AN5317

Loading Code on Cortex-M from U-Boot/Linux for the i.MX Asymmetric Multi-Processing Application Processors

Rev. 3.0 — 7 October 2024 Application note

Document information

Information	Content
Keywords	AN5317, Cortex-M, U-Boot, Linux, i.MX Asymmetric Multi-Processing Application
Abstract	This application note shows how to load code into Cortex-M from the software running on Cortex-A cores.

1 Introduction

There is a growing number of embedded use cases that require concurrent execution of isolated and secure software environments. Multiple software execution environments are useful for:

- Off-loading tasks and improving real-time performance
- Increasing system integrity and security
- Optimizing power consumption

The i.MX 8M, i.MX 8, i.MX 8X, i.MX 7D/S/UL, and i.MX 6SoloX SoC families offer Asymmetric Multi-Processing (AMP) solutions with both Arm Cortex-A processors and Cortex-M4 microcontroller on a single SoC. The cores can be partitioned into two respective processing domains that can be programmed to run a different OS to cater for the real-time latency and application processing requirements.

In some applications, it is very useful to have the Arm Cortex-A processors reload code into the Cortex-M4 microcontroller. This application note shows how to reload code on Cortex-M from the Linux shell/U-Boot using the Arm Cortex-A processor. The same method can be used for any other OS or bare metal implementation.

This application note shows how to load code into Cortex-M from the software running on Cortex-A cores.

The following table lists some NXP SoCs that can be used in the AMP configuration and the software support for Cortex-M loading from a side in U-Boot and Linux starting with NXP Linux BSP versions ≥lf-6.6.23-2.0.0.

Table 1. Examples of NXP SoCs that can be used in AMP configuration

SoC name	U-Boot - Cortex-M load support	RemoteProc support
i.MX 8QM	√	√
i.MX 8QXP	√	√
i.MX 8MP	√	√
i.MX 8MQ	√	√
i.MX 8MM	√	√
i.MX 8MN	√	√
i.MX 7D/S	√	√
i.MX 6SoloX	√	√

2 Overview of i.MX 8QM/QXP implementations

The i.MX 8QM application processor provides a powerful fully coherent core complex based on a dual (2x) Cortex-A72 cluster for use cases requiring high computing performances, a quad (4x) Cortex-A53 cluster running most of the use cases at a lower power consumption and two clusters, each with one Cortex-M4 for real-time performance.

The i.MX 8QXP application processor provides a quad Arm Cortex-A35 cluster providing full 64-bit ARMv8-A support while maintaining seamless backward compatibility with 32-bit ARMv7-A software and a single Arm Cortex-M4.

The current U-Boot source code for both i.MX 8QM and i.MX 8QXP application processors provides a *bootaux* and a *loadm4image_<coreid>* command that helps with loading the code to the Cortex-M4 core and bringing it up. For example, loading the *sensor_demo.bin* file to location 0x80280000 and booting the Cortex-M4_0 core with the image loaded can be done by running the following command: *"run m4boot_0"* whose implementation is "*run loadm4image_0; dcache flush; bootaux ${loadaddr} 0*", where *loadm4image_0* is *"fatload mmc ${mmcdev}:${mmcpart} ${loadaddr} ${m4_0_image}"* with *loadaddr=0x80280000* and *m4_0_image= sensor_demo.bin.*

This U-Boot feature is useful to bring the Cortex-M4 core up as soon as possible after the boot up or power-on reset, every time a user wants to modify the application. i.MX 8QM/iMX 8QXP introduces the System Controller Firmware (SCFW) for interaction with hardware. Controlling the Cortex-M4 from Linux is easier, enabling users to start, stop, and reload it. This implementation does not allow users to reload another application while a task is already running on the Cortex-M4 core. This application note describes the interaction with Cortex-M4 cores from the Linux shell via SCFW API.

3 Overview of i.MX 8M family implementations

The i.MX 8MQ/i.MX 8MM application processors offer a quad Arm Cortex-A53 cluster providing full 64-bit ARMv8-A support and also a single Arm Cortex-M4 processor.

The i.MX 8MP/i.MX 8MN application processors provide a quad Arm Cortex-A53 cluster and a single Arm Cortex-M7 processor.

The current U-Boot source code for both i.MX 8MM and i.MX 8MN application processors provides *bootaux* and *fatload* commands that help users with loading the code into the Cortex-M4/Cortex-M7 core and bringing it up. For example, "*fatload mmc 0:1 0x7e0000 sensor_demo.bin*" loads the *sensor_demo.bin* file to location 0x7e0000 and "*bootaux 0x7e00000*" boots the Cortex-M4/Cortex-M7 core with the image loaded at *0x007e_0000*.

4 Overview of i.MX 7Dual/7Solo and i.MX 6SoloX implementations

There are many similarities between the i.MX 7Dual/7Solo and i.MX 6SoloX application processors in terms of where the TCM_U and TCM_L (Tightly Coupled Memory - Upper/Lower) memories are located. However, the boot vector and the specific register to issue the platform reset and reset the Cortex-M4 are different on the two application processors. The bit locations inside the registers for both i.MX 7Dual/7Solo and i.MX 6SoloX application processors are also different.

The i.MX 7Dual/7Solo application processor provides a multicore solution of Arm Cortex-A7 cores (dual or single) and a single Arm Cortex-M4 core.

The i.MX 6SoloX application processor provides a single Arm Cortex-A9 and a single Arm Cortex-M4. The Arm Cortex-A7 on i.MX 7Dual/7Solo and the Arm Cortex-A9 on i.MX 6SoloX are both capable of booting using different interfaces and they are also responsible for bringing up the different interfaces of the chip. It is the responsibility of Cortex-A7 on i.MX 7Dual/7Solo application processors and Cortex-A9 on i.MX 6SoloX application processors to enable the Cortex-M4 core.

