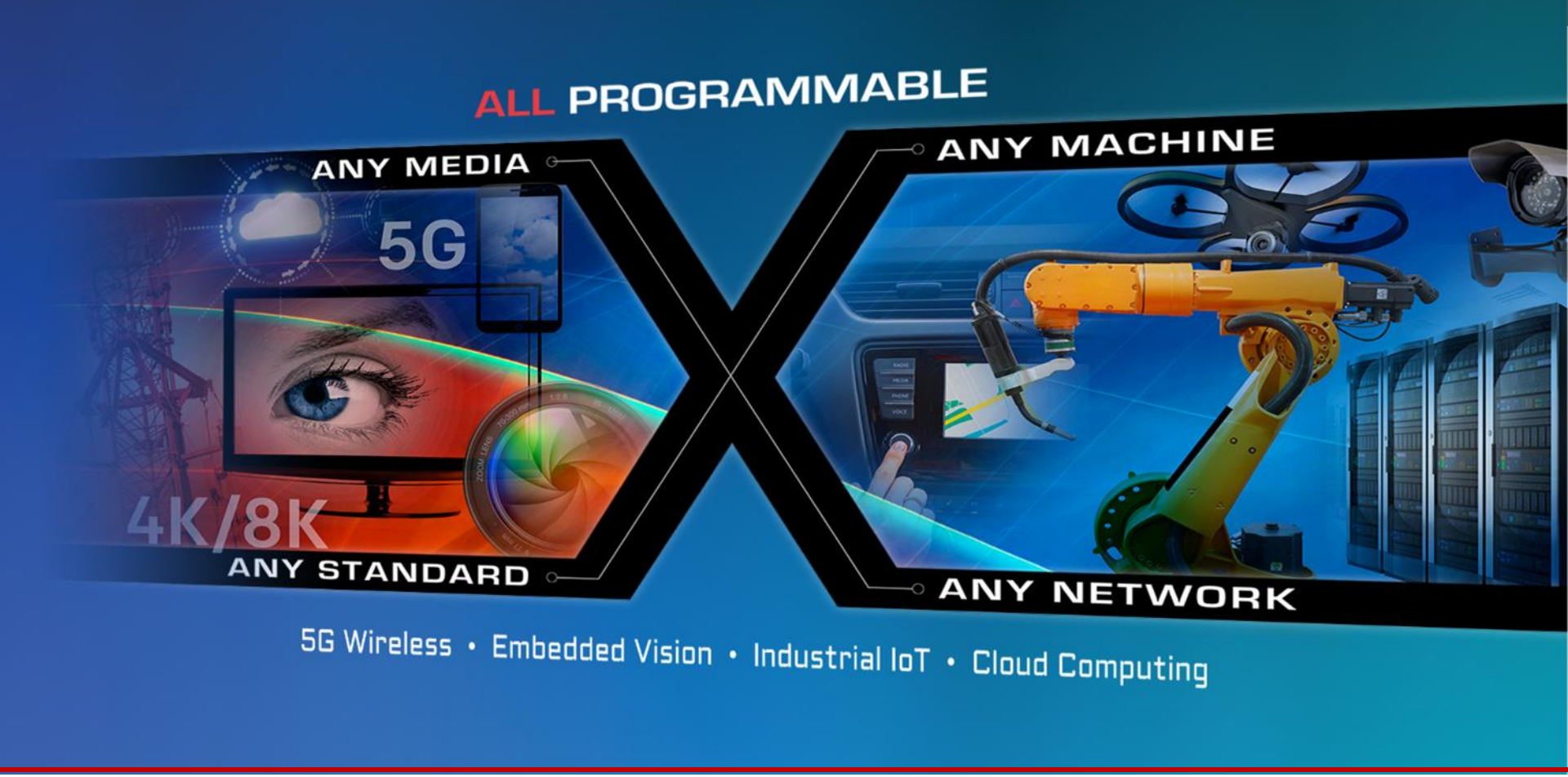


XILINX
ALL PROGRAMMABLE™

OpenAMP Discussion - Linaro2018HK

XILINX
ALL PROGRAMMABLE™

Agenda

- SPDX Short Licenses Identifier
- Coding Guideline
- API Standardisation
- Coprocessor Image Format
- OpenAMP and Container

