Coprocessor Application Manual

PHYTEC Messtechnik GmbH

Oct 02, 2025

CONTENTS

Warning

This manual is a draft version and is currently **work in progress**. It will undergo significant changes over time.

We value your feedback, questions, and suggestions and encourage you to open an issue or pull request in the linked repository to get in contact.

Coprocessor Application Manual	
Document Title	Coprocessor Application Manual
Document Type	Generic Application Guide
Release Date	XXXX/XX/XX

This manual applies to all Phytec releases from kernel version x.

Most modern SoCs include one or more coprocessors beside an application processor. In most cases the application processor runs Linux, while the coprocessor may run an RTOS. This manual goes into detail how to utilize the coprocessor efficiently for projects.

The manual explains generic principles and applies those principles in examples for a specific platform and tools. It gives an introduction into coprocessor software stacks and RTOS like Zephyr, MCUXpresso and OpenAMP

For now this manual is focused on the NXP i.MX platform, but an attempt is made to keep the manual as generic as possible.

CHAPTER

ONE

INTERNAL VS. EXTERNAL COPROCESSOR

A coprocessor can be internal or external (Of the application processors SoC). Both have their advantages and disadvantages. Advantages on internal coprocessors are for example a more simple firmware update management, a more efficient communication between the coprocessor and the application processor and a probably more inexpensive PCB design. External coprocessors have, for example, the advantages of more interfaces in addition to the ones of the application processor, and they are starting up directly, not depending on the application processor.

This manual focuses on internal coprocessors of the PHYTEC SoMs.

CHAPTER

TWO

USE CASES

There are several use cases for coprocessors in embedded systems. Almost every time-critical task that cannot be handled by the application processor can be offloaded to a coprocessor.

Here are some more explicit use-case examples to give an idea of the possibilities:

2.1 Energy Constrained Applications

For energy constrained applications, it may be beneficial to reduce the active time of the entire SoC to conserve power. In such cases, the application processor can be put into sleep mode while the coprocessor remains active to, for example, monitor I2C communication and wake up the application processor upon receiving a specific command.

2.2 Time-Critical Communication

Some protocols may require sending or receiving data in real-time. If there is no hardware IP-core that is capable of handling the desired protocol, the coprocessor could help out to support it through building it in software.

2.3 Sensors and Real-Time

Some applications may require a sensor to be read in a time-critical manner (e.g. an accelerometer) to detect small value changes in a short time frame. This can be done by a coprocessor to ensure that the sensor is read at the right time. The data can be buffered and fed to the application processor if it has time to process the data.

2.4 Interface Virtualization

On SoCs like the i.MX9 series there is the FLEXIO interface (compare RPi PIO, Microchip CLC). Received data on this interface needs to be processed in a time-critical manner because it is lacking a FIFO buffer. If serial data with higher speeds is received, the application processor may need to process too many interrupts. That could slow down other running applications. Another problem is the interrupt latency. The application processor could possibly lose data frames.

The coprocessor can be used to read the data from the interface, buffer it and send it to the application processor when it has time to process it.

CHAPTER

THREE

OVERVIEW OF TECHNOLOGIES

3.1 Real-Time Operating Systems (RTOS) and Development Frameworks

There are multiple RTOS and SDKs available that can be used on coprocessors.

3.1.1 Zephyr

Zephyr is an Open Source and vendor neutral RTOS that is is governed by the Linux Foundation and supported by various companies and a large community. It is designed to be small and efficient and is suitable for a wide range of devices from simple embedded devices to complex SoCs. The key feature is the platform independence, which allows developing applications with a generic API that can run on multiple platforms without modification.

It supports a wide range of SoCs and boards from various manufacturers based on different processor architectures. There is also support for a lot of different peripherals and interfaces, as well as a wide range of communication protocols. (e.g. TCP/IP stack, Bluetooth, CAN, USB, etc.)

Zephyr supports the coprocessor on multiple phyBOARDs, including the phyBOARD Pollux (i.MX8MP), Polis (i.MX8MM), Nash (i.MX93), Electra (AM64x) and Lyra (AM62x).