The current U-Boot source code for both i.MX 7Dual/7Solo and i.MX 6SoloX application processors provides *bootaux* and *fatload* commands that help users with loading the code to the Cortex-M4 core and bringing it up. For example, "*fatload mmc 0:1 0x7f8000 sensor_demo.bin*" loads the *sensor_demo.bin* file to location 0x7f8000 and "*bootaux 0x7f80000*" boots the Cortex-M4 core with the image loaded at *0x007f_8000*.

Although this feature is useful to bring the Cortex-M4 core up as soon as possible after the boot up or power-on reset (with the existing implementation every time a user wants to modify the application), the U-Boot must be reconfigured. This implementation does not allow users to reload another application while a task is already running on the Cortex-M4 core. This application note describes the registers that are required to be programmed to reload the application from the Linux shell.

For this application note, we assume that the Cortex-M4 core is compiled to execute from the TCM_L and TCM_U on the chip memories. We also assume that the Cortex-M4 clock is enabled. Users using U-Boot can enable the clock with the *bootaux* command by loading a primary image, which, on the Cortex-M4 side, tells the Linux kernel not to disable the Cortex-M4 clock when the Linux kernel takes over. If you do not run the U-Boot and want to enable the clock of the Cortex-M4, see the respective clock chapters in the *i.MX 7Dual Applications Processor Reference Manual* (document IMX7DRM) and the *i.MX 6SoloX Applications Processor Reference Manual* (document IMX6SXRM).

5 Reloading code on i.MX 8QM/QXP

This section describes how to reload code on i.MX 8QM/QXP.

5.1 On-chip memory view from each Arm core on i.MX 8QM/8QXP

The memory view of different peripherals is different between the Cortex-A and Cortex-M4 sides. Table 2 shows only the memory areas relevant for this application note. For more details, see the Memory Map chapter in the *i.MX 8QM/QXP Applications Processor Reference Manual*.

Note: *The TCM memory can be accessed from the A cores using the Cortex M4 platform-specific areas from the system memory map. The TCM memory is mapped in the same address ranges as those that Cortex-M4 cores see as their TCM memory.*

i.MX 8QM has two Cortex-M4 cores, each with their own cluster. i.MX 8QXP has one Cortex-M4 core. The Cortex-M4_0 from both platforms has the memory map even for the TCM memory.

Table 2. Start and end addresses of different memories from Cortex-M4 side on i.MX

Peripheral	Start address Cortex-M4_0	End address Cortex-M4_0	Start address Cortex-M4_1	End address Cortex-M4_1	Size
TCM_L	0x34FE_0000	0x34FF_FFFF	0x38FE_0000	0x38FF_FFFF	128 kB
TCM_H	0x3500_0000	0x3501_FFFF	0x3900_0000	0x3901_FFFF	128 kB

5.2 Detailed procedure

To reload the code on the Cortex-M4 core using the Cortex-A processor on i.MX 8QM/QXP, follow the steps below if M4 resources belong to the Cortex-A partition:

1. Open an IPC channel to communicate with the SCFW that runs on the SCU using the **sc_ipc_open (sc_ipc_t _ ipc, sc_ipc_id_t id)** function.
2. Issue a software platform stop for the Cortex-M4 core using the **sc_pm_cpu_start (sc_ipc_t ipc, sc_rsrc_t resource, bool enable, sc_faddr_t address)** function.
3. Issue a software platform power off for the Cortex-M4 core using the **sc_pm_set_resource_power_mode (sc_ipc_t ipc, sc_rsrc_t resource, sc_pm_power_mode_t mode)** function. Power on the Cortex-M4 core using the above function. This step ensures that the TCM_L memory is reset.
4. Load the code for the Cortex-M4 processor into the TCM_L memory. For this application, we assume that the Cortex-M4 code is linked to the TCM_L memory. Program the FreeRTOS binary file to the TCM_L address.
5. When the image is loaded, start the Cortex-M4 core using the **sc_pm_cpu_start (sc_ipc_t ipc, sc_rsrc_t resource, bool enable, sc_faddr_t address)** function.

6 Reloading code on i.MX 8M family

This section describes how to reload code on the i.MX 8M family.

6.1 On-chip memory view from each Arm core on i.MX 8M SoC

The memory view of different peripherals is different between the Cortex-A53 and Cortex-M4/M7 sides. Table 3 and Table 4 show only the memory areas relevant for this application note. For more details, see the Memory Map chapter in the *i.MX 8MM/8MN Applications Processor Reference Manual*.

Note: *The TCM memory can be accessed from the A cores using the Cortex-M platform-specific areas from the system memory map. The TCM memory is mapped in the same address ranges as those that Cortex-M cores see as their TCM memory.*

Table 3. Start and end addresses of different memories from Cortex-M4 side on i.MX8MQ and i.MX8MM

Peripheral	Start address Cortex-A53	End address Cortex-A53	Start address Cortex-M4	End address Cortex-M4	Size
TCM_L	0x007E_0000	0x007F_FFFF	0x1FFE_0000	0x1FFF_FFFF	128 kB
TCM_H	0x0080_0000	0x0081_FFFF	0x2000_0000	0x2001_FFFF	128 kB

Table 4. Start and end addresses of different memories from Cortex-M7 side on i.MX 8MN and i.MX 8MP

Peripheral	Start address Cortex-A53	End address Cortex-A53	Start address Cortex-M7	End address Cortex-M7	Size
ITCM	0x007E_0000	0x007F_FFFF	0x0000_0000	0x0001_FFFF	128 kB
DTCM	0x0080_0000	0x0081_FFFF	0x2000_0000	0x2001_FFFF	128 kB

6.2 Detailed procedure

To reload the code on the Cortex-M4 core using the Cortex-A53 processor on i.MX 8MM, follow the steps below:

