Manual for i.MX-based Hardware and BSPs

Attention: Preliminary State

This documentation and the software BSP is in an early state of development. All parts are subject to change without notice. For more information on the development state, please see the Release Notes and the Known Issues.

Mainline Platform Support & Vendor BSP

For our i.MX6 and i.MX8 hardware we are aiming at using mainline components as much as possible. In contrast to the vendor BSP, that heavily relies on unmaintained and unstable code, we are constantly working towards having our boards supported upstream in major software components like the Linux kernel and the U-Boot bootloader.

This means that we **benefit from the work of the communities and developers around the world and join them with our own efforts**. But it also means that support for certain features of the platform **won't be available until an acceptable and stable solution has been found** within the communities.

This guide is intended to help developers setup, use and integrate the hardware and software provided by Kontron Electronics, that is based on the NXP i.MX SoCs.

For a more generic overview of the platform-independent parts of the software environment please see the "Main Documentation".

BL i.MX8MM Boards and Demo Kits

For an introduction to the i.MX8MM boards visit the Getting Started page as well as the Board Overview and Using the System pages.

BSP Overview

Hardware

Kontron Electronics offers Eval-Kits (EVKs) and devices ready for integrating into your product to get you started quickly with your product and application design. The abilities of your customized hardware can be fitted to your specific needs.

Here are some general specs for our standard i.MX hardware. Please navigate to the description of the specific hardware you are using and visit our website for more information about the available boards, modules and devices.

SoCs

SoCs	Cores	NXP Website
NXP i.MX6 Solo	ARM® Cortex®-A9 @ 800 MHz	Link
NXP i.MX6 Dual	2 x ARM® Cortex®-A9 @ 1200 MHz	Link
NXP i.MX6 UltraLite	ARM® Cortex®-A7 @ 528 MHz	Link
NXP i.MX6 ULL	ARM® Cortex®-A7 @ 792 MHz	Link
NXP i.MX8M Mini	4 x ARM® Cortex®-A53 @ 1.8 GHz, ARM® Cortex®-M4 @ 400 MHz	Link

On-Board Memory

Memory Type	Size
DDR3 or (LP)DDR4 RAM	up to 2GB
SPI NOR Flash	up to 2MB
Parallel or Serial NAND	up to 512MB
eMMC	up to 4GB

Interfaces

Interface	Standard/Specs	Ports
Ethernet	100/1000 MBit/s	up to 2
Display	RGB, LVDS, HDMI, DSI	up to 2
USB	Host, Device, OTG	up to 4
Serial	RS232, RS485, CAN, I2C	up to 1 each

Miscellaneous

Spec	Value
Supply Voltage	24V DC

Software

- U-Boot 2020.01 bootloader
- Linux 5.4 mainline kernel with board adaptations

- Linux userland based on the Yocto reference distribution "Poky"
- GStreamer multimedia framework with hardware acceleration
- Qt 5.x based on eglfs with OpenGL and QML/QtQuick support

Yocto Build System

For generic information on how to setup and use the Yocto BSPs provided by KED, please read the main documentation first.

Repository and Directory Structure

This is how the directory tree with the most important files and directories of the i.MX BSP will look like:

```
yocto-ktn                   # the core repository
│
├── build-ktn-imx           # the build repository for the i.MX BSP
│   │
│   ├── conf
│   │   ├── repo.conf       # specifies the revisions of all layers
│   │   ├── local.conf      # specifies local settings for the
build
│   │   └── bblayers.conf   # specifies all layers that will be
parsed by bitbake
│   │
│   └── tmp                 # contains all of the build data
│       ├── deploy
│       │   ├─ images       # contains image files and binaries
for the target
│       │   ├─ ipk          # contains packages
│       │   ├─ licenses     # contains licenses of the packages in
use
│       │   └─ sdk          # contains SDK and toolchain binaries
│       │
│       └── work
│           └─ ...          # contains all source and build files
for the packages
│
├── layers                  # contains all meta layers with recipes
│   │                       # (each one is a git repository)
│   │
│   ├ poky                  # contains the Yocto/Poky build system
and meta data
│   ├ meta-openembedded     # contains basic meta layers
│   ├ meta-ktn              # contains basic Kontron adaptations
and modifications
```

```
|    ├ meta-ktn-imx          # contains Kontron platform
adaptations and modifications for i.MX
|    ├ meta-freescale        # contains NXP platform adaptations
and modifications
|    └ ...
|
├── scripts                  # contains scripts to automate certain
tasks
├── downloads                # contains all the files downloaded by
the fetcher
|                            # (shared by all builds)
├── sstate-cache             # contains the sstate cache (shared by
all builds)
└── init-env                 # this is a script to initialize the
build environment
```