It should be mentioned, that not all hardware features are available in Zephyr yet. The support is constantly being expanded through NXP and the Zephyr community. Despite this PHYTEC recommends using Zephyr for new projects because of its many advantages.

You can find more information about using Zephyr in the Zephyr Documentation website.

Hint

Please reach out to us if there is any feature missing in Zephyr that you need for your project. We will try our best to get that feature implemented.

3.1.2 MCUXpresso SDK (NXP)

The MCUX SDK is a software development kit for NXP microcontrollers and microprocessors. It provides a comprehensive set of peripheral drivers, middlewares and examples for all NXP Platforms. MCUX gives the possibility to use different RTOS like FreeRTOS, Azure RTOS or even using it BareMetal.

If Zephyr is not suitable for your project (e.g. because of missing features), MCUX SDK is the alternative. You can use it either with the MCUX SDK or repository-managed via make and cpp.

Here are some resources to get started with MCUX:

- MCUXpresso SDK

- PHYTEC MCUX-SDK
- MCUXpresso VS-Code IDE
- MCUXpresso IDE

3.2 Additional Software Stacks

3.2.1 OpenAMP

The OpenAMP Project "seeks to standardize the interactions between operating environments in a heterogeneous embedded system through open source solutions for Asymmetric MultiProcessing (AMP)."

This introduction explains the main components and terms, the OpenAMP documentation goes into further detail. OpenAMP is available in Linux as well as in RTOS (e.g. Zephyr) and Vendor SDKs (e.g. NXP MCUX, TI SDK, STM32Cube).

In general, OpenAMP is a framework that allows communication between asymmetric processor cores inside a SoC via shared memory.

A differentiation is made between a master core (mostly the application processor) and one or more remote cores (coprocessors). The master core has to load the firmware on the remote core, start it and prepare shared memory regions for communication.

The OpenAMP framework consists of two main components:

remoteproc

The remoteproc framework is used to control the life cycle of a remote processor. It is responsible for loading the firmware, starting and stopping the remote processor and managing the resources of both cores. The remoteproc documentation on Kernel.org goes into further technical details.

RPMsg

RPMsg is a messaging protocol that is used to exchange messages between the master core and remote cores. It is built on top of VirtIO and Virtqueue and uses the shared memory regions prepared by remoteproc to exchange messages.

The communication stack is consisting of several protocol layers, similar to the OSI model:

Transport Layer (3):
RPMsg

MAC Layer (2):
VirtIO, Virtqueue, Vring

Physical Layer (1):
Shared Memory, Inter-core Interrupts e.g. via Messaging Unit (MU)

Normally VirtIO is used to exchange messages between virtual machines in a hypervisor environment. In the context of OpenAMP, VirtIO is used to exchange messages between the master core and remote cores while being very efficient. The Virtqueue is underlying VirtIO and organizes the messages in a circular buffer. The Vring is the specific implementation of the buffer inside the Virtqueue.

The rpmsg documentation on Kernel.org goes into further technical details.

Requirements

Shared Memory
To exchange messages between the cores, a shared memory region is required.

Interrupts
Minimum set of one interrupt line per communicating core. This interrupt is often implemented in hardware blocks of the SoC, e.g. the "Messaging Unit (MU)" on the NXP i.MX8MP.

Resource Table
The resource table is a data structure that describes the shared memory regions and the VirtIO devices that are used for communication between the cores. It is used by the remoteproc framework to prepare the shared memory regions and the VirtIO devices. Ensure that the resource table is correctly included in the firmware binary of the remote core. (e.g. in Zephyr use `CONFIG_OPENAMP_RSC_TABLE=y`)