1. Issue a software platform reset by setting up **SW_M4P_RST** (Bit 2) in the **SRC_M4RCR** (SRC_M4RCR[2]) register. Issuing a platform reset resets the Cortex-M4 cores and their associated memories. The address of the SRC_M4RCR register is 0x3039_000C for the i.MX 8MM SoC.
2. Load the code for the Cortex-M4 processor into the TCM_L memory. For this application, we assume that the Cortex-M4 code is compiled to execute from the TCM_L memory. In Table 3, the TCM_L address for the Cortex-A53 side is 0x007E_0000. Program the FreeRTOS binary file to that address.
3. When the file is loaded, the next step is to set **ENABLE_M4** (Bit 3) in the **SRC_M4RCR** (SRC_M4RCR[3]) register. Because *bootaux* already booted a primary image, this bit should be 1. Performing a platform reset using **SW_M4P_RST** (Bit 2) in the **SRC_M4RCR** (SRC_M4RCR[2]) register does not clear this bit. The last step is to set the **SW_M4C_RST** (Bit 1) in the **SRC_M4RCR** (SRC_M4RCR[1]) register, which boots the new code in the Cortex-M4 processor.

To reload the code on the Cortex-M7 core using the Cortex-A53 processor on i.MX 8MN, follow the steps below:

1. Issue a software core reset by setting up **SW_M7C_RST** (Bit 1) in the **SRC_M7RCR** (SRC_M7RCR[2]) register. Issuing a core reset resets the Cortex-M7 core and the associated memories. The address of the SRC_M7RCR register is 0x3039_000C for the i.MX 8MN SoC.
2. Load the code for the Cortex-M7 processor into the ITCM memory. For this application, we assume that the Cortex-M7 code is compiled to execute from the ITCM memory. In Table 4, the ITCM address for the Cortex-A53 side is 0x007E_0000. Program the binary file generated by FreeRTOS to that address.
3. When the file is loaded, the next step is to set the **ENABLE_M7** (Bit 3) in the **SRC_M7RCR** (SRC_M7RCR[3]) register. Because *bootaux* already booted a primary image, this bit should be 1. Performing a core reset using **SW_M7C_RST** (Bit 1) in the **SRC_M7RCR** (SRC_M7RCR[1]) register does not clear this bit. The last step is to set the **SW_M7C_RST** (Bit 1) in the **SRC_M7RCR** (SRC_M7RCR[1]) register, which boots the new code in the Cortex-M4 processor.

7 Reloading code on i.MX 7Dual/7Solo

This section describes how to reload code on i.MX 7Dual/7Solo.

7.1 On-chip memory view from each Arm core on i.MX 7Dual/7Solo

The memory view of different peripherals is different between the Cortex-A7 and Cortex-M4 sides. Table 5 shows only the memory areas relevant for this application note. For more details, see the Memory Map chapter in the *i.MX 7Dual Applications Processor Reference Manual* (document IMX7DRM).

Note: *On i.MX 7Dual/7Solo, the boot vector for the Cortex-M4 core is located at the start of the OCRAM_S (On-Chip RAM - Secure), whose address is 0x0018_0000 for Cortex-A7.*

Table 5. Start and end addresses of different memories for Cortex-A7 and Cortex-M4 sides

Peripheral	Start address Cortex-A7	End address Cortex-A7	Start address Cortex-M4 side	End address Cortex-M4 side	Size
OCRAM_S	0x0018_0000	0x0018_7FFF	0x2018_0000	0x2018_7FFF	32 kB
TCM_L	0x007F_8000	0x007F_7FFF	0x1FFF_8000	0x1FFF_FFFF	32 kB
TCM_H	0x0080_0000	0x0080_7FFF	0x2000_0000	0x2000_7FFF	32 kB

7.2 Detailed procedure

To reload the code on the Cortex-M4 core using the Cortex-A7 processor on i.MX 7Dual/7Solo, follow the steps below:

1. Issue a software platform reset by setting up **SW_M4P_RST** (Bit 2) in the **SRC_M4RCR** (SRC_M4RCR[2]) register. Issuing a platform reset resets the Cortex-M4 cores and associated memories. The address of the SRC_M4RCR register is 0x3039_000C for i.MX 7Dual/7Solo SoC.
2. Load the code for the Cortex-M4 processor into the TCM_L memory. For this application, we assume that the Cortex-M4 code is compiled to execute from the TCM_L memory. In Table 5, the TCM_L address from the Cortex-A7 side is 0x007F_8000. Program the FreeRTOS binary file to that address.
3. When the file is loaded, set up the Stack and PC pointer in the OCRAM_S memory. After reset, the processor uses the OCRAM_S start address (0x0018_0000) as the first instruction. For this implementation, the stack value is the first four bytes found in the binary file generated for the Cortex-M4 processor using FreeRTOS source. The PC value is also 4 bytes long and located at an offset of 0x4 in the binary file. This PC value is written to the OCRAM_S base address plus 4, which is (0x0018_0004) for this platform. Table 6 further clarifies the Stack and PC addresses.

   Table 6. Boot vectors' location for Cortex-M4 core

Boot vectors	OCRAM_S location for boot vectors	Location of boot vectors in binary file
Stack pointer	0x0018_0000	First 4 bytes
Program counter	0x0018_0004	4 bytes after first 4 bytes

4. When the start-up address in the OCRAM_S is adjusted according to the binary file, the file is loaded into the memory. The next step is to set the **ENABLE_M4** (Bit 3) in the **SRC_M4RCR** (SRC_M4RCR[3]) register. Because *bootaux* already booted a primary image, this bit should be 1. Performing a platform reset using **SW_M4P_RST** (Bit 2) in the **SRC_M4RCR** (SRC_M4RCR[2]) register does not clear this bit. The last step is to set the **SW_M4C_RST** (Bit 1) in the **SRC_M4RCR** (SRC_M4RCR[1]) register, which boots the new code on the Cortex-M4 processor.
5. Repeat steps 1-3 to reload a new image. More details about the **SRC_M4RCR** register are shown in Figure 1.