Example Setup

This is an example for setting up and running a build for the `kontron-mx6ul` machine. For more details on each of these steps, please visit the main documentation. For information and examples on your specific hardware, please look in the hardware section.

```
# move to your working directory
cd ~

# clone the core repository
git clone https://git.kontron-electronics.de/yocto-ktn/yocto-
ktn.git

# clone the build repository for NXP i.MX and initialize the build
environment
# for the 'kontron-mx6ul' machine configuration.
. init-env -u -m kontron-mx6ul build-ktn-imx

# build an image including the Qt5 libraries and demos for the
target hardware
bitbake image-ktn-qt
```

Modify the BSP

This section provides some examples for modifying the BSP for i.MX. For further general information on how to modify the BSPs, please visit the main documentation. For information and examples on your specific hardware, please look in the hardware section.

Modifying the Kernel Configuration

With Bitbake

```
# use menuconfig to change the configuration and save them
in .config
bitbake virtual/kernel -c menuconfig

# rebuild the kernel (and the image if needed) to test the changes
bitbake virtual/kernel -C compile -f
bitbake image-ktn

# create a reduced defconfig
bitbake virtual/kernel -c savedefconfig

# make the changes persistent by copying the defconfig to your
meta-layer
cp ~/yocto-ktn/build-ktn-imx/tmp/work/kontron_mx6ul-ktn-linux-
gnueabi/linux-ktn/5.4-r0/build/defconfig ~/yocto-ktn/layers/meta-
<customer>/recipes-kernel/linux/linux-ktn/mx6/defconfig
```

In a 'devshell'

```
# open a devshell for the kernel
bitbake virtual/kernel -c devshell

# use menuconfig to change the configuration and save them
in .config
make menuconfig

# build the kernel to test the changes
make

# create a reduced defconfig
make savedefconfig
```

```
# make the changes persistent by copying the defconfig to your meta-layer
cp ~/yocto-ktn/build-ktn-imx/tmp/work/kontron_mx6ul-ktn-linux-gnueabi/linux-ktn/5.4-r0/build/defconfig ~/yocto-ktn/layers/meta-<customer>/recipes-kernel/linux/linux-ktn/mx6/defconfig
```

Modifying the Kernel Code

```
# start devtool to create a temporary workspace for the kernel source code
devtool modify linux-ktn

# modify the code and rebuild with
bitbake linux-ktn

# test your changes, create patches if necessary and reset the recipe with
devtool reset linux-ktn
```

Booting an Image

Boot Chain Overview

After powering up the device, the BootROM code in the i.MX tries to load a first stage bootloader which is U-Boot SPL in our case. From there, the chain continues with loading of TF-A (Trusted-Firmware-ARM, only for i.MX8MM) and U-Boot. Finally we load the Linux Kernel and Devicetrees.

Boot Stage	Name	Description	Filename in Yocto-Build	Image Format
1	BootROM	Fixed program in the SoC	-	-
2	U-Boot SPL	Secondary Program Loader	flash.bin	Raw Binary
3*	TF-A BL31	Trusted-Firmware-ARM BL31	u-boot.itb	FIT
4	U-Boot	Main Bootloader	u-boot.itb	FIT
5	Linux	Linux Kernel + Devicetrees	fitImage	FIT

* only i.MX8MM

SPL

The U-Boot SPL brings up the most important parts of the SoC, such as the core and the DDR-RAM. Afterwards it loads a FIT image from a fixed offset on the boot device (NOR, SD card, eMMC) to the DDR RAM and jumps to the TF-A or U-Boot entrypoint. It also prints a few lines to the debug UART interface. You should see something like the following:

```
U-Boot SPL 2020.01_ktn-zeus_3.0.0-alpha2-dirty+gc6e6927046 (Feb 20
2020 - 10:00:34 +0000)
1GB RAM detected, assuming Kontron N801x module...
Normal Boot
Trying to boot from MMC2
```

TF-A BL31 (only i.MX8MM)

The TF-A BL31 is run at EL3 (exception level 3). It sets up the default security settings for peripherals and can act as a secure monitor for code running in the non-secure world (Linux). For more information about the TF-A and the overall firmware design, see the TF-A documentation.