3.2.2 Protocol Buffers

CHAPTER

FOUR

APPLICATION ARCHITECTURES

4.1 Typical Usage

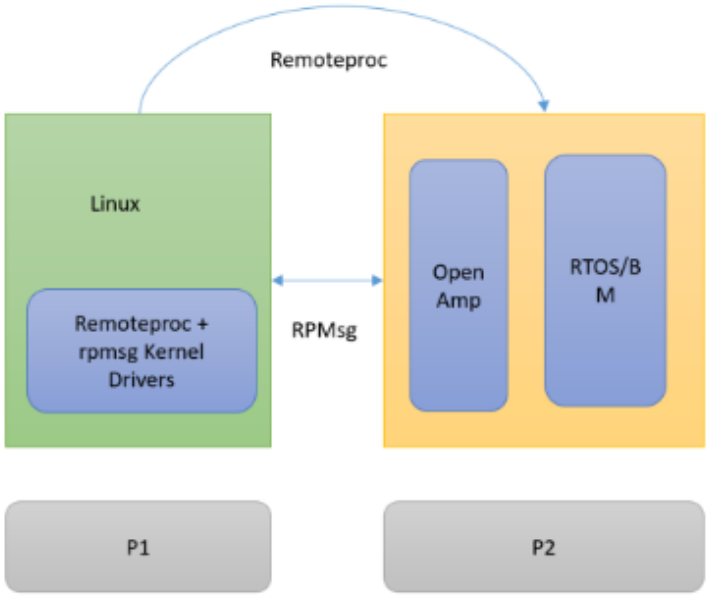

Fig. 1: Typical Application Architecture with OpenAMP (source: OpenAMP Whitepaper)

A typical application architecture when using OpenAMP is using two cores. One application processor (typically running Linux) while the coprocessor is processing time-critical tasks.

4.2 VirtIO

4.3 RPmsg + Overlaying Protocol

Sometimes it can be necessary to use an overlaying protocol on top of RPMsg to exchange more complex data structures.

This could be done with using a protocol like Protocol Buffers or Flat Buffers to serialize and deserialize the data structures.

CHAPTER

FIVE

GETTING STARTED

There are multiple ways to get started with using a coprocessor.

First of all you need to decide which RTOS you want to use.

If you want to use Zephyr, you can use the Zephyr Getting Started Guide.

Note

When building a Zephyr project / sample for a SoC, the board naming can be confusing. The naming convention is `<board>/<soc>/<core>`. For example, to build a Zephyr project for the phyBOARD Pollux (i.MX8MP) with the M7 core, the board name is `phyboard_pollux/mimx8ml8/m7`.

When compiling the firmware, you'll get two binary files. One `.elf` file for starting the remote processor via remoteproc, which includes the resource table, and one `.bin` file for starting the remote processor via the bootloader.

5.1 Starting the Coprocessor via Remoteproc

To start a remote processor via remoteproc, you need to place the firmware into the `/lib/firmware` directory on the target.

This can be done using SCP (e.g., for development), by copying the file to the SD card, or by including it in the Yocto build (e.g., for production use).

Make sure the devicetree overlay that enables remoteproc support is activated. You can find more information about how to activate the devicetree overlay in the BSP manual for your platform.

```
target:~$ echo /lib/firmware/{your_firmware}.elf > /sys/class/remoteproc/remoteproc0/firmware
target:~$ echo start > /sys/class/remoteproc/remoteproc0/state
```

Hint

If your device has multiple coprocessors, please make sure you use the correct remoteproc device.

5.2 Starting the Coprocessor via Bootloader

Starting the Coprocessor via the bootloader is platform specific. You can find more information in the BSP manual for your platform.

Using this method can be useful, if you want to have the coprocessor running before the application processor boots up, for example for applications that need to have a fast response time on startup.

Here is the manual for the i.MX8MP for example: Running the M7 Core

5.3 Starting the Coprocessor via Debug Probe

It is possible to start the coprocessor via a debug probe like J-Link or OpenOCD. This is useful for debugging the firmware on the coprocessor, or for starting up the coprocessor in a development environment.

On most PHYTEC boards, you can use a PEB-EVAL-01 shield to connect the debug probe to the board via a 20-pin JTAG connector.

When using Zephyr you can simply use the command

```
host:zephyrproject/zephyr$ west debug
```

to start GDB and load / start the firmware on the coprocessor.

Warning

Please note that it is not possible to use inter processor communication via RPMsg when not starting the coprocessor via remoteproc! This is because remoteproc prepares Linux and the shared memory for communication!

