EMBEDDED
OPEN SOURCE
SUMMIT

Zephyr® Project
Developer Summit

Remote Processor Framework in Zephyr RTOS

2024-04-18, Seattle

Vaishnav Achath

Texas Instruments

About us: TI Processors and Open source

Decades of contribution and collaboration

Ingrained culture to give back to the community

Upstream FIRST!

Focus on long term, sustainable and quality products

Upstream and opensource ecosystem in device architecture

Upstream FIRST mentality!

Speaker | Intro

Vaishnav Achath, Software Engineer at Texas Instruments India. Vaishnav primarily works on Linux Kernel and U-Boot as part of the Texas Instruments Linux development team. Vaishnav is also the maintainer for TI platforms in Zephyr RTOS.

Disclaimers

- This is a technology presentation, not product-readiness or roadmap commitment
- Opinions presented here are that of the speakers and may not reflect that of Texas Instruments Inc.
- This is a proposal presentation for a **work-in-progress implementation** ,and discussion on benefits of having a separate subsystem for remoteproc management and is not a complete solution.

Overview

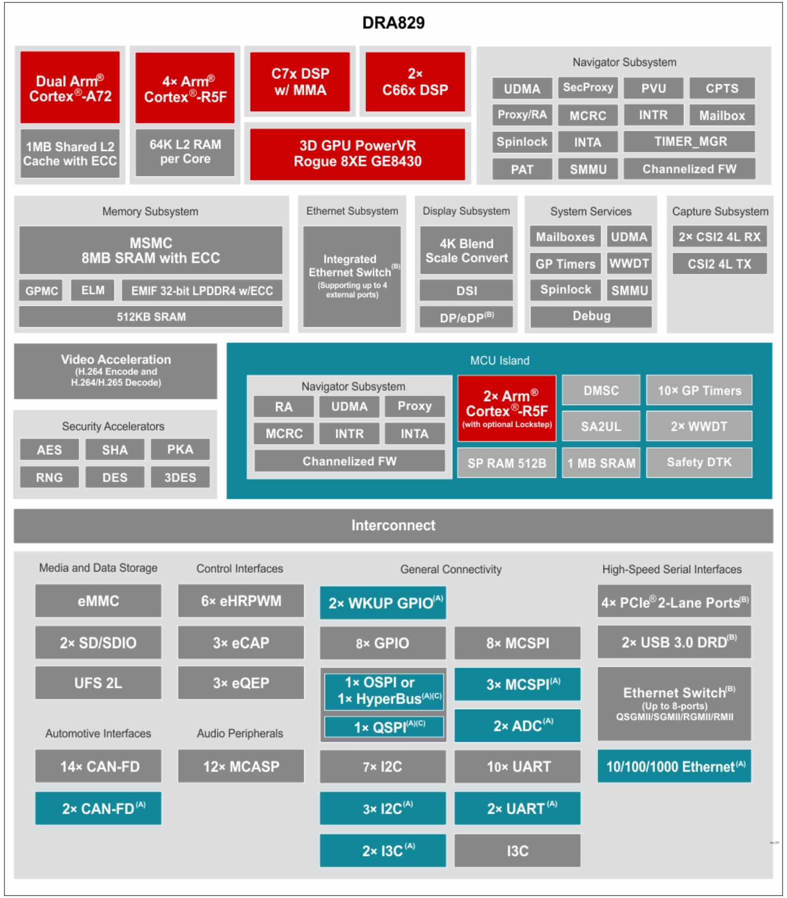
- Remote processor
- Use cases and Motivation
- Remote processor framework in Linux/U-Boot
- Proposal for Remote processor framework in Zephyr RTOS
  - Host implementation
  - Client implementation
- Benefits of Remoteproc framework in Zephyr RTOS
- Open Items and Future plans

Remote Processor

Modern SoCs have heterogeneous multicore processors with multiple remote cores in an Asymmetric multi-processing (AMP) configuration. Example:

- Multiple ARM Application cores (Dual-core ARM Cortex A72)
- Multiple ARM Realtime cores (6 x ARM Cortex R5)
- DSP Cluster (C7x, C6x)
- Central System Controller

Remote Processors are other processing entities in a system that are managed by software running on one of the cores(host).