In our configuration TF-A BL31 also prints a few lines to the debug UART interface. You should see something like the following:

```
NOTICE:  BL31: v2.2(release):v2.2-dirty
NOTICE:  BL31: Built : 10:25:02, Feb 13 2020
```

U-Boot

U-Boot acts as a main bootloader. It has a command line interface and runs from DDR RAM. It has support for filesystems and many other things. The typical output from U-Boot will look something like this:

```
U-Boot 2020.01_ktn-zeus_3.0.0-alpha2-dirty+gc6e6927046 (Feb 20
2020 - 10:00:34 +0000)

CPU:   Freescale i.MX8MMQ rev1.0 at 1200 MHz
Reset cause: POR
Model: Kontron i.MX8MM N8011 S
DRAM:  1 GiB
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
In:    serial
Out:   serial
Err:   serial
Net:   eth0: ethernet@30be0000 [PRIME]
```

Booting from SD Card

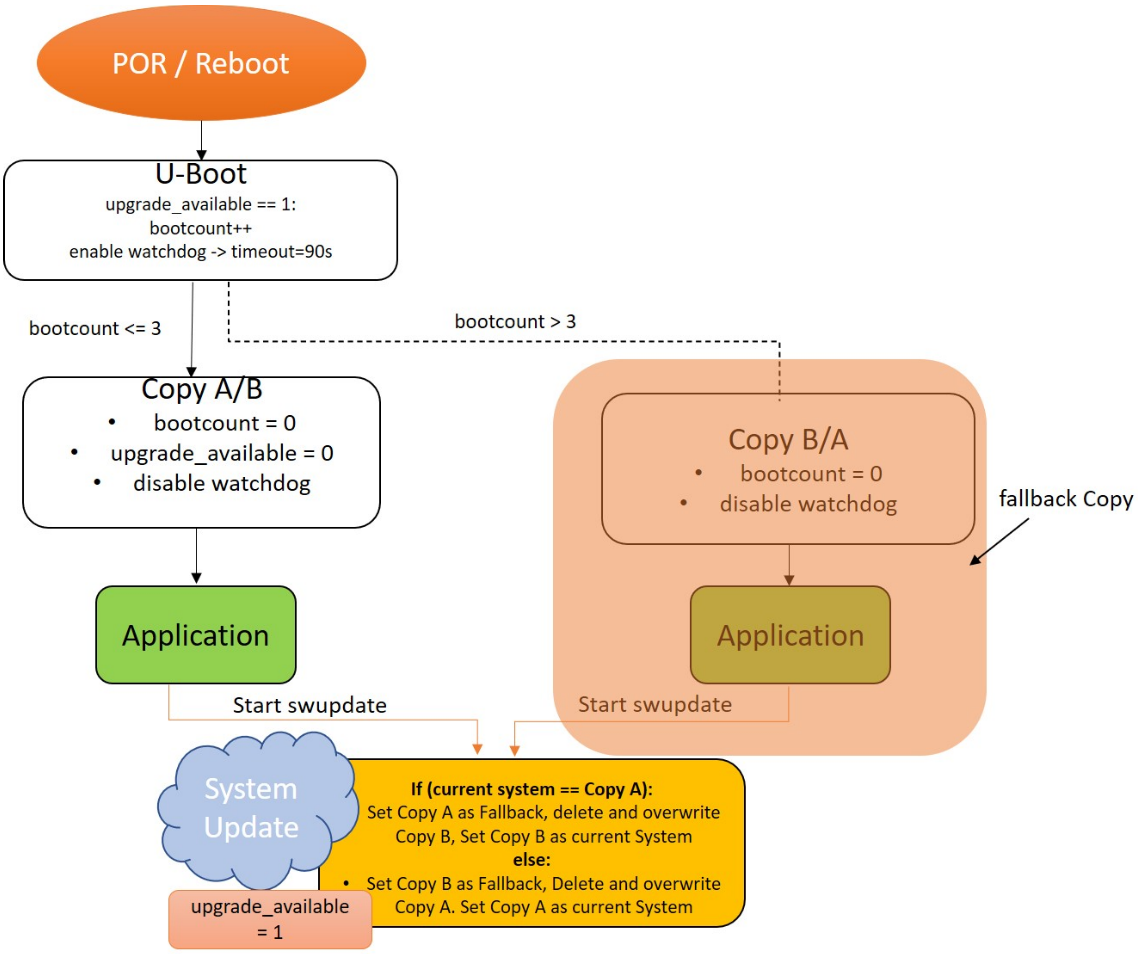
You can use the prebuilt images from files.kontron-electronics.de or the images from your Yocto build. To write the image onto the SD card using a Linux host, use the following command and adjust the image file name and the device filename for the SD card to your needs.

> **Danger**
>
> The usage of the `dd` command in combination with `sudo` can be dangerous. If you use wrong parameters, this might cause severe loss of data on your hard disk or other drives!

```
gunzip -c image-ktn-kontron-mx8mm-ktn-zeus-v3.0.0-alpha2-20200203.rootfs.wic.gz | sudo dd of=/dev/<mmcdevice> bs=1M iflag=fullblock oflag=direct conv=fsync status=progress
```

System Boot diagram A/B Partition

The Flash Layout described in the chapter Flash Layout + the provided bootscripts follow the Boot-diagram below.