This is especially impractical when you want to debug your coprocessor firmware via a debug probe, if your system requires the use of communication between the cores.

5.4 Accessing the serial console

The coprocessor firmware can output messages via a serial console. It differs from platform to platform how to gain access to the serial console.

For example, on the i.MX8 platform, you'll get a serial console via the debug USB port on the board. On i.MX93 (on segin board) on the other hand, you can access it via RS232 on the PEB-EVAL-01.

It's recommended to take a look into the corresponding BSP or Zephyr manual for your platform to find out how to access the serial console.

Zephyr offers a shell backend to be able to access a shell via RPMsg. This can help for debugging purposes or to send commands to the coprocessor. Take a look here: *OpenAMP using resource table*

The easiest way to communicate on the Linux side through RPMsg is via the `tty-rpmsg` driver. This driver creates a tty device in `/dev` that can be used to send and receive messages to the coprocessor.

5.5 Debugging the Coprocessor

In some cases it can be necessary to get a deeper insight into the coprocessor firmware to find bugs or to optimize the performance. To do that you can use any JTAG debugger. The following sections describe it with the J-Link as an example.

If the firmware does not need to communicate with the application processor via RPMsg, the coprocessor can be started easily via the debug probe and debugged with GDB. (see *Starting the Coprocessor via Debug Probe*)

If the firmware needs to communicate with the application processor via RPMsg, the preparation in order to start the coprocessor and the communication between the cores is a bit more complex. This is because remoteproc prepares the shared memory and the Linux Kernel for communication but GDB also needs to know in which state the coprocessor is.

Hint

Before you start debugging, please make sure your J-Link is compatible with the architecture of the ARM-core you want to debug. Most cores are possible, for example the A- or M-Core or the DSP. You can find this information in the Segger knowledge base. (for example: J-Link Base 9)

When debugging the coprocessor firmware, you can use the following methods:

5.5.1 Debug a non remoteproc firmware

1. Connect the debug probe to the board. (e.g. via PEB-EVAL-01)
2. Start the coprocessor via the debug probe. With west:

```
host:zephyrproject/zephyr$ west debug
```

Hint

The only thing *west debug* does is to start a JLinkGDBServer and GDB with the correct parameters for your target. If you don't want to use or can't use west, you can do this manually as well.

3. Load the firmware on the coprocessor via GDB.

```
(gdb) load
```

4. Set a breakpoint and start the firmware on the coprocessor via GDB.

```
(gdb) break main
(gdb) continue
```

5.5.2 Debugging a remoteproc firmware using GDB

Note

This is a workaround to debug a remoteproc firmware. It is neither the most convenient way to debug a processor nor is it recommended by NXP. Maybe there will be a better solution in the future but for now this is the only way found to debug a remoteproc firmware.

Prerequisites:

- Have the target booted up and connected to the host via debug usb and J-Link.
- Have the firmware in `/lib/firmware` on the target. For example *OpenAMP using resource table* (Make sure it is the same file you are debugging with GDB!)
- Have the resource table included in the firmware binary.
- Have the remoteproc device enabled in the devicetree.

- Have a serial console to the coprocessor. (e.g. via ttyUSB1)
- Have a shell of the application processor open (e.g. via SSH)

1. Start a debugserver with west:

```
host:zephyrproject/zephyr$ west debugserver --runner jlink --iface jtag
```

2. Start GDB with your firmware in a new terminal:

```
host:zephyrproject/zephyr$ gdb-multiarch build/zephyr/zephyr.elf -tui
```

3. Connect to the debugserver:

```
(gdb) target remote :2331
```

4. Reset the coprocessor and load the firmware:

```
(gdb) monitor reset
(gdb) load
```

5. Insert needed kernel modules on the target (e.g. rpmsg_tty.ko)

```
target:~$ modprobe rpmsg_tty
```

6. Start the firmware on the coprocessor via remoteproc:

```
target:~$ echo /lib/firmware/zephyr.elf > /sys/class/remoteproc/remoteproc0/firmware
target:~$ echo start > /sys/class/remoteproc/remoteproc0/state
```