TI DRA829 SoC Block Diagram

Remote Processor

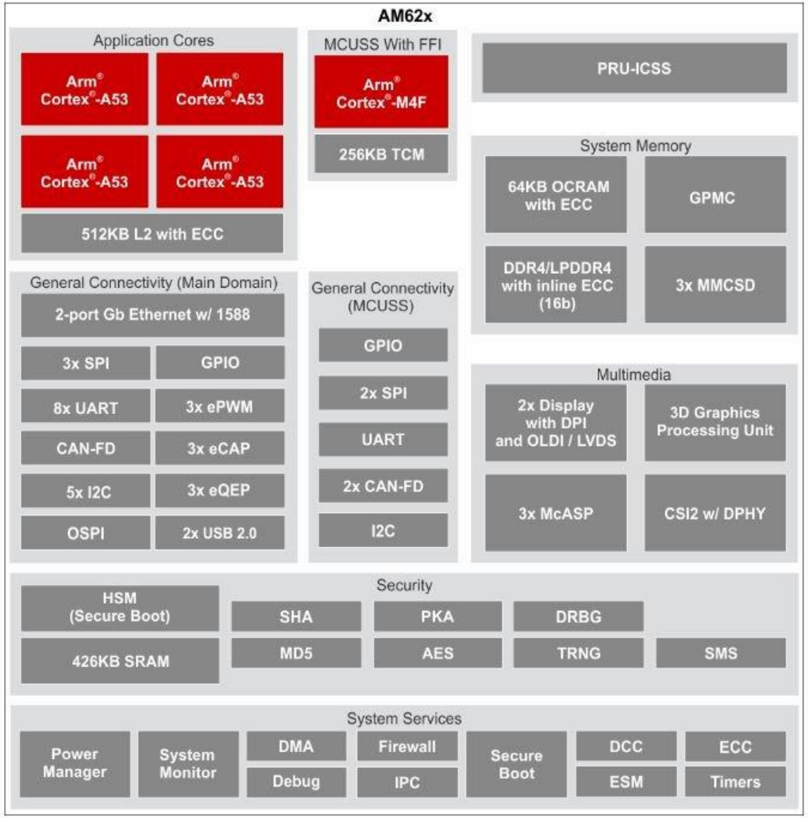
Today there is a remoteproc framework in Linux and U-Boot for the management of remote cores. From Linux and U-Boot you can,

– Load firmware on a remote processor.

– Start a remote processor.

– Stop/Shutdown a remote processor.

– Linux can attach to a remote processor booted by U-Boot, for Inter Processor Communication through RPMsg framework.

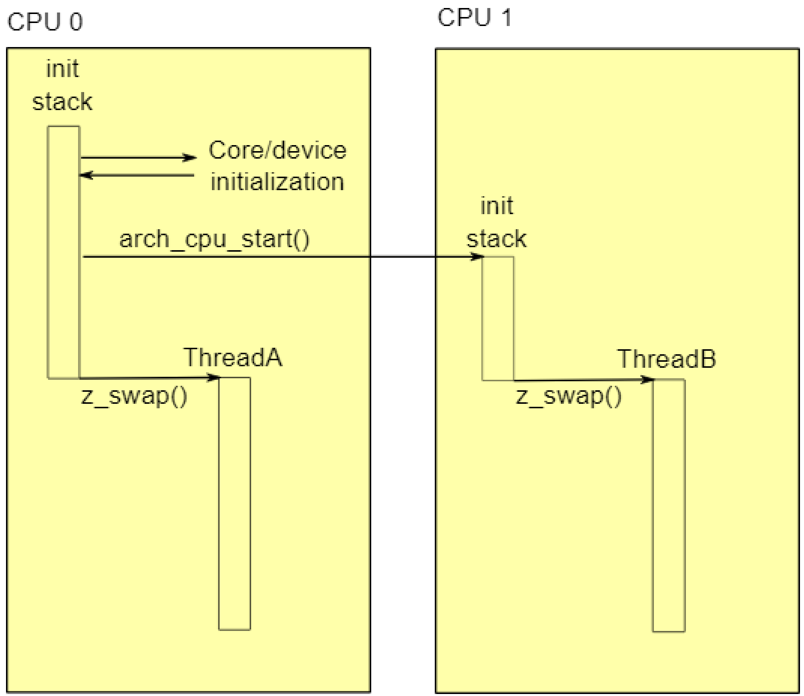


TI AM62X SoC Block Diagram

Why use Zephyr on Application Processors?

While Linux is the popular OS choice for running on Application Processors due to its versatility, Zephyr RTOS running on an Application core in SMP mode has very good potential to be a Remoteproc Host due to the following features:

- Real-time
- Subsystems like networking, LVGL, Display, USB .etc.
- Symmetric Multiprocessing support
- Power management.
- Simplicity compared to Linux and smaller application size.