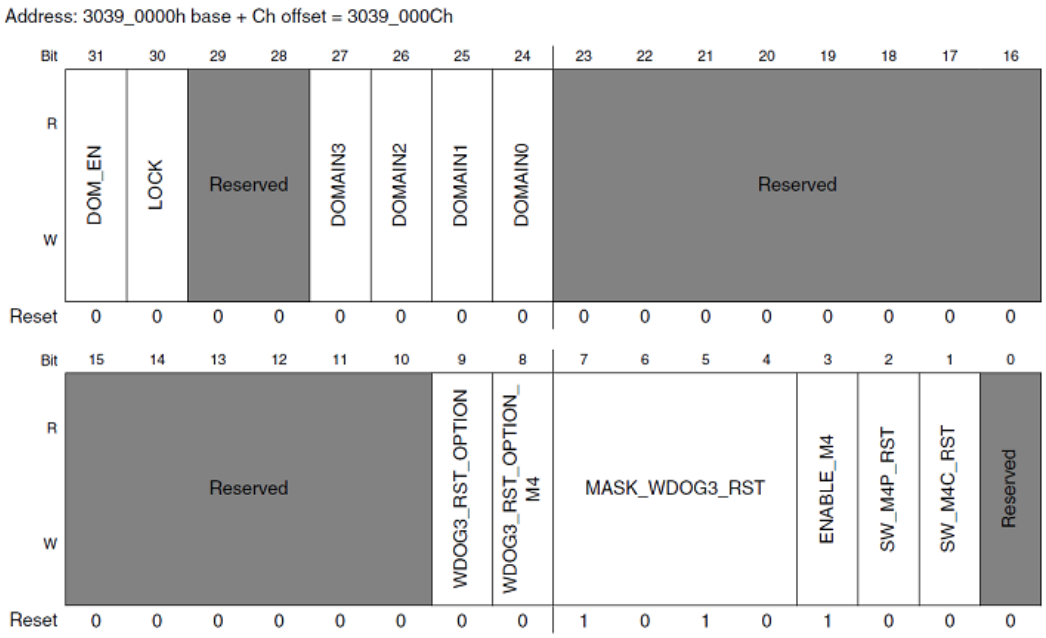

Figure 1. Cortex-M4 reset control register (SRC_M4RCR)

7.3 Steps for reloading code on i.MX 7Dual/7Solo

The steps to reload code on i.MX 7Dual/7Solo are as follows:

1. Issue a platform reset by setting the SRC_M4RCR[2] bit in the SRC_MRCR register.
2. Wait for SRC_M4RCR[2] to be cleared.
3. Set the stack pointer to the first 4 bytes of the binary file.
4. Set the PC pointer to the next 4 bytes after the first 4 bytes of the binary file.
5. Load the binary file starting at address 0x007F_8000.
6. Reset the Cortex-M4 microcontroller by setting the SRC_M4RCR[1] bit in the SRC_M4RCR register.

8 Reloading code on i.MX 6SoloX

This section describes how to reload code on i.MX 6SoloX.

8.1 On-chip memory view from each Arm core on the i.MX 6SoloX

The memory view of different peripherals is different between the Cortex-A9 and Cortex-M4 sides. Table 7 shows only the memory areas relevant for this application note. For more details, see the Memory Map chapter in the *i.MX 6SoloX Applications Processor Reference Manual* (document IMX6SXRM).

Table 7. Start and end addresses of different memories from Cortex-A9 and Cortex-M4 side

Peripheral	Start address Cortex-A9	End address Cortex-A9	Start address Cortex-M4 side	End address Cortex-M4 side	Size
TCM_L	0x007F_8000	0x007F_7FFF	0x1FFF_8000	0x1FFF_FFFF	32 kB
TCM_U	0x0080_0000	0x0080_7FFF	0x2000_0000	0x2000_7FFF	32 kB

Note: *The boot vector for the Cortex-M4 core is located at the start of the TCM_L, whose address is 0x007F_8000 from the Cortex-A9 core. This is a location different from that on the i.MX 7Dual/7Solo.*

8.2 Detailed procedure

To reload the code on the Cortex-M4 core using the Cortex-A9 processor on i.MX 6SoloX, follow the steps listed below:

1. Issue a software platform reset by setting up **M4P_RST** (Bit 12) in the **SRC_SCR** (SRC_M4RCR[12]) register. Issuing a platform reset resets the Cortex-M4 cores and associated memories. The address of the **SRC_SCR** register is 0x020D_8000 for i.MX 6SoloX.
2. Load the code for the Cortex-M4 processor into the TCM_L memory. For this application, we assume that the M4 code is compiled to execute from the TCM_L memory. Program the binary file generated by FreeRTOS to the TCM_L address from the Cortex-A9 side, which is 0x007F_8000, as listed in Table 7.
3. When the file is loaded, set up the stack and PC pointers in the TCM_L memory. After a reset, the processor uses the TCM_L start address (0x007F_8000) as the first instruction. For this implementation, the stack value is the first 4 bytes found in the binary file generated for the Cortex-M4 processor using the FreeRTOS source. The PC value is also 4 bytes long and located at an offset of 0x4 in the binary file. This PC value is written to the TCM_L base address plus four, which is (0x007F_8004). Table 8 shows the stack and PC addresses.