7. The Linux shell freezes now because GDB is halting the coprocessor. Continue the execution in GDB:

```
(gdb) continue
```

8. Zephyr will not boot up and hang in a fault condition. This is expected. To overcome this issue, break execution with Ctrl+C, reset the coprocessor and continue again.

```
(gdb) monitor reset
(gdb) continue
```

9. The coprocessor should now boot up, and you can debug the firmware via GDB.

Hint

It is important to use JTAG as debug interface. Using SWD will reset and halt the whole SoC which will cause unexpected behavior.

5.5.3 GDB hints

Here are some useful GDB commands to debug the coprocessor firmware:

- `monitor reset`: Reset the coprocessor
- `monitor halt`: Halt the coprocessor
- `break main`: Set a breakpoint at the main function
- `break main.c:42`: Set a breakpoint at line 42 in main.c

- `watch *(unsigned short*)0x30a30010`: Set a watchpoint on a 16-bit memory address(e.g. some register)
- `print var`: Print the value of a variable in the current context
- `backtrace`: Print the current stack trace
- `continue`: Continue the execution
- `step`: Step into the next function

If the command doesn't get ambiguous, you can shorten the command. For example, you can use `b main` instead of `break main` or `c` instead of `continue`. This is useful if you have to type the command multiple times.

5.5.4 Debugging a remoteproc firmware using SEGGER Ozone

Prerequisites:

- Have the target booted up and connected to the host via debug USB and J-Link.
- Have the firmware in `/lib/firmware` on the target. For example *OpenAMP using resource table* (Make sure it is the same file you are debugging with Ozone!)
- Have the resource table included in the firmware binary.
- Have the remoteproc device enabled in the devicetree.
- Have a serial console to the coprocessor. (e.g. via ttyUSB1)
- Have a shell of the application processor open (e.g. via SSH)

SEGGER Ozone is a powerful graphical debugging tool that can be used to debug any kind of target with any kind of architecture. It makes it more easy to attach to a running program than GDB.

Here are the steps how to connect to a running program:

1. Start the target and load the firmware via remoteproc:

```
target:~$ echo /lib/firmware/zephyr.elf > /sys/class/remoteproc/remoteproc0/firmware
target:~$ echo start > /sys/class/remoteproc/remoteproc0/state
```

2. Start SEGGER Ozone and use the new project wizard
3. Select your target (for example MIMX8ML8_M7 for i.MX8MP), click next and select the connected J-Link debug probe.
4. Select the compiled elf file of your firmware and click next.
5. Select "Do not set" for initial PC and Stack Pointer to ensure that nothing is overwritten.
6. Click "Finish" to create the project.
7. Click on the small green arrow directly next to the "On/Off" button in the top left corner of the window. Click on "Attach to running program".
8. The target will halt, even though Ozone shows "CPU Running…"
9. To fix this behavior just restart the coprocessor via remoteproc on the target:

```
target:~$ echo stop > /sys/class/remoteproc/remoteproc0/state
target:~$ echo start > /sys/class/remoteproc/remoteproc0/state
```

10. The target should now boot up, and you can debug the firmware via Ozone.

You can use debugging with Ozone not just to debug the firmware, but also to debug the remoteproc framework itself. This can be useful if you want to find out why the coprocessor is not booting up or why the communication is not working or if you just want to get a deeper insight into the remoteproc framework.

CHAPTER

SIX

EXAMPLES AND RESOURCES

This section gives an overview of examples and resources that can be used to get started with a coprocessor.

The examples are focused on the NXP i.MX platform and Zephyr for now, but the principles can be applied to other platforms as well.

Resources:

- NXP AN5317 - Loading code to Coprocessor
- Zephyr IPC Samples

6.1 Hello World

The hello_world sample is a simple example Zephyr project, that prints "Hello World!" to the serial console.