Zephyr RTOS SMP Boot Process

Linux , Realtime Linux and Zephyr RTOS

Versatility

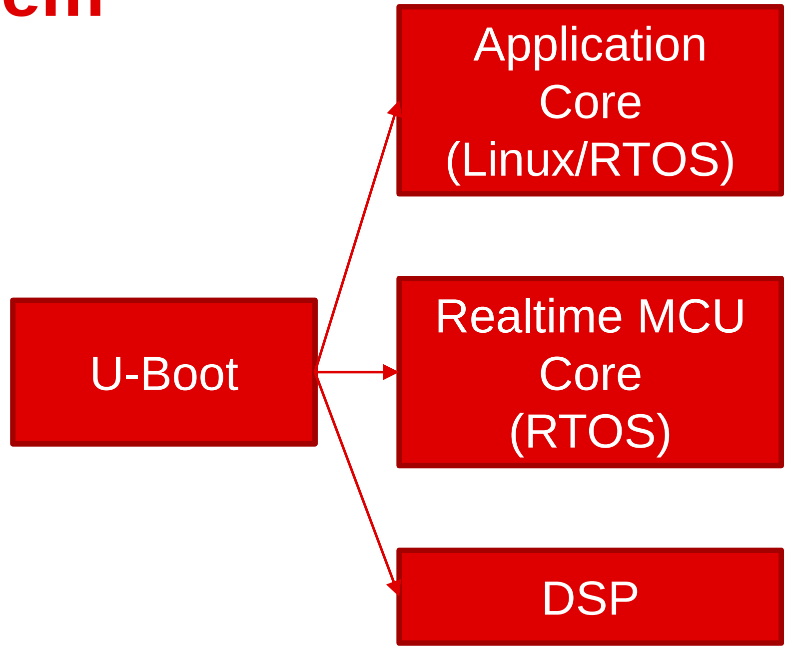
Complexity

Real-time

U-Boot Remoteproc Subsystem

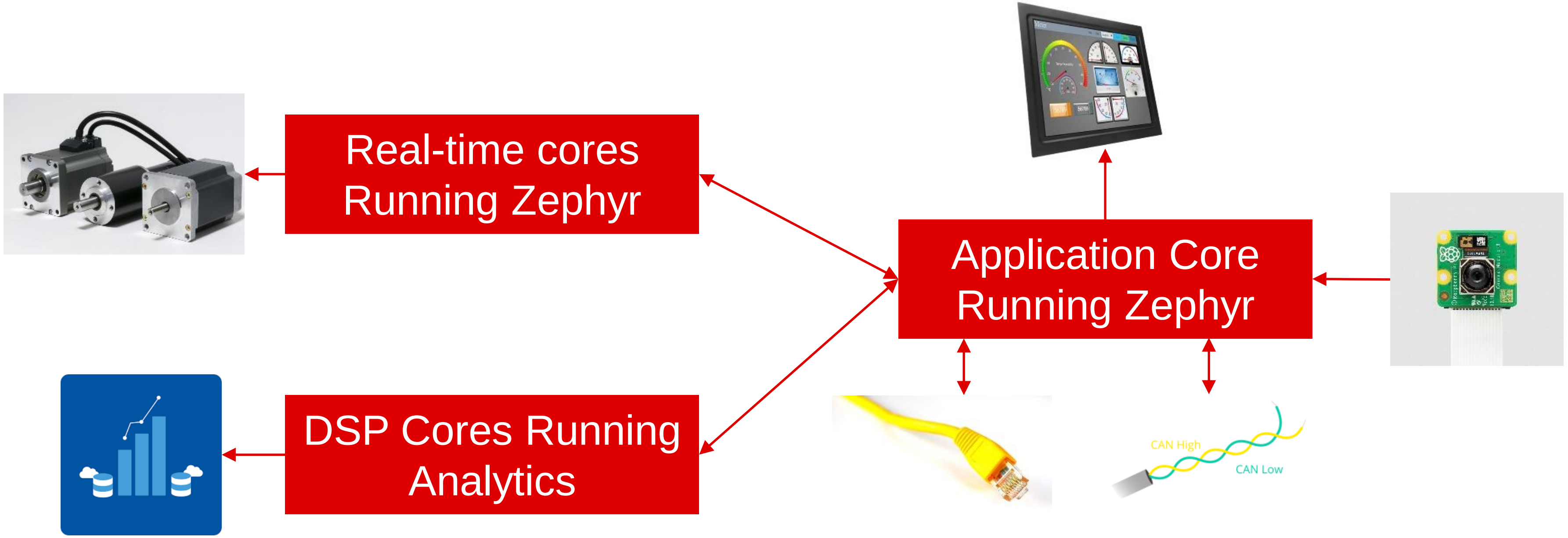
U-Boot Remoteproc framework can boot remote cores from the bootloader, but starting up remote cores just at the boot time might not cover all the system-level use cases, For example:

- For power management, the system may choose to suspend and resume at times to save power, in this scenario, if the remoteproc support is only present in U-Boot, then after suspending and resuming, the remote processors in the system will not be booted properly.
- In the above case, if Linux is the remoteproc host, then it can resume the operation of the remote cores using the existing framework.



Motivation

Zephyr can run on multiple cores on the same system where on a real-time core it is used for control systems, Application cores it may be used for Human Machine interaction, networking (Ethernet, CAN .etc), video capture, and then DSP cores run analytics,



Linux Remote Processor Subsystem

The remoteproc framework in Linux allows different platforms/architectures to control:

– Power On

– Load firmware

– Power off

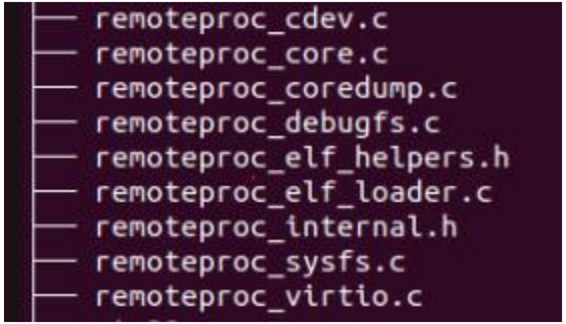
the remote processors while abstracting the hardware differences, so the entire driver doesn’t need to be duplicated.