OpenAMP License

SPDX Short Licenses Identifier

- Why use SPDX licenses list short license identifiers in the open source implementation?
  - Easy to use, machine-readable
  - Concise and now standardized format
  - https://spdx.org/sites/cpstandard/files/pages/files/using_spdx_license_list_short_identifiers.pdf
- E.g.
  - Use this in the *.c and *.h files in the source codes in the OpenAMP open source implementation:
    - /* SPDX-License-Identifier: BSD-3-Clause */

Coding Guideline

Coding Guideline

- Linux coding style
- Safety Certification Consideration
  - MISRA C, coding guidelines for style and naming conventions
  - Unit testing
    - exercising parameters boundaries, interfaces
    - code coverage
      - 100% branch coverage

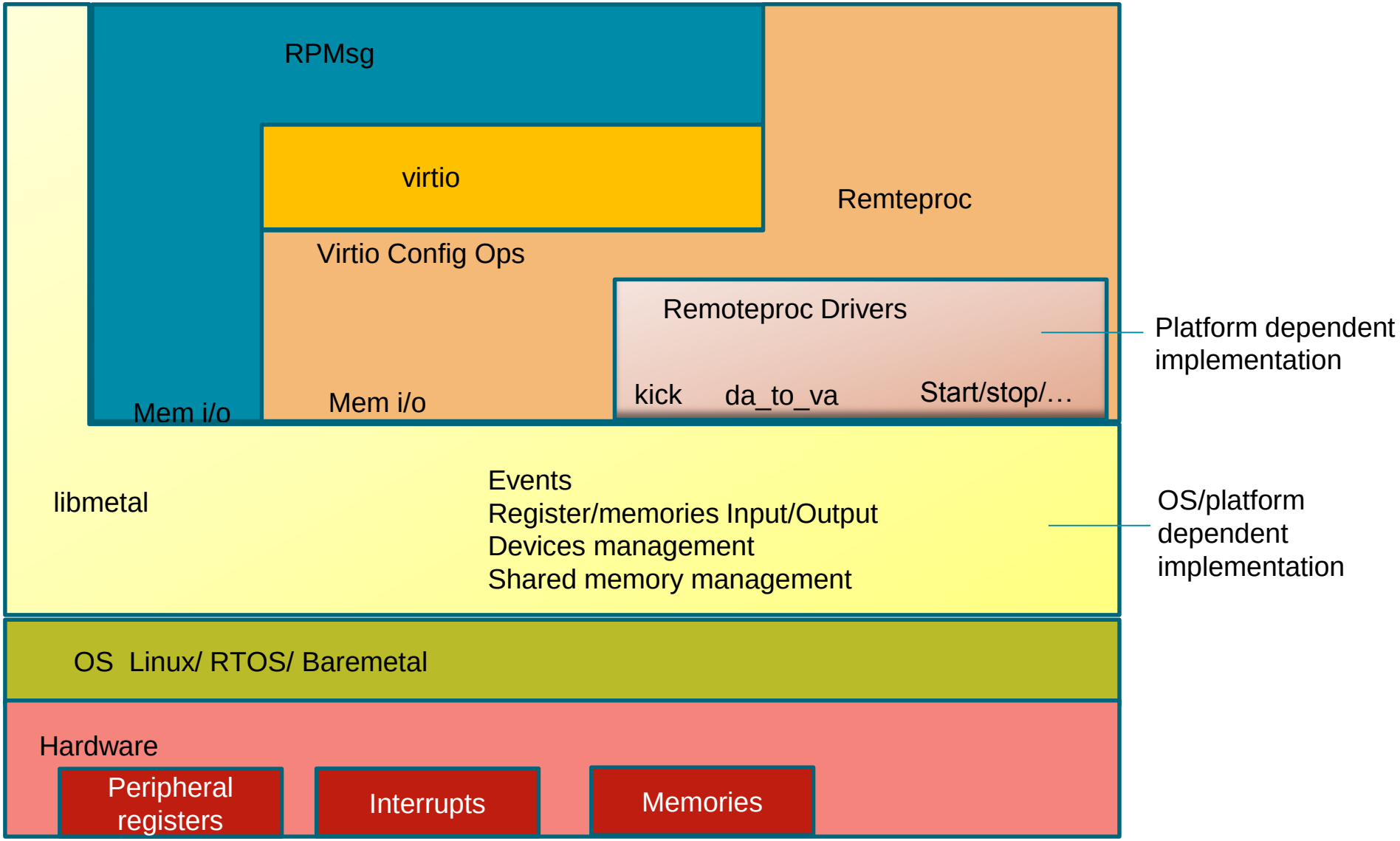
API Standardisation

APIs Standardisation

With standardised flow and APIs, it allows a developer to run the same application in different environment, e.g. Linux kernel, Linux userspace, baremetal and RTOS.

- RPMsg
  - Standardised APIs
  - One common implementation
- Virtio config
  - Standardised APIs
  - One common implementation
  - Supports more than RPMsg virtio driver in future
- Remoteproc
  - Standardised APIs
  - One common implementation
  - Rempteproc drivers are platform dependent
- Libmetal
  - Standardised APIs
  - OS/platform dependent implementation

Libraries Layers



Coprocessor Image Format

Coprocessor Images

In order to launch a Image on a coprocessor, the image includes the following:

- Binary
- Resources required to start the application
- Authentication
- Encryption

There is use case that the once started getting the image, the image data will come in streams, and the master's memory is very limited, authentication needs to be done before decrypt the image data.

- Today we have our own image format to solve this issue. How about standardise it in OpenAMP?

Image Format	Authentication	Encryption	Streaming
ELF	No out-of-box solution	No out-of-box solution	No
FIT	yes	No out-of-box solution	No
PE (Portable Executable)	yes	No out-of-box solution	No
OCI Image Specification	yes	No	No
Tar ball (vendor specific format)	Yes	Yes	Yes

OpenAMP and Container

Container and OpenAMP

A container is a self contained execution environment that shares the kernel of the host system and which is (optionally) isolated from other containers in the system.