6.1.1 Run the Sample

1. Make sure the devicetree overlay `imx8mp-phycore-rpmsg.dtbo` is activated, the BSP manual for your platform explains how to activate this.
2. Restart the target and execute in U-Boot:

```
u-boot=> run prepare_mcore
```

3. Save the environment in U-Boot in order to enable the m-core on every boot by default. Executing `saveenv` twice will save the environment to the redundant MMC partition as well.

```
u-boot=> saveenv
Saving Environment to MMC... Writing to MMC(1)... OK
u-boot=> saveenv
Saving Environment to MMC... Writing to redundant MMC(1)... OK
```

3. The target will now boot and you can build and flash the Zephyr application with:

```
host:zephyrproject/zephyr$ west build -b phyboard_pollux/mimx8ml8/m7 samples/hello_world -p
```

4. Zephyr should now boot with

```
target_m7:~$ *** Booting Zephyr OS build v3.7.0 ***
             Hello World! phyboard_pollux/mimx8ml8/m7
```

6.2 OpenAMP using resource table

The openamp_rsc_table sample "demonstrates how to use OpenAMP with Zephyr based on a resource table. It is designed to respond to [..]" the rpmsg client and rpmsg tty samples in the Linux Kernel. This sample demonstrates communication between Zephyr (coprocessor) and Linux (application processor) using OpenAMP. It creates the two RPMsg endpoints:

rpmsg-client-sample
: Demonstrates generic RPMsg message exchange (Ping-pong) between Zephyr and Linux.

rpmsg-tty
: A TTY service that virtualizes a serial connection at */dev/rpmsg-tty* in Linux, facilitating data exchange with Zephyr over this virtualized interface.

6.2.1 Prepare Linux

The example has been tested with the imx8mp and the BSP-Yocto-NXP-i.MX8MP-PD24.1.0. However, some modifications are necessary to be able to communicate in between Zephyr and Linux with RPMsg. The devicetree overlay that enables rpmsg has to be enabled. You can edit this line directly in bootenv.txt in the boot partition.

Listing 1: Changes in 'bootenv.txt'

```
+++ b/recipes-bsp/bootenv/phytec-bootenv/phyboard-pollux-imx8mp-3/bootenv.txt
@@ -1 +1 @@
-overlays=conf-imx8mp-phyboard-pollux-peb-av-10.dtbo
+overlays=conf-imx8mp-phyboard-pollux-peb-av-10.dtbo#conf-imx8mp-phycore-rpmsg.dtbo
```

Listing 2: Changes in the devicetree overlay 'imx8mp-phycore-rpmsg.dtbo'

```
+++ b/arch/arm64/boot/dts/freescale/imx8mp-phycore-rpmsg.dtso
@@ -14,11 +14,11 @@
        core-m7 {
                compatible = "fsl,imx8mn-cm7";
                clocks = <&clk IMX8MP_CLK_M7_DIV>;
-               mboxes = <&mu 0 1>,
-                        <&mu 1 1>,
-                        <&mu 3 1>;
+               mboxes = <&mu 0 0>,
+                        <&mu 1 0>,
+                        <&mu 3 0>;
                mbox-names = "tx", "rx", "rxdb";
-               memory-region = <&vdevbuffer>, <&vdev0vring0>, <&vdev0vring1>, <&rsc_table>;
+               memory-region = <&vdevbuffer>, <&vdev0vring0>, <&vdev0vring1>;
        };

        reserved-memory {
@@ -27,29 +27,31 @@ reserved-memory {
                #size-cells = <2>;

                vdev0vring0: vdev0vring0@55000000 {
-                       no-map;
+                       compatible = "shared-dma-pool";
```

(continues on next page)

(continued from previous page)

```
                        reg = <0 0x55000000 0 0x8000>;
+                       no-map;
                };

                vdev0vring1: vdev0vring1@55008000 {
-                       no-map;
+                       compatible = "shared-dma-pool";
                        reg = <0 0x55008000 0 0x8000>;
+                       no-map;
                };
```

6.2.2 Prepare Zephyr

The sample needs some board specific settings and a devicetree overlay for the phyBOARD Pollux. This will be upstreamed soon and maybe it is possible to make the Zephyr sample fully generic.