Table 8. Boot vectors location for Cortex-M4 core

Boot vectors	TCM_L location for boot vectors	Location of boot vectors in binary file
Stack pointer	0x007F_8000	First 4 bytes
Program counter	0x007F_8004	4 bytes after first 4 bytes

4. When the startup address in the TCM_L is modified according to the binary file and loaded into the memory, ensure that the **ENABLE_M4** (Bit 22) in the **SRC_SCR** (SRC_SCR[22]) register is set to 1. Because the *bootaux* already booted a primary image, this bit should be 1. Performing a platform reset using **M4P_RST** (Bit 12) in the **SRC_SCR** (SRC_SCR[12]) register does not clear this bit. The last step is to set the **M4C_RST** (Bit 3) in the **SRC_SCR** (SRC_SCR[3]) register, which boots the new code on the Cortex-M4 processor.
5. Repeat steps 1-3 to reload a new image. More details about the **SRC_SCR** register are in Figure 2.

The Reset control register (SCR), contains bits that control operation of the reset controller.

Address: 20D_8000h base + 0h offset = 20D_8000h

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	mask_wdog3_rst				mix_rst_strch		dbg_rst_msk_pg	wdog3_rst_optn	wdog3_rst_optn_m4	m4_enable	cores_dbg_rst	mask_tempsense_reset			core0_dbg_rst	0
W																
Reset	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0	0	core0_rst	m4p_rst	eim_rst	mask_wdog_rst			warm_rst_bypass_count			m4c_non_sclr_rst	m4c_rst	Reserved	sw_gpu_rst	warm_reset_enable
W																
Reset	0	0	0	1	0	1	0	1	0	0	1	1	1	0	0	1

Figure 2. SRC control register (SRC_SCR)

8.3 Steps for reloading code on i.MX 6SoloX

The steps to reload code on i.MX 6SoloX are as follows:

1. Issue a platform reset by setting the SRC_SCR[12] bit in the SRC_SCR register.
2. Wait for the SRC_SCR[12] bit to be cleared by the hardware.
3. Set the stack pointer to the first 4 bytes of the binary file.
4. Set the PC pointer to the next 4 bytes after the first 4 bytes of the binary file.
5. Load the binary file starting at address 0x007F_8000.
6. Reset the Cortex-M4 by setting the SRC_SCR[3] bit in the SRC_SCR register.

9 Linux Remote Processor (rproc) framework

Most modern SoCs are heterogenous platforms presenting Asymmetric Multiprocessing (AMP) configuration with different types of processors, which allows to run various instances of operating system (like Linux) in parallel with a real-time OS.

For example, i.MX 8MP presents a cluster of quad Cortex-A53 cores and a cluster with a Cortex-M7 core. The quad Cortex-A53 cluster usually runs Linux in SMP configuration and the Cortex-M7 core may run an RTOS.

The Remote Processor (rproc) framework is a Linux community effort to introduce the possibility to control (power on, load firmware, power off) the remote processors abstracting the hardware differences in the same time on AMP SoCs. It offers monitoring and debug services for the remote coprocessor.

Later versions of the Linux kernel (≥ 5.x) implement the rproc framework in the "remote proc" section of the Linux kernel repository drivers/remoteproc. The Linux kernel community implemented the framework abstractization between the user interaction (other Linux kernel modules, sysfs, user space) and the hardware to provide a uniform API, which can be used in a similar way for all platforms that support Linux rproc.

For the interaction with hardware, each SoC support must implement *rproc_ops* callbacks. The following code snippet shows the operations from *rproc_ops*.

```
"/**
 * struct rproc_ops - platform-specific device handlers
 * @prepare: prepare device for code loading
 * @unprepare: unprepare device after stop
 * @start: power on the device and boot it
 * @stop: power off the device
 * @attach: attach to a device that his already powered up
 * @detach: detach from a device, leaving it powered up
 * @kick: kick a virtqueue (virtqueue id given as a parameter)
 * @da_to_va: optional platform hook to perform address translations
 * @parse_fw: parse firmware to extract information (e.g. resource table)
 * @handle_rsc: optional platform hook to handle vendor resources. Should return
 *  RSC_HANDLED if resource was handled, RSC_IGNORED if not handled
 *  and a negative value on error
 * @find_loaded_rsc_table: find the loaded resource table from firmware image
 * @get_loaded_rsc_table: get resource table installed in memory
 *     by external entity
 * @load:  load firmware to memory, where the remote processor
 *   expects to find it
 * @sanity_check: sanity check the fw image
 * @get_boot_addr: get boot address to entry point specified in firmware
 * @panic: optional callback to react to system panic, core will delay
 *  panic at least the returned number of milliseconds
 * @coredump:   collect firmware dump after the subsystem is shutdown
 */

struct rproc_ops {
 int (*prepare)(struct rproc *rproc);
 int (*unprepare)(struct rproc *rproc);
 int (*start)(struct rproc *rproc);
 int (*stop)(struct rproc *rproc);
 int (*attach)(struct rproc *rproc);
 int (*detach)(struct rproc *rproc);
 void (*kick)(struct rproc *rproc, int vqid);
 void * (*da_to_va)(struct rproc *rproc, u64 da, size_t len, bool *is_iomem);
 int (*parse_fw)(struct rproc *rproc, const struct firmware *fw);
 int (*handle_rsc)(struct rproc *rproc, u32 rsc_type, void *rsc,
     int offset, int avail);
 struct resource_table *(*find_loaded_rsc_table)(
    struct rproc *rproc, const struct firmware *fw);
 struct resource_table *(*get_loaded_rsc_table)(
    struct rproc *rproc, size_t *size);
 int (*load)(struct rproc *rproc, const struct firmware *fw);
 int (*sanity_check)(struct rproc *rproc, const struct firmware *fw);
 u64 (*get_boot_addr)(struct rproc *rproc, const struct firmware *fw);
 unsigned long (*panic)(struct rproc *rproc);
 void (*coredump)(struct rproc *rproc);
};
"
```

rproc Linux Documentation describes each API and their main functionality, which can be a useful resource when you start implementing the SoC support from scratch.