In addition, this framework also adds rpmsg virtio devices for remote processors that supports this kind of communication. This way, platform-specific remoteproc drivers only need to provide a few low-level handlers, and then all rpmsg drivers will then just work.

The subsystem implementation is mainly split into:

- remoteproc_core -> core helpers for start, stop, prepare, unprepare .etc
- remoteproc_elf_loader -> cross architecture elf loader.
- remoteproc_virtio -> virtio helpers
- remoteproc_cdev -> Char dev interface implementation.

drivers/remoteproc/



Resource Table

A resource table is essentially a list of system resources required by the remote processor. It may also include configuration entries. If needed, the remote processor firmware should contain this table as a dedicated ".resource_table" ELF section.

Some resource entries are mere announcements, where the host is informed of specific remoteproc configurations. Other entries require the host to do something (e.g. allocate a system resource). Sometimes a negotiation is expected, where the firmware requests a resource, and once allocated, the host should provide back its details (e.g. address of an allocated memory region).

```
struct resource_table {
      u32 ver;
      u32 num;
      u32 reserved[2];
      u32 offset[];
} __packed;
```

Resource Table Header

```
struct fw_rsc_hdr {
      u32 type;
      u8 data[];
} __packed;
```

Resource Entry Header

rproc->ops->prepare()

The prepare API prepares the necessary requirements for device loading.

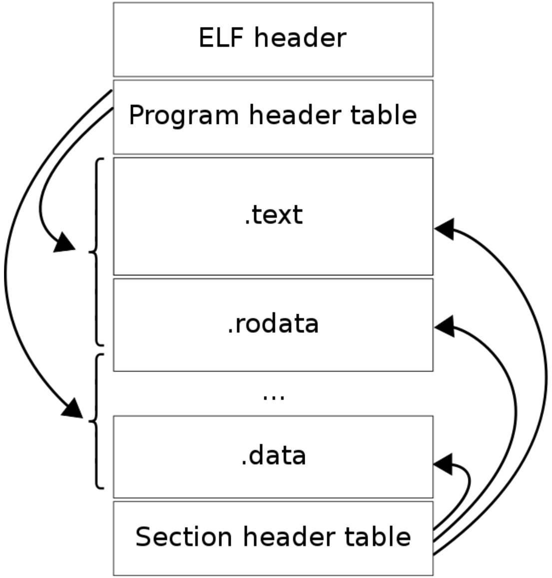
The prepare() ops is invoked by remoteproc core before any firmware loading, and is followed by the .start() ops after loading to actually let the R5 cores run.`

```
int (*prepare)(struct rproc *rproc);
```

```
static int rproc_prepare(struct rproc *rproc)
{
	struct my_r5_rproc *kproc = rproc->priv;
	struct device *dev = kproc->dev;
	int ret;

	/* Initialize memory */
	/* Power on necessary power domains */

	return 0;
}
```

rproc->ops->load()

Load firmware to memory, where the remote processor expects to find it

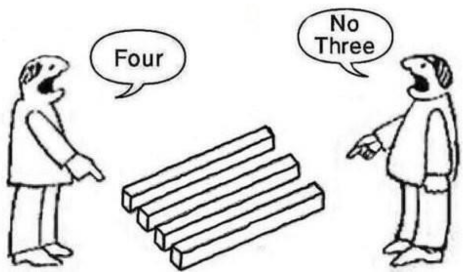



```c
int (*load)(struct rproc *rproc, const struct firmware *fw);
```

```c
int rproc_elf_load_segments(struct rproc *rproc, const struct firmware *fw)
{
	struct device *dev = &rproc->dev;
	const void *ehdr, *phdr;

	/* go through the available ELF segments */
	for (i = 0; i < phnum; i++, phdr += elf_phdr_get_size) {
		u64 da = elf_phdr_get_p_paddr(class, phdr);
		u64 memsz = elf_phdr_get_p_memsz(class, phdr);
		u64 filesz = elf_phdr_get_p_filesz(class, phdr);
		u64 offset = elf_phdr_get_p_offset(class, phdr);
		u32 type = elf_phdr_get_p_type(class, phdr);
		bool is_iomem = false;
		void *ptr;

		if (type != PT_LOAD || !memsz)
			continue;

		/* ELF Checks */
		/* grab the kernel address for this device address */
		ptr = rproc_da_to_va(rproc, da, memsz, &is_iomem);
		if (!ptr) {
			dev_err(dev, "bad phdr da 0x%llx mem 0x%llx\n", da,
				memsz);
			ret = -EINVAL;
			break;
		}

		/* put the segment where the remote processor expects it */
		if (filesz) {
			if (is_iomem)
				memcpy_toio((void __iomem *)ptr, elf_data + offset, filesz);
			else
				memcpy(ptr, elf_data + offset, filesz);
		}
	}

	return ret;
}
```

rproc->ops- >da_to_va()