You can see a branch with the required changes here.

6.2.3 Run the Sample

1. Make sure the devicetree overlay `imx8mp-phycore-rpmsg.dtbo` is activated, the BSP manual for your platform explains how to activate this.
2. Restart the target and execute in U-Boot:

   ```
   u-boot=> run prepare_mcore
   ```

3. Build Zephyr and copy the firmware to `/lib/firmware` on the target:

   ```
   host:zephyrproject/zephyr$ west build -b phyboard_pollux/mimx8ml8/m7 samples/subsys/ipc/
   ↪openamp_rsc_table/ -p
   ```

4. Start the Zephyr application with remoteproc:

   ```
   root@phyboard-pollux-imx8mp-3:~# echo stop > /sys/class/remoteproc/remoteproc0/state
   root@phyboard-pollux-imx8mp-3:~# echo /lib/firmware/zephyr_openamp_rsc_table.elf > /sys/
   ↪class/remoteproc/remoteproc0/firmware
   root@phyboard-pollux-imx8mp-3:~# echo start > /sys/class/remoteproc/remoteproc0/state
   ```

4. Zephyr should now boot now. The kernel module `rpmsg_client_sample` should load automatically and respond to the running m-core.

   ```
   target_m7:~$ *** Booting Zephyr OS build v4.0.0-870-g6d87bd65aebf ***
                I: Starting application threads!
                I: OpenAMP[remote] Linux responder demo started
                D: mailbox_notify: msg received
                I: OpenAMP[remote] Linux sample client responder started
                D: mailbox_notify: msg received
                I: OpenAMP[remote] Linux TTY responder started
                D: mailbox_notify: msg received

                : platform_ipm_callback: msg received from mb 0
                I: [Linux sample client] incoming msg 1: hello world!
   ```

(continues on next page)

(continued from previous page)

```
            D: mailbox_notify: msg received
            D: platform_ipm_callback: msg received from mb 0
            I: [Linux sample client] incoming msg 1: hello world!
            D: mailbox_notify: msg received
```

5. If the the kernel Module does not load automatically, you can manually load it:

```
target:~$ modprobe rpmsg_client_sample
target:~$ dmesg | tail                          # Check module messages
target:~$ modprobe -u rpmsg_client_sample       # Unload Kernel module
```

Serial Communication

Once the demo is running, it opens two serial devices (`/dev/ttyRPMSG0`, `/dev/ttyRPMSG1`), one to send/receive any messages to Zephyr and one for the Zephyr shell backend.

```
# Open the tty channel
root@phyboard-pollux-imx8mp-3:~# cat /dev/ttyRPMSG1 &
[3] 504
root@phyboard-pollux-imx8mp-3:~# echo "Hello Zephyr" >/dev/ttyRPMSG1
root@phyboard-pollux-imx8mp-3:~# TTY 0x0402: Hello Zephyr
TTY 0x0402: Hello Zephyr

# Open the Zephyr shell with micocom
root@phyboard-pollux-imx8mp-3:~# microcom /dev/ttyRPMSG0

clear    device   devmem   help     history  kernel   rem      resize
retval   shell
ipc:~$
```

Note

Remoteproc ensures to register the resource table and the RPMsg service. Running firmware via debug probe is not possible when using RPMsg.

Warning

Remoteproc only reads firmware files from the `/lib/firmware` directory! If you try to load a binary from another location errors will occur!

6.2.4 Console Output Linux

```
# Stop a running m-core
root@phyboard-pollux-imx8mp-3:~# echo stop > /sys/class/remoteproc/remoteproc0/state
[18375.572034] imx-rproc core-m7: Not in wfi, force stopped
[18375.577423] remoteproc remoteproc0: stopped remote processor imx-rproc

# Load the firmware
root@phyboard-pollux-imx8mp-3:~# echo /lib/firmware/zephyr_openamp_rsc_table.elf > /sys/class/
↪remoteproc/remoteproc0/firmware
```

(continues on next page)