Open Container Initiative(OCI) runtime specification defines container life cycles. Open source container development platform such as docker provides solution to distribute and orchestrate containers.

Containerisation is a very popular topics, can OpenAMP leverage container solutions?

- Deploy a application running on a coprocessor in a container?
- Deploy a application running on a coprocessor as deploying a container?
  - User should be able to deploy a coprocessor application in the same way as they deploy a application on VM.
- Can we make use of solutions for data sharing/synchronisation between containers, or between host and containers?
- Can we make use of container scheduling solution to schedule application on coprocessors?
  - e.g. Xilinx Zynq UltraScale+ MPSoC, Linux running on A53 has two Cortex-R5 coprocessors
- How about a OpenAMP runtime?

API Standardisation Backup Slides

Rpmsg APIs

Rpmsg APIs for user to transmit data with rpmsg(Remote Processor Messaging framework.

- Based on Linux kernel implementation with additions

Rpmsg APIs (1 / 2)

API	Description
int rpmsg_send(struct rpmsg_endpoint *ept, void *data, int len);	Send data to the target with RPMsg endpoint
int rpmsg_sendto(struct rpmsg_endpoint *ept, void *data, int len, uint32_t dst);	Send data to the specified RPMsg endpoint address
int rpmsg_send_offchannel(struct rpmsg_endpoint *ept, void *data, int len, uint32_t src, uint32_t dst);	Send data to the specified RPMsg endpoint address using the specified source endpoint address
int rpmsg_trysend((struct rpmsg_endpoint *ept, void *data, int len);	Send data to the target with RPMsg endpoint, if the endpoint doesn't have available buffer, it will return instead of blocking until there is buffer available.
int rpmsg_trysendto(struct rpmsg_endpoint *ept, void *data, int len, uint32_t dst);	Send data to the specified RPMsg endpoint address, it will return instead of blocking until there is buffer available.
int rpmsg_trysend_offchannel(struct rpmsg_endpoint *ept, void *data, int len, uint32_t src, uint32_t dst);	Send data to the specified RPMsg endpoint address using the specified source endpoint address, it will return instead of blocking until there is buffer available.
int (*rpmsg_callback)(struct rpmsg_endpoint *ept, void *data, int len, uint32_t src, void *priv);	User defined RPMSg callback
void (*rpmsg_endpoint_destroy_callback)(struct rpmsg_endpoint *ept, void *priv);	User defined RPMsg endpoint destroy callback