Custom function to provide address translation (device address to kernel virtual address) for internal RAMs present in a DSP or IPU device). The translated addresses can be used either by the remoteproc core for loading, or by any rpmsg bus drivers




```
void * (*da_to_va)(struct rproc *rproc, u64 da, size_t len, bool *is_iomem);
```

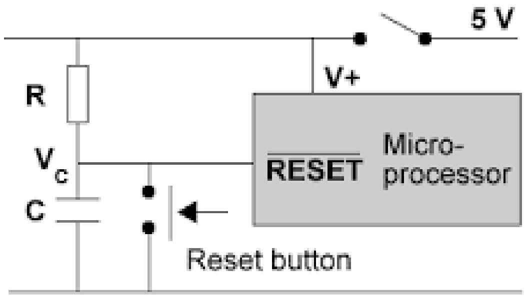
```
static void *k3_r5_rproc_da_to_va(struct rproc *rproc, u64 da, size_t len, bool *is_iomem)
{
    struct k3_r5_rproc *kproc = rproc->priv;
    struct k3_r5_core *core = kproc->core;
    void __iomem *va = NULL;
    phys_addr_t bus_addr;
    u32 dev_addr, offset;
    size_t size;
    int i;

    if (len == 0)
        return NULL;

    /* handle both R5 and SoC views of ATCM and BTCM */
    for (i = 0; i < core->num_mems; i++) {
        bus_addr = core->mem[i].bus_addr;
        dev_addr = core->mem[i].dev_addr;
        size = core->mem[i].size;

        /* handle R5-view addresses of TCMs */
        if (da >= dev_addr && ((da + len) <= (dev_addr + size))) {
            offset = da - dev_addr;
            va = core->mem[i].cpu_addr + offset;
            return (__force void *)va;
        }

        /* handle SoC-view addresses of TCMs */
        if (da >= bus_addr && ((da + len) <= (bus_addr + size))) {
            offset = da - bus_addr;
            va = core->mem[i].cpu_addr + offset;
            return (__force void *)va;
        }
    }
}
```

rproc->ops- > start()

The rproc start function is used to power on the CPU and start the execution (programming boot address, release reset .etc)

```c
int (*start)(struct rproc *rproc);
```

```c
static int k3_r5_rproc_start(struct rproc *rproc)
{
	struct k3_r5_rproc *kproc = rproc->priv;
	struct k3_r5_cluster *cluster = kproc->cluster;
	struct device *dev = kproc->dev;
	struct k3_r5_core *core;
	u32 boot_addr;
	int ret;

	ret = k3_r5_rproc_request_mbox(rproc);
	if (ret)
		return ret;

	boot_addr = rproc->bootaddr;

	/* boot vector need not be programmed for Core1 in LockStep mode */
	core = kproc->core;
	ret = ti_sci_proc_set_config(core->tsp, boot_addr, 0, 0);
	if (ret)
		goto put_mbox;

	ret = ti_sci_proc_set_control(core->tsp,
				      0, PROC_BOOT_CTRL_FLAG_R5_CORE_HALT);
	if (ret)
		goto put_mbox;

	return 0;
}
```



TEXAS INSTRUMENTS

rproc->ops->stop

The stop function provides implementation to stop or reset the core, depending on the remote core requirements there could be a sequence of power down for processors in a cluster.


```c
int (*stop)(struct rproc *rproc);
```

```c
static int k3_r5_rproc_stop(struct rproc *rproc)
{
	struct k3_r5_rproc *kproc = rproc->priv;
	struct k3_r5_cluster *cluster = kproc->cluster;
	struct k3_r5_core *core = kproc->core;
	int ret;

	/* halt all applicable cores */

	ret = ti_sci_proc_set_control(core->tsp,
					PROC_BOOT_CTRL_FLAG_R5_CORE_HALT, 0)
	if (ret)
		goto out;


	mbox_free_channel(kproc->mbox);
}
```

rproc->ops->unprepare()

The unprepare function provides implementation to power down the remote cores, and is usually invoked after a stop operation. This performs the complementary operations of that of prepare() API.

```c
int (*unprepare)(struct rproc *rproc);
```

```c
static int k3_r5_lockstep_reset(struct k3_r5_cluster *cluster)
{
    struct k3_r5_core *core;
    int ret;

    /* assert local reset on all applicable cores */
    list_for_each_entry(core, &cluster->cores, elem) {
        ret = reset_control_assert(core->reset);
        if (ret) {
            dev_err(core->dev, "local-reset assert failed, ret = %d\n",
                ret);
            core = list_prev_entry(core, elem);
            goto unroll_local_reset;
        }
    }

    /* disable PSC modules on all applicable cores */
    list_for_each_entry(core, &cluster->cores, elem) {
        ret = core->ti_sci->ops.dev_ops.put_device(core->ti_sci,
                                                   core->ti_sci_id);
        if (ret) {
            dev_err(core->dev, "module-reset assert failed, ret = %d\n",
                ret);
            goto unroll_module_reset;
        }
    }

    return 0;
}
```