(continued from previous page)

```
# Start the m-core
root@phyboard-pollux-imx8mp-3:~# echo start > /sys/class/remoteproc/remoteproc0/state
[18402.215721] remoteproc remoteproc0: powering up imx-rproc
[18402.221215] remoteproc remoteproc0: Direct firmware load for /lib/firmware/zephyr.elf failed␣
↪with error -2
[18402.230900] remoteproc remoteproc0: Falling back to sysfs fallback for: /lib/firmware/zephyr.
↪elf
[18402.243066] remoteproc remoteproc0: Booting fw image /lib/firmware/zephyr.elf, size 1402364
[18402.252283] rproc-virtio rproc-virtio.3.auto: assigned reserved memory node␣
↪vdevbuffer@55400000
[18402.262788] virtio_rpmsg_bus virtio0: rpmsg host is online
[18402.268484] rproc-virtio rproc-virtio.3.auto: registered virtio0 (type 7)
[18402.275367] virtio_rpmsg_bus virtio0: creating channel rpmsg-tty addr 0x400
[18402.276433] remoteproc remoteproc0: remote processor imx-rproc is now up
[18402.282735] virtio_rpmsg_bus virtio0: creating channel rpmsg-client-sample addr 0x401
[18402.297625] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1025: new channel: 0x401 ->␣
↪0x401!
[18402.308941] virtio_rpmsg_bus virtio0: creating channel rpmsg-tty addr 0x402
[18402.320915] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1025: incoming msg 1 (src:␣
↪0x401)
[18402.341810] rpmsg_client_sample virtio0.rpmsg-client-sample.-1.1025: incoming msg 2 (src:␣
↪0x401)
```

6.2.5 Debugging

Listing 3: Print resource table in Linux

```
root@phyboard-pollux-imx8mp-3:~# cat /sys/kernel/debug/remoteproc/remoteproc0/resource_table
Entry 0 is of type vdev
  ID 7
  Notify ID 0
  Device features 0x1
  Guest features 0x1
  Config length 0x0
  Status 0x7
  Number of vrings 2
  Reserved (should be zero) [0][0]

  Vring 0
    Device Address 0x55000000
    Alignment 16
    Number of buffers 8
    Notify ID 0
    Physical Address 0x0

  Vring 1
    Device Address 0x55008000
    Alignment 16
    Number of buffers 8
    Notify ID 1
    Physical Address 0x0
```

Listing 4: Print related memory areas in Linux:

```
root@phyboard-pollux-imx8mp-3:~# cat /sys/kernel/debug/remoteproc/remoteproc0/resource_table
Entry 0 is of type vdev
  ID 7
  Notify ID 0
  Device features 0x1
  Guest features 0x1
  Config length 0x0
  Status 0x7
  Number of vrings 2
  Reserved (should be zero) [0][0]

  Vring 0
    Device Address 0x55000000
    Alignment 16
    Number of buffers 8
    Notify ID 0
    Physical Address 0x0

  Vring 1
    Device Address 0x55008000
    Alignment 16
    Number of buffers 8
    Notify ID 1
    Physical Address 0x0
```

6.3 Other Examples

The following examples exist in Zephyr, however, they are specific to SoCs that have multiple instances of Zephyr running in the same SoC. They are partly related to Zephyrs ipc_service and not suitable for communication with Linux.

OpenAMP Sample

> sample builds different images for two targets running Zephyr. Both targets setup virtqueue and virtio and communicate with each other via RPMsg. This sample is mainly used to evaluate SoCs with two Cortex M devices and can not be used with Linux.

openamp-system-reference

> Several samples for both platforms, Linux and Zephyr that demonstrate different aspects of OpenAMP.

Samples in ipc_service/

> Examples related to Zephyr ipc_service subsystem. Note that not all of those examples may be applicable to heterogeneous systems with one core running Linux and the other Zephyr.

CHAPTER

SEVEN

CURRENT PROBLEMS

This section lists current problems that need work.

1. Shell not working in Zephyr for Linux SoCs.

   There may be a problem with interrupts and nxp deactivated the shell for the imx8qm boards. https://github.com/zephyrproject-rtos/zephyr/pull/79428