9.1 i.MX Linux rproc support

NXP Linux BSP provides support for i.MX Linux rproc on the following platforms: i.MX 8MP, i.MX 8MQ, i.MX 8MM, i.MX 8MN, i.MX 8QM, i.MX 8QXP, i.MX 7D, i.MX 7UL, and i.MX 6SX. The implementation is realized in imx_rproc.c. The supported platforms can be also identified in the code by checking compatible strings from the *imx_rproc_of_match* structure.

Rproc implements the callback from *rproc_ops*, following the recommendations listed in previous chapters. For example, the i.MX 8/i.MX 8X rproc uses the SCFW API to start/stop the M4 core. The i.MX 8M rproc programs the SRC registers directly or via ATF to start/stop the M4/M7 cores.

The *imx_rproc* implementation sets the memory map for each supported platform inside to know which memory areas are allowed for the Cortex-M to contain code and data and those limits are checked at runtime when the ELF is parsed and its sections will be copied into the targeted memories.

For instance, Figure 3 shows how the memory map for i.MX 8MN / i.MX 8MP is defined in "imx_rproc":

```
static const struct imx_rproc_att imx_rproc_att_imx8mn[] = {
 /* dev addr , sys addr  , size     , flags */
 /* ITCM   */
 { 0x00000000, 0x007E0000, 0x00020000, ATT_OWN | ATT_IOMEM },
 /* OCRAM_S */
 { 0x00180000, 0x00180000, 0x00009000, 0 },
 /* OCRAM */
 { 0x00900000, 0x00900000, 0x00020000, 0 },
 /* OCRAM */
 { 0x00920000, 0x00920000, 0x00020000, 0 },
 /* OCRAM */
 { 0x00940000, 0x00940000, 0x00050000, 0 },
 /* QSPI Code - alias */
 { 0x08000000, 0x08000000, 0x08000000, 0 },
 /* DDR (Code) - alias */
 { 0x10000000, 0x40000000, 0x0FFE0000, 0 },
 /* DTCM */
 { 0x20000000, 0x00800000, 0x00020000, ATT_OWN | ATT_IOMEM },
 /* OCRAM_S - alias */
 { 0x20180000, 0x00180000, 0x00008000, ATT_OWN },
 /* OCRAM */
 { 0x20200000, 0x00900000, 0x00020000, ATT_OWN },
 /* OCRAM */
 { 0x20220000, 0x00920000, 0x00020000, ATT_OWN },
 /* OCRAM */
 { 0x20240000, 0x00940000, 0x00040000, ATT_OWN },
 /* DDR (Data) */
 { 0x40000000, 0x40000000, 0x80000000, 0 },
};
```

1. The rproc framework must be activated in the kernel configuration if it is not enabled by default.

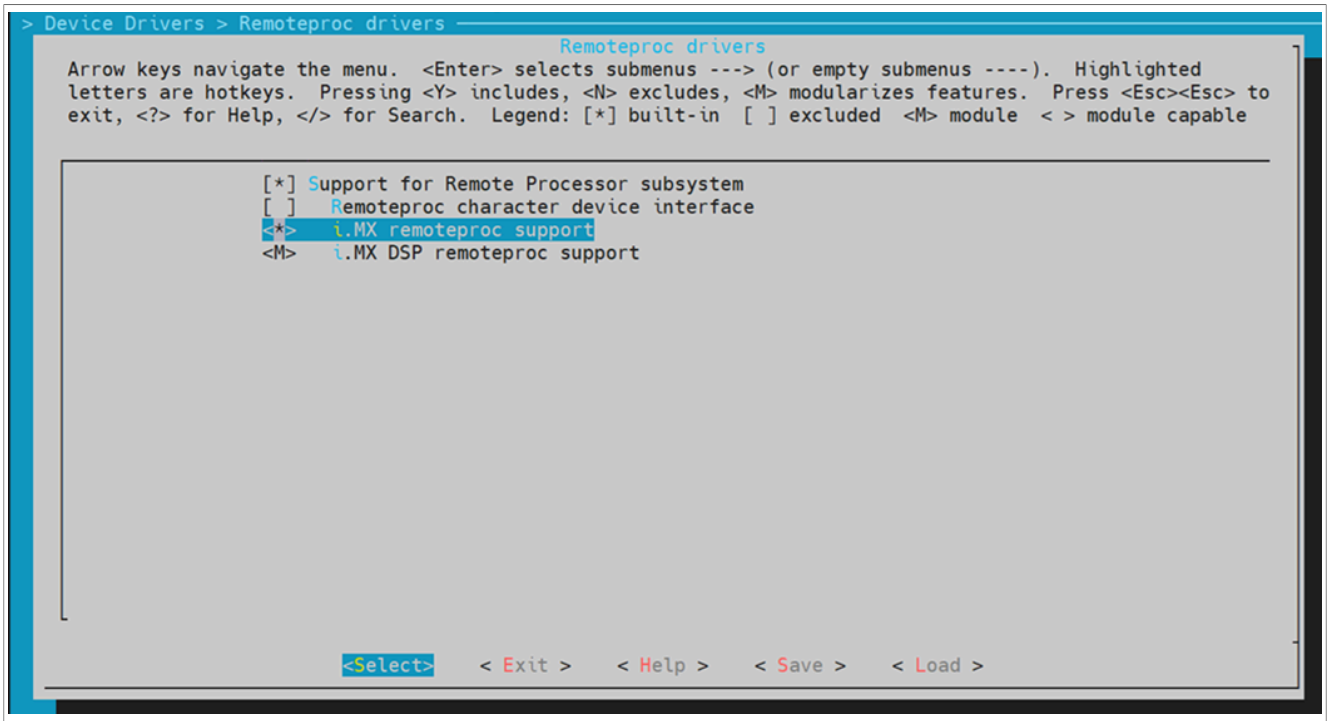

Figure 3. Memory map for i.MX 8MN / i.MX 8MP defined in "imx_rproc"

2. The rproc framework is enabled in device tree using a dedicated DTS node. This required node is usually provided in "imx-*-rpmsg.dts" by default in NXP Linux BSP. For example, i.MX 8MP uses the following DTS node, which specifies what mailboxes are needed for the communication between Linux and the M7 SDK app.

```
imx8mp-cm7 {
  compatible = "fsl,imx8mn-cm7";
  rsc-da = <0x55000000>;
  clocks = <&clk IMX8MP_CLK_M7_DIV>;
  mbox-names = "tx", "rx", "rxdb";
  mboxes = <&mu 0 1
            &mu 1 1
            &mu 3 1>;
  memory-region = <&vdevbuffer>, <&vdev0vring0>, <&vdev0vring1>,
 <&rsc_table>;
  status = "okay";
};
```

The "memory-region" attribute must contain the memory sections that are used by the firmware ELF to allow reloading from sysfs.