Rpmsg APIs (2 / 2)

API	Description
struct rpmsg_endpoint *rpmsg_create_ept(struct rpmsg_virtio *rpmsgv, const char *ept_name, uint32_t src, uint32_t dest, rpmsg_callback cb, rpmsg_destroy_callback destroy_cb, void *priv);	Create a RPMsg endpoint from virtio device. API is different to the Linux kernel implementation
void rpmsg_destroy_endpoint(struct rpmsg_endpoint *ept);	Destroy RPMsg endpoint API is different to the Linux kernel implementation
struct rpmst_virtio *rpmsg_vdev_init(struct virtio_dev *vdev, void *shm, int len);	Initialize RPMsg virtio queues and shared buffers, the address of shm can be ANY. In this case, function will get shared memory from system shared memory pools. If the vdev has RPMsg name service feature, this API will create an name service endpoint. On the slave side, this API will not return until the driver ready is set by the master side. Not in Linux kernel

Virtio Config Ops APIs

Virtio config operations APIs are used to configure the virtio device (1.0 Specification)

- For every configuration change, it should notify the other side.
  - A mailbox type of notification, with vdev id, and which field is changed.

Virtio Config Ops APIs

API	Description
void (*get)(struct virtio_device *vdev, unsigned offset, void *buf, unsigned len);	read the value of a configuration field
void (*set)(struct virtio_deviced *vdev, unsigned offset, void *buf, unsigned len);	write the value of a configuration field
uint32_t (*generation)(struct virtio_device *vdev);	read configure generation counter
u8 (*get_status)(struct virtio_device *vdev);	read the virtio status byte
void (*set_status)(struct virtio_device *vdev, u8 status);	write the virtio status byte
void (*reset)(struct virtio_device *vdev);	set the virtio device (used by virtio front end)
int (*find_vqs)(struct virtio_device *, unsigned nvqs, struct virtqueue *vqs[], vq_callback_t *callbacks[], const char * const names[], const bool *ctx, struct irq_affinity *desc);	find virtqueues and instantiate them
void (*del_vqs)(struct virtio_device *);	free virtqueues found by find _vqs()
uint64_t (*get_features)(struct virtio_device *vdev);	get the array of feature bits for this device.
int (*finalize_features)(struct virtio_device *vdev);	confirm what device features we'll be using

Virtio Device Configuration Space (1/2)

- In OpenAMP, the virtio device configuration space is in shared memory
  - It is in the resource table today

id	Viritio device id
Notified id	(unique to remoteproc)
dfeatures	virtio device features supported by the firmware
gfeatures	host to write back the negotiated features that are supported by both sides
config_len	The length of the config space right after vrings
status	The host will indicate its virtio progress
num_of_vrings	Number of vrings
Vrings resource	Resource description for vrings
Virtio driver related config space	Virtio driver defined configuration

Virtio Device Configuration Space (2/2)

- It is missing the generation counter information
- Shall we add the generation counter to virtio config space?
  - It is used for device to indicate there is change in the configuration
  - We can have generation counter in the vdev resource space, and reserved for future use
    - Enable to support more than RPMsg virtio
- Virtio doesn't support remote to reports its status. It is difficult to restart connection if the remote restarts.
  - Shall we work with virtio specification to introduce virtio backend to report its status?