Remote Processor Client interface

Linux and U-Boot has remoteproc host implementations, but Zephyr can run of processors that can act as remoteproc host and also on coprocessors that can act as remoteproc client, in case of client implementation, Zephyr already has the necessary support with the help of openAMP. Since there is no subsystem there is a bit of duplication of effort across multiple platforms, also there are some challenges like:

- Graceful shutdown of remote cores.
- Client firmware needs to request resources like carveout memory, device memory .etc

Since these functions are common and can be reused across multiple platforms, it helps to have helper functions implemented under a common subsystem.

```
soc/nxp/imx/imx8m/m7/CMakeLists.txt:17:  zephyr_linker_section(NAME .resource_table GROUP ROM_REGION NOINPUT)
soc/nxp/imx/imx8m/m7/CMakeLists.txt:18:  zephyr_linker_section_configure(SECTION .resource_table KEEP INPUT ".resource_table*")
soc/nxp/imx/imx8m/m7/linker.ld:24:        SECTION_PROLOGUE(.resource_table,, SUBALIGN(8))
soc/nxp/imx/imx8m/m7/linker.ld:26:            KEEP(*(.resource_table*))
soc/nxp/imx/imx8m/adsp/linker.ld:175:    SECTION_PROLOGUE(.resource_table,, SUBALIGN(4))
soc/nxp/imx/imx8m/adsp/linker.ld:177:      KEEP(*(.resource_table*))
soc/nxp/imx/imx8m/m4_mini/CMakeLists.txt:15:  zephyr_linker_section(NAME .resource_table GROUP ROM_REGION NOINPUT)
soc/nxp/imx/imx8m/m4_mini/CMakeLists.txt:16:  zephyr_linker_section_configure(SECTION .resource_table KEEP INPUT ".resource_table*")
soc/nxp/imx/imx8m/m4_mini/linker.ld:12:        SECTION_PROLOGUE(.resource_table,, SUBALIGN(8))
soc/nxp/imx/imx8m/m4_mini/linker.ld:14:            KEEP(*(.resource_table*))
soc/st/stm32/stm32mp1x/linker.ld:18:    SECTION_PROLOGUE(.resource_table,, SUBALIGN(4))
soc/st/stm32/stm32mp1x/linker.ld:20:          KEEP(*(.resource_table*))
soc/ti/k3/am6x/CMakeLists.txt:16:    zephyr_linker_section(NAME .resource_table GROUP ROM_REGION NOINPUT)
soc/ti/k3/am6x/CMakeLists.txt:17:    zephyr_linker_section_configure(SECTION .resource_table KEEP INPUT ".resource_table*")
soc/ti/k3/am6x/m4/linker.ld:13: SECTION_PROLOGUE(.resource_table,, SUBALIGN(4))
soc/ti/k3/am6x/m4/linker.ld:15:         KEEP(*(.resource_table*))
```

Duplication of linker commands related to adding resource table (common linker include?)

Remote Processor Host Proposal for Zephyr RTOS

include/zephyr/drivers/remoteproc.h

```c
__subsystem struct remoteproc_driver_api {
    /** prepare the  remote processor */
    int (*prepare)(const struct device *dev);

    /** disable the  remote processor */
    int (*unprepare)(const struct device *dev);

    /** start the  remote processor */
    int (*start)(const struct device *dev);

    /** load firmware to the remote processor */
    int (*load)(const struct device *dev, void *fw_start, size_t len);

    /** Convert device address to virtual address */
    void * (*da_to_va)(const struct device *dev, mem_addr_t da, size_t len);
};
```

```c
static int rproc_k3_m4_prepare(const struct device *dev)
{
    const struct rproc_k3_m4_config *config = dev->config;
    struct rproc_k3_m4_data *data = dev->data;

    ti_sci_cmd_proc_request(dev, config->host_id);
    ti_sci_set_device_state(dev, config->device_id, MSG_FLAG_DEVICE_EXCLUSIVE, MSG_DEVICE_SW_STATE_ON);

    ti_sci_cmd_set_device_resets(dev, config->device_id, 1);

    ti_sci_cmd_proc_release(dev, config->host_id);
}
```

```c
static int rproc_k3_m4_start(const struct device *dev)
{
    const struct rproc_k3_m4_config *config = dev->config;
    struct rproc_k3_m4_data *data = dev->data;

    ti_sci_cmd_proc_request(dev, config->host_id);

    ti_sci_cmd_set_device_resets(dev, config->device_id, 0);

    ti_sci_cmd_proc_release(dev, config->host_id);
}
```

```c
static int rproc_k3_m4_unprepare(const struct device *dev)
{
    const struct rproc_k3_m4_config *config = dev->config;
    struct rproc_k3_m4_data *data = dev->data;

    ti_sci_cmd_proc_request(dev, config->host_id);
    ti_sci_set_device_state(dev, config->device_id, MSG_FLAG_DEVICE_EXCLUSIVE, MSG_DEVICE_SW_STATE_OFF);

    ti_sci_cmd_proc_release(dev, config->host_id);
}
```