For example, for i.MX 8MP, the following sections can be added:

- Memory regions definitions from the reserved-memory node:

```
m4_reserved: m4@0x80000000 {
  no-map;
  reg = <0 0x80000000 0 0x1000000>;
  };
m7_ddr_alias: m4@0x10000000 {
  no-map;
  reg = <0 0x10000000 0 0x1000000>;
  };
```

```
m7_itcm: m4@0x7E0000 {
  no-map;
  reg = <0 0x7E0000 0 0x20000>;
  };
m7_dtcm: m4@0x800000 {
  no-map;
  reg = <0 0x800000 0 0x20000>;
};
```

- rproc DTS node referring to used memory nodes:

```
imx8mp-cm7 {
  compatible = "fsl,imx8mn-cm7";
  rsc-da = <0x55000000>;
  clocks = <&clk IMX8MP_CLK_M7_DIV>;
  mbox-names = "tx", "rx", "rxdb";
  mboxes = <&mu 0 1
            &mu 1 1
            &mu 3 1>;
  memory-region = <&vdev0vring0>, <&vdev0vring1>, <&vdevbuffer>,
 <&m4_reserved>, <&m7_ddr_alias>, <&m7_itcm>, <&m7_dtcm>;
  status = "okay";
};
```

9.1.1 How to use rproc on i.MX platforms

This subsection shows the usage of rproc on i.MX platforms using the i.MX 8MP platform as a reference.

There are three possibilities to load and start the remote processor firmware:

- The firmware can be started via the sysfs interface.
- The firmware can be started automatically by the remoteproc driver at the probing stage.
- Boot the firmware early using U-Boot and control the firmware using the sysfs interface.

Note: *For i.MX 8M platforms, the root clock for M7/M4 must be kept always enabled by Linux to load the firmware code and start Cortex M7/M4. By default, NXP Linux BSP keeps the root clock enabled for the M core when it is started from U-Boot. Otherwise, if you must firstly start the M core from the Linux booting phase, the clock driver (drivers/clk/imx/clk-composite-8m.c) must be updated to always skip the gate registration to keep the root clock always enabled for the M core.*

9.1.1.1 Starting firmware using sysfs interface

1. Power up the board and stop in U-Boot and run the following command:

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

2. rproc exports the rproc functionalities to UserSpace using sysfs.

```
$ ls /sys/class/remoteproc/remoteproc0/
consumers  device    name   recovery  subsystem  uevent
coredump   firmware  power  state     suppliers
```

Note: *If /sys/class/remoteproc/remoteproc*/ is empty, the rproc framework is not enabled in the device tree. The previous section describes how to enable rproc. When the default NXP image is used, the rproc can be enabled by setting the device tree to imx8mp-evk-rpmsg.dtb.*

3. If the ELF is not stored in */lib/firmware*, set the new path.

```
# echo -n <firmware_path> > /sys/module/firmware_class/parameters/path
```

```
$ echo -n /run/media/mmcblk1p1/ > /sys/module/firmware_class/parameters/path
```

4. If the filename of the firmware ELF is different from the default one, set it to the new one.

```
# echo -n <firmware_name.elf> > /sys/class/remoteproc/remoteproc<N>/firmware
$ echo -n imx8mp_m7_TCM_hello_world.elf > /sys/class/remoteproc/remoteproc0/
firmware
```

5. Check the state of the remote processor before starting it with a new firmware. If it is online, it should be stopped.

```
# cat /sys/class/remoteproc/remoteproc<N>/state
$ cat /sys/class/remoteproc/remoteproc0/state
offline
```

6. Start the remote processor with the new firmware.

```
# echo start > /sys/class/remoteproc/remoteproc<N>/state
$ echo start > /sys/class/remoteproc/remoteproc0/state
$ cat /sys/class/remoteproc/remoteproc0/state
running
```

7. Stop the remote processor.

```
# echo stop > /sys/class/remoteproc/remoteproc<N>/state
$ echo stop > /sys/class/remoteproc/remoteproc0/state
$ cat /sys/class/remoteproc/remoteproc0/state
offline
```

Important: *On i.MX 8M platforms, "remoteproc" stops only the Cortex-M CPU, not the Cortex-M system. Therefore, any in-flight Cortex-M bus transactions would hang after the CPU is halted and this can only be resolved with a full SoC reset. It is not recommended to stop the Cortex-M7 CPU in a production system. If the system must stop the Cortex-M7 CPU, reload the image and restart it. Then, the Cortex-M7 CPU must be in the WFI state and have no external access to the Cortex-M7 TCM memory through eDMA or other similar transactions. A possible solution is to implement a handshake between the Cortex-M and Cortex-A CPUs to confirm that the Cortex-M CPU is safe to stop or reset.*