Remoteproc APIs

Remoteproc APIs provides:

- Resource table handling
- remote life cycles management
- Remote notification
- Provide virtio config operations

Remoteproc Resource Table Handling

- The sequence of the resource in the resource table is the sequence the remoteproc will handle the resource.
- Resource table can be provided from the application image
  - The remoteproc will parse the resource table before loading the firmware
- Resource table is optional to application image
- Resource table should be parsed before remote application starts
- If resource table is in the application image, it will be passed before loading the image data into the target memory

Remoteproc Operations APIs

API (User defined remoteproc operations)	Description

struct remoteproc_ops rproc_ops {
  .init = user_rproc_init, /* mandatory */
  .remove = user_rproc_remove, /* mandatory */
  .mmap = user_rproc_mmap, /* optional, but will be used in both life cycle management and parsing rsc */
  .handle_rsc = user_rproc_handle_vendor_rsc, /* optional, required if there is vendor specific resource in the rsc */
  .start = user_rproc_start, /* optional, required for life cycle management */
  .stop = user_rproc_stop, /* optional, required for life cycle management */
  .shutdown = user_rproc_shutdown, /* optional, required for life cycle management */
  .kick = user_rproc_kick, /* optional, required for communication */
};	

struct remoteproc *(*user_remoteproc_init)(void *priv, struct remoteproc_ops *ops);	User defined remoteproc initialization
void (*user_remoreproc_remove)(struct remoteproc *rproc);	User defined removing remoteproc resource
void *(*user_remoteproc_mmap)(struct remoteproc *rproc, metal_phys_addr_t pa, metal_phys_addr_t da, size_t size, unsigned int attribute, struct metal_io_region **io);	User defined memory map operation, it will be used when passing the resource table, and when parsing the remote application image.
int (*user_remoteproc_handle_rsc)(struct remoteproc *rproc, void *rsc, size_t len);	User defined vendor specific resource handling
int (*user_remoteproc_start)(struct remoteproc *rproc);	User defined remote processor start function
int (*user_remoteproc_stop)(struct remoteproc *rproc);	User defined remote processor stop function, the processor resource is not released.
int (*user_remoteproc_shutdown)(struct remoteproc *rproc);	User defined remote processor offline function, the processor will be shutdown and its resource will be released.
void (*user_remtoeproc_kick)(*struct remoteproc *rproc, int id)	User defined kick operation

Remoteproc Operations APIs Examples

API (User defined remoteproc operations)	Description

struct remoteproc_ops rproc_ops {
.init = user_rproc_init,
.remove = user_rproc_remove,
.mmap = user_rproc_mmap,
.handle_rsc = user_rproc_handle_vendor_rsc,
.start = user_rproc_start,
.stop = user_rproc_stop,
.shutdown = user_rproc_shutdown,
.kick = user_rproc_kick,
};
	Remoteproc operations definition in case both LCM and communication is required

struct remoteproc_ops rproc_ops {
.init = user_rproc_init,
.remove = user_rproc_remove,
.mmap = user_rproc_mmap,
.handle_rsc = user_rproc_handle_vendor_rsc,
.start = user_rproc_start,
.stop = user_rproc_stop,
.shutdown = user_rproc_shutdown,
};
	Remooteproc operations definition in case only LCM is used

struct remoteproc_ops rproc_ops {
.init = user_rproc_init,
.remove = user_rproc_remove,
.mmap = user_rproc_mmap,
.handle_rsc = user_rproc_handle_vendor_rsc, /* optional */
.kick = user_rproc_kick,
};
	Remoteproc operations definition in case only communication is used

Remoteproc Core APIs (1 / 3)