ELF Loader and Device Address to Virtual Address Conversion

```c
int rproc_elf32_load_image(unsigned long addr, unsigned long size)
{
    struct elf32_ehdr *ehdr; /* Elf header structure pointer */
    struct elf32_phdr *phdr; /* Program header structure pointer */
    unsigned int i, ret;

    ret =  rproc_elf32_sanity_check(addr, size);
    if (ret) {
        LOG_ERR("Invalid ELF32 Image %d\n", ret);
        return ret;
    }

    ehdr = (struct elf32_ehdr *)addr;
    phdr = (struct elf32_phdr *)(addr + ehdr->e_phoff);

    /* Load each program header */
    for (i = 0; i < ehdr->e_phnum; i++, phdr++) {
        void *dst = (void *)(uintptr_t)phdr->p_paddr;
        void *src = (void *)(addr + phdr->p_offset);

        if (phdr->p_type != PT_LOAD)
            continue;

        dst = k3_m4_da_to_va((unsigned long)dst, phdr->p_memsz);

        LOG_DBG("Loading phdr %i to %p (%i bytes)\n",
            i, dst, phdr->p_filesz);
        if (phdr->p_filesz)
            memcpy(dst, src, phdr->p_filesz);
        if (phdr->p_filesz != phdr->p_memsz)
            memset((void *)((unsigned long)dst + phdr->p_filesz), 0x00,
                   phdr->p_memsz - phdr->p_filesz);
        /* Flush Cache */
    }

    return 0;
}
```

```c
static void *k3_m4_da_to_va(unsigned long da, unsigned long len)
{
    mem_addr_t bus_addr, dev_addr;
    mem_addr_t cpuaddr, va;
    size_t size;
    uint32_t offset;

    if (len <= 0)
        return NULL;

    LOG_DBG("mapping %lx size %lx", da, len);

    device_map((mm_reg_t *)&cpuaddr, bus_addr, size, K_MEM_CACHE_NONI
    if (da >= dev_addr && ((da + len) <= (dev_addr + size))) {
        offset = da - dev_addr;
        va = cpuaddr + offset;
        return (void *)va;
    }

    if (da >= bus_addr && (da + len) <= (bus_addr + size)) {
        offset = da - bus_addr;
        va = cpuaddr + offset;
        return (void *)va;
    }

    /* Assume it is DDR region and return da */
    device_map((mm_reg_t *)&cpuaddr, da, len, K_MEM_CACHE_NONE);
    return (void *)cpuaddr;
}
```

Summary

Host Mode

- Remoteproc HOST allows to boot and interact with other cores running firmware with a resource table.
- Remoteproc HOST support is similar to what is found in Linux/U-Boot.
- Host allows to load, start, reset, stop other remote cores.

Client Mode

- Currently supported in Zephyr, improvements to be made to open-amp/lib/remoteproc
- Create a generic framework for firmware booting in remote processor mode
- Client framework handles packing of resource table and other complexities associated with preparing firmware for remote proc host booting.