9.1.1.2 Starting firmware automatically by remote PROC driver during Linux kernel boot time

Starting the firmware automatically by a remote PROC and not from U-Boot or Linux console is possible if the "fsl,auto-boot" property is set to 1 in the "imx8xx-evk-rpmsg.dts" file:

```
imx8mp-cm7 {
  compatible = "fsl,imx8mn-cm7";
  rsc-da = <0x55000000>;
  clocks = <&clk IMX8MP_CLK_M7_DIV>,
    <&audio_blk_ctrl IMX8MP_CLK_AUDIOMIX_AUDPLL_ROOT>;
  clock-names = "core", "audio";
  mbox-names = "tx", "rx", "rxdb";
  mboxes = <&mu 0 1
     &mu 1 1
     &mu 3 1>;
memory-region = <&vdevbuffer>, <&vdev0vring0>, <&vdev0vring1>, <&rsc_table>;
  status = "okay";
  fsl,startup-delay-ms = <500>;
  fsl,auto-boot = <1>;
 };
```

To apply the changes, recompile the "imx8xx-evk-rpmsg.dtb" and set it as the device tree when U-Boot is paused. As an alternative, the default "imx8mx-evk-rpmsg.dtb" can be configured directly from U-Boot using "fdt" (this example is for "imx8mp-evk-rpmsg.dtb"):

```
u-boot=> setenv fdtpatch 'fdt addr $fdt_addr; fdt resize; fdt set /imx8mp-cm7
 "fsl,auto-boot" "<1>"; fdt print /imx8mp-cm7'
u-boot=> editenv loadfdt
edit: fatload mmc ${mmcdev}:${mmcpart} ${fdt_addr_r} ${fdtfile}; run fdtpatch
u-boot=> run prepare_mcore
u-boot=> boot
```

To select an ELF firmware file to be auto-booted, store the file in the "/lib/firmware" folder and rename it to "rproc-imx-rproc-fw" without specifying the ELF extension at the end of the filename. For example, if the ELF file is named "hello_world.elf", then it should be renamed to "rproc-imx-rproc-fw".

Note: *To find out how the ELF file should be named, use the following command (this is useful to check when the name is changed in a future version of the BSP):*

```
# The firmware name is stored in /sys/class/remoteproc/remoteproc0/firmware
$ cat /sys/class/remoteproc/remoteproc0/firmware
rproc-imx-rproc-fw
```

9.1.1.3 Booting firmware early using U-Boot and controlling firmware using sysfs interface

i.MX U-Boot can start Cortex-M from the U-Boot console level using the **bootaux** command. Depending on the used i.MX 8M, i.MX 8, or i.MX 8X platforms, some of them have already defined Cortex-M booting commands.

To define a booting command for the Cortex-M7 CPU on i.MX 8MP, use the following commands in U-Boot:

```
@ M7 Bin Filename - stored on 1st partition
u-boot=>  setenv m7image imx8mp_m7_TCM_hello_world.bin
# M7 Load command
u-boot=>  setenv load_m7image 'fatload mmc 1:1 0x48000000 ${m7image}; cp.b
 0x48000000 0x7e0000 0x20000;'
# M7 Start Command
u-boot=>  setenv m7boot 'run load_m7image; bootaux 0x7e0000'
# Start M7
u-boot=>  run m7boot
13062 bytes read in 28 ms (455.1 KiB/s)
## Starting auxiliary core stack = 0x20020000, pc = 0x00002B0D...
u-boot=> run prepare_mcore; boot
```

After the Cortex-M CPU is started from U-Boot, Linux can be started using the “boot” command. To stop and reload the Cortex-M CPU, perform the following steps:

1. Stop the remote processor before deploying new firmware.

```
# echo stop >  /sys/class/remoteproc/remoteproc<N>/state
$ echo stop >  /sys/class/remoteproc/remoteproc0/state
```

2. If ELF is not stored in */lib/firmware*, set a new path.

```
# echo -n <firmware_path> > /sys/module/firmware_class/parameters/path
$ echo -n “/run/media/mmcblk1p1/” > /sys/module/firmware_class/parameters/
path
```

3. If the filename of the firmware ELF is different from the default one, set it to a new one.

```
# echo -n <firmware_name.elf> > /sys/class/remoteproc/remoteproc<N>/firmware
```

```
$ echo -n imx8mp_m7_TCM_hello_world.elf > /sys/class/remoteproc/remoteproc0/
firmware
```

4. Start the remote processor with the new firmware.

```
# echo start >  /sys/class/remoteproc/remoteproc<N>/state
$ echo start >  /sys/class/remoteproc/remoteproc0/state
```

10 References

1. *i.MX 8MP Applications Processor Reference Manual* (document IMX8MP)
2. *i.MX 8MQ Applications Processor Reference Manual* (document IMX8MDQLQRM)
3. *i.MX 8MM Applications Processor Reference Manual* (document IMX8MMRM)
4. *i.MX 8MN Applications Processor Reference Manual* (document IMX8MNRM)
5. *i.MX 8QM Applications Processor Reference Manual* (document IMX8QMRM)
6. *i.MX 8QXP Applications Processor Reference Manual* (documentIMX8DQXPRM)
7. *i.MX 7Dual Applications Processor Reference Manual* (document IMX7DRM)
8. *i.MX 6SoloX Applications Processor Reference Manual* (document IMX6SXRM)
9. Linux Remote Processor Framework documentation

11 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2024 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials must be provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

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

12 Revision history

Table 9. Revision history

Document ID	Release date	Description
AN5317 v.3.0	07 October 2024	Information related to the i.MX 8M family has been updated
2	18 November 2021	Introduced rproc and updated the look and feel of the document

Table 9. Revision history...*continued*

Document ID	Release date	Description
1	08/2019	Introduced i.MX 8M support
0	08/2016	Initial release

Legal information

Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

HTML publications — An HTML version, if available, of this document is provided as a courtesy. Definitive information is contained in the applicable document in PDF format. If there is a discrepancy between the HTML document and the PDF document, the PDF document has priority.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Loading Code on Cortex-M from U-Boot/Linux for the i.MX Asymmetric Multi-Processing Application Processors

Contents