API	Description
struct virtio_dev *rproc_virtio_create(struct remoteproc *rproc);	create remoteproc virtio device
struct handle_rsc_table(struct remoteproc *rproc, struct resource_table *rsc_table, int len);	handle resource table
int remtoeproc_load(struct remoteproc *rproc, void *fw, struct image_store_ops *store_ops);	load firmware for the remote processor system
int remoteproc_start(struct remoteproc *rproc);	start the remote processor
int remoteproc_stop(struct remoteproc *rproc);	stop the remote processor, halt the processor, but the resource of the processor such as memory is not released
int remoteproc_shutdown(struct remoteproc *rproc);	shutdown the remote processor and release its resource
int remoteproc_suspend(struct remoteproc *rproc)	suspend the remote processor system
int remoteproc_resume(struct remoteproc *rproc)	resume the remote processor system
struct remoteproc *remoteproc_init(struct remoteproc_ops *ops, void *priv);	Initialize remote processor system instance
void remoteproc_remove(struct remoteproc *rproc)	Remove remoteproc instance and its resource

Libmetal APIs

Libmetal provides an abstraction layer for device operation and I/O region abstraction, and other helper functions such as list, locks and so on.

- Libmetal APIs is used by rpmsg and remoteproc implementation in OpenAMP code base for requesting shared memories and operate shared memories.

Libmetal APIs used in OpenAMP library(1 / 3)

API	Description
int metal_register_generic_device(struct metal_device *device);	statically register a generic libemtal device. The subsequent calls to metal_device_open() look up in this list of pre-registered devices on the “generic” bus.
int metal_device_open(const char *bus_name, const char *dev_name, struct metal_device **device);	open a libmetal device by name
int (*user_remoteproc_handle_rsc)(struct remoteproc *rproc, void *rsc, size_t len);	User defined vendor specific resource handling
void metal_device_close(struct metal_device *device);	close a libmetal device
struct metal_io_region * metal_device_io_region(struct metal_device *device, unsigned index)	Get an I/O region accessor for a device region

Libmetal APIs used in OpenAMP library (2 / 3)

API	Description
#define METAL_IO_DECLARE((struct metal_io_region *io, void *virt, const metal_phys_addr_t *physmap, size_t size, unsigned page_shift, unsigned int mem_flags, const struct metal_io_ops *ops)	open an libmetal I/O region dynamically
int metal_io_init((struct metal_io_region *io, void *virt, const metal_phys_addr_t *physmap, size_t size, unsigned page_shift, unsigned int mem_flags, const struct metal_io_ops *ops);	open an libmetal I/O region dynamically
void metal_io_finish(struct metal_io_region *io);	close a libmetal I/O region
size_t metal_io_region_size(struct metal_io_region *io)	get the size of a libmetal I/O region

void *metal_io_virt(struct metal_io_region *io, unsigned long offset;);
unsigned long metal_io_virt_to_offset(struct metal_io_region *io, void *virt);
metal_phys_addr_t metal_io_phys(struct metal_io_region *io, unsigned long offset);
unsigned long metal_io_phys_to_offset(struct metal_io_region *io, metal_phys_addr_t phys);
void *metal_io_phys_to_virt(struct metal_io_region *io, metal_phys_addr_t phys);
metal_phys_addr_t
metal_io_virt_to_phys(struct metal_io_region *io, void *virt);
	conversion between virtual address, physical address, and offset

uint64_t metal_io_read8/16/32/64(struct metal_io_region *io, unsigned long offset); void metal_io_write8/16/32/64(struct metal_io_region *io, unsigned long offset, uint64_t value);	metal I/O read/write

int metal_io_block_read(struct metal_io_region *io, unsigned long offset, void *restrict dst, int len);
int metal_io_block_write(struct metal_io_region *io, unsigned long offset, const void *restrict src, int len);
int metal_io_block_set(struct metal_io_region *io, unsigned long offset, unsigned char value, int len);
	metal I/O region block read/write/set

Libmetal APIs used in OpenAMP library (3 / 3)

API	Description

struct metal_generic_shmem {
const char *name; /**< name of the shared memory */
struct metal_io_region io; /**< I/O region of the shared memory */
struct metal_list node; /**< list node */
};	

int metal_shmem_register_generic(struct metal_generic_shmem *shmem);	statically register a generic shared memory. Subsequent calls to metal_shmem_open() look up in this list of pre-registered regions.
int metal_shmem_open(const char *name, size_t size, struct metal_io_region **io);	Open a shared memory with the specified name