Benefits of Remoteproc framework in Zephyr

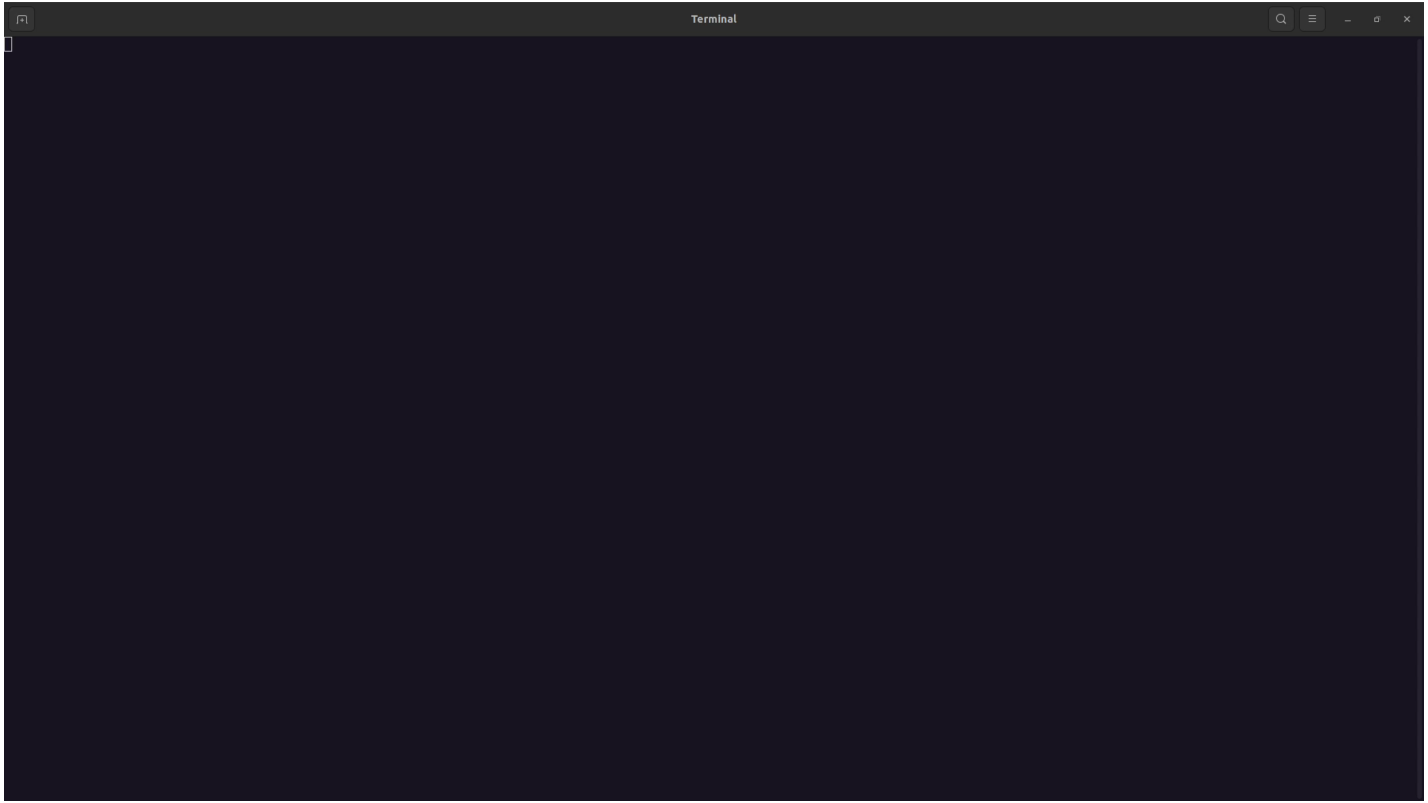
- MCUBoot as a general-purpose bootloader for AMP systems.
  - MCUBoot has rich set of security features and firmware update features (Over the Air firmware update for remoteprocessors?)
- Reusing remote processor host and client mode code without duplicating for each platform.
- Enable applications involving multiple heterogenous cores like ARM64, ARMv7 and DSP cores working together.

Demo: Booting from U-Boot

Terminal

```
U-Boot SPL 2023.04-dirty (Apr 12 2024 - 16:35:32 +0530)
SYSFW ABI: 3.1 (firmware rev 0x0009 '9.2.7--v09.02.07 (Kool Koala)')
SPL initial stack usage: 13440 bytes
Trying to boot from MMC2
Warning: Detected image signing certificate on GP device. Skipping certificate to prevent boot failure. T
his will fail if the image was also encrypted
Warning: Detected image signing certificate on GP device. Skipping certificate to prevent boot failure. T
his will fail if the
```

Demo: Booting from Zephyr



Open Items and Future Plans

- RFC Pull request
- Test cases
- Present in Dev Review
- Client implementation
  - Graceful shutdown of cores
  - Resource requests to host (resource table configuration)
- MCUBoot booting multiple firmware on a heterogenous multicore processor.
- Currently using ELF helpers from include/zephyr/llext/elf.h , reuse openAMP implementations if available.

References

- Linux Remote Processor Framework - https://docs.kernel.org/staging/remoteproc.html
- Remote Processor Messaging Framework - https://docs.kernel.org/staging/rpmsg.html
- OpenAMP Project - https://www.openampproject.org/

Credits and Acknowledgement

Thank you!

- Texas Instruments Inc.
- The Linux Foundation.
- The OpenAMP Project.
- Suman Anna, Texas Instruments
- Andrew F Davis, Texas Instruments
- Carlo Caione, Baylibre

Q&A

- Contact Information:
  - Vaishnav Achath vaishnav.a@ti.com
  - vaishnav on Zephyr Discord.
- Also on IRC @ libera.chat #linux-ti

Learn more about TI products

- https://www.ti.com/linux
- https://www.ti.com/processors
- https://www.ti.com/edgeai

Why choose TI MCUs and processors?

Scalability

Our products offer scalable performance that can adapt and grow as the needs of your customers evolve.

Efficiency

We design products that extend battery life, maximize performance for every watt expended, and unlock the highest levels of system efficiency.

Affordability

We strive to make innovation accessible to all by creating cost-effective products that feature state-of-the-art technology and package designs.

Availability

Our investment in internal manufacturing capacity provides greater assurance of supply, supporting your growth for decades to come.