rsc_size);

	/* Setup the virtio device for communication */
	vdev = remoteproc_create_virtio(&rproc , 0, VIRTIO_DEV_SLAVE, NULL);


	/* Initialize the underlying virtio device */
	ret = rpmsg_init_vdev(rpmsg_vdev, vdev, rpmsg_name_service_bind_cb,
	                      shbuf_io, NULL);
/* Get the rpmsg device */
	rpmsg_dev = rpmsg_virtio_get_rpmsg_device(rpmsg_vdev);

	/* Create the endpoint */
	(void) rpmsg_create_ept(&ept, rdev, RPMSG_SERV_NAME, RPMSG_ADDR_ANY,
RPMSG_ADDR_ANY,
	                        rpmsg_endpoint_cb, rpmsg_service_unbind);

	/* Wait for  the endpoint to get ready */

	while(!is_rpmsg_ept_ready(&ept));

	/* Endpoint is created - send data using the rpmsg comm APIs */
```

```
        rpmsg_send(&ept, HELLO_MSG, strlen(HELLO_MSG));

        :
        :
}
```

IX. OpenAMP Key Features

OpenAMP offers a rich set of features to facilitate the development of an AMP system. Some important features, other than the basic ones are discussed in this section.

1. Independent RPMsg and Remoteproc Operation - The v2018.10 release of OpenAMP, largely restructured the code base and remoteproc & RPMsg were decoupled to minimize the memory footprint.
2. Proxy support for serial console and file system. This feature enables the software context on the slave processor to use the console and file system support on the master software. Internally, RPMsg is used to route the console and file system Run Time Library (RTL) calls to the master software. In a typical scenario, master software is a high level OS running on high end CPUs and slave software is RTOS/BME on low end CPU. The slave software leverages the features provided by the master software using RPMsg.
3. Linux User Space Support - OpenAMP can be used from the Linux userspace. The libmetal contains the porting layer for Linux. Internally, the UIO drivers are used for interrupts and device MMIO space access.
4. Native Linux Execution - OpenAMP can be compiled for the native Linux environment, thanks to cmake. The RPMsg master runs in one Linux process and the RPMsg remote in the other. They can then communicate over RPMsg which internally uses the Linux sockets interface. This feature enables a development environment without any additional hardware support, it is useful for testing the generic RPMsg features.
5. OS Support - OpenAMP supports FreeRTOS, Linux, Zephyr and NuttX.
6. Platform Support – Currently, Xilinx Zynq UltraScale + MPSoC and Zynq-7000 platforms are supported.
7. Some vendors, such as Mentor, A Siemens business, provide out-of-tree OpenAMP support with their RTOS Nucleus on multiple platforms.

X. Upcoming Features

The Open-Source community is actively working on improvements and feature enhancements for the OpenAMP project. The future topics for OpenAMP core group are captured at https://github.com/OpenAMP/open-amp/wiki/Future-Topics-for-OpenAMP-remoteproc-sub-group. Some of the important topics are elaborated below.

1. **Large Buffer Support** - RPMsg currently supports fixed size shared buffers. If an application buffer size is greater than the size of the shared buffer than an error is returned. To transfer large data, an application must first split it into small chunks. This incurs some performance overhead, especially due to notifications required for each split transfer. RPMsg can be augmented to support large buffers by leveraging the buffer chaining feature of the VirtIO vring descriptors. The `next` field of the vring descriptors can be used to link the buffers belonging to the same transfer. So if a user passes the large buffer it can be copied into different shared buffers which can be chained together in vring and transferred under a single notification.

2. **System Partitioning** - The idea of system partitioning is to distribute the available resources such as peripherals, memories and interrupts to the participating software environments. This happens at build time to consequently enable cooperative usage of system resources. One approach is to augment the Device Tree Source (DTS) with the complete system definition including the slave processors and their bus views. Subsequently, develop a resource assignment description file that would assign the resources from the System Device Tree to machines (software) listed in the description file. The description file can conform to device tree syntax and define additional properties where needed, to effectively distribute the resources. Host tools will have to be developed to take the System DTS and the resource assignment description to generate the device tree for each of the software environments. This will avoid the need for manually modifying the platform definition available to participating contexts for available system resources. However, DTS may not be supported for all the OSs/BME and will, therefore, need enhancements to leverage this feature. Another area of concern would be to see how the shared resources, such as interrupt controllers, will be partitioned. These resources are expected to be initialized by the master context in the system and are later used by both the master and remote contexts. To address this problem, the tool can generate the run time artifact such as,

driver code for shared peripherals that will enable the conflict-free usage of these resources. The System Device Tree working group is looking into this feature.

3. **System Resource Management** - Hardware resources can be categorized into two types: non-shareable - resources which are exclusively assigned to one software and includes peripherals such as UART, I2C etc. and System/Shared resources such as clocks, GPIO, voltage regulators and resets, which are used by the multiple applications. The mutually exclusive resources may, in turn, need system resources to operate such as GPIO and clocks. The resource manager can be dynamic or static. TI pushed the dynamic System Resource Manager (SRM) where the remote dynamically requests the resources after boot up using the RPMsg. In the static approach, the remote requests these resources at the boot time and the assignment does not change afterwards. The former can support the dynamic power management and later is more geared towards the conflict free usage of resources. The static resource assignment will provide centralized access to shared resources and hence prevent conflicting configurations.

XI. Getting Started with the OpenAMP Project

The best place to start with the OpenAMP project is the landing page on the GitHub at https://github.com/OpenAMP/open-amp. The readme provides the instructions to use OpenAMP with the supported OS and hardware platforms.