POR / Reboot
U-Boot
upgrade_available == 1:
bootcount++
enable watchdog -> timeout=90s
bootcount <= 3
bootcount > 3
Copy A/B
• bootcount = 0
• upgrade_available = 0
• disable watchdog
Copy B/A
• bootcount = 0
• disable watchdog
fallback Copy
Application
Application
Start swupdate
Start swupdate
System Update
upgrade_available = 1
If (current system == Copy A):
Set Copy A as Fallback, delete and overwrite Copy B, Set Copy B as current System
else:
• Set Copy B as Fallback, Delete and overwrite Copy A. Set Copy A as current System

Using the Internal Flash Storage

As described in the chapter Booting an Image you can generate a SD card for your device and use it during development. When you are interested in using the internal flash (SPI-NOR, eMMC, NAND) then continue reading. In the following it is described how to setup your device with a A/B partition scheme so that you will be able to update your system seamlessly in the future.

Partition Layout

Kontron i.MX8MM

Component	Storage	Linux Devicenode	Size	Filesystem
Linux Root-Filesystem A/B Layout, Boot Volume and Data Volume	eMMC, NAND	`/dev/mmcblk0pX`	32GB	ext4
U-Boot	NOR	`/dev/mtd0`	1920 KiB	raw
U-Boot env (copy 1)	NOR	`/dev/mtd1`	64 KiB	raw
U-Boot env (copy 2)	NOR	`/dev/mtd2`	64 KiB	raw

Kontron i.MX6UL

Component	Storage	Linux Devicenode	Size	Filesystem
Linux Root-Filesystem A/B Layout, Boot Volume and Data Volume	eMMC, NAND	`/dev/mtd0`	512MB	ubifs
U-Boot SPL	NOR	`/dev/mtd1`	896 KiB	raw
U-Boot env (copy 1)	NOR	`/dev/mtd2`	64 KiB	raw
U-Boot env (copy 2)	NOR	`/dev/mtd3`	64 KiB	raw

Factory Setup / Initial Partitioning

if you have a Kontron Demoboard with Software release version **greater than v5.0.0** initial partitioning is already completed in factory. You don't have to repeat the step. Just skip to the chapter Perform seamless system Update

Install Images

using ptool

For factory setup you need the images listed below. The images are created during the Yocto build when executing `bitbake image-ktn` or bitbake `bitbake image-ktn-qt` . The images are then deployed to the directory `~/yocto-ktn/build-ktn-imx/tmp/deploy/$MACHINE/images` .

Description	Image Name (mx6ul)	Image Name (mx8mm)
U-Boot	`flash.bin`	`flash.bin`
U-Boot environment	`u-boot-initial-env-kontron-mx6ul`	`u-boot-initial-env-kontron-mx8mm`
bootfs	`image-ktn-bootfs-kontron-mx6ul-ktn-dunfell.tar.gz`	`image-ktn-bootfs-kontron-mx8mm-ktn-dunfell.tar.gz`
rootfs	`image-ktn-kontron-mx6ul.tar.gz`	`image-ktn-kontron-mx8mm.tar.gz`
userfs	`image-ktn-userfs-kontron-mx6ul-ktn-dunfell.tar.gz`	`image-ktn-userfs-kontron-mx8mm-ktn-dunfell.tar.gz`

1. Copy the images into the directory `/home/root/fw/` on your device.
2. Flash bootloader binaries and environment

```
ptool flash_bl_spl && ptool flash_bootvars
```

using Swupdate (only Kontron i.MX8MM)

Initial setup is also implemented for i.MX8MM devices. You only need the swu archive (bitbake swupdate-img-qt) and you should be able to install it locally or via integrated webserver.

eMMC / NAND Partition sizes

To change the default partition sizes, see `/usr/share/production/prod-env.sh` and change the default values. Consider that the eMMC has a maximum of 32GB.

```
# partition sizes on eMMC (MB)
rootfs_size=10000
bootfs_size=50
# userfs (if size empty -> all remaining space taken else SIZE in
MiB)
userfs_size=""
```

1. Flash the eMMC on i.MX8MM or the NAND flash on i.MX6UL devices

    3.1 Kontron i.MX8MM: `ptool flash_emmc_AB`

    3.2 Kontron i.MX6UL: `ptool flash_ubi_AB`

This command will take some time and the status will be prompted. Afterwards you can check the eMMC device with `lsblk`

```
root@kontron-mx8mm:~# lsblk | grep mmcblk0
mmcblk0      179:0    0 29.1G  0 disk
├─mmcblk0p1  179:1    0   50M  0 part /mnt
├─mmcblk0p2  179:2    0  9.8G  0 part
├─mmcblk0p3  179:3    0  9.8G  0 part
└─mmcblk0p4  179:4    0  9.6G  0 part /usr/local
mmcblk0boot0 179:32   0    4M  1 disk
mmcblk0boot1 179:64   0    4M  1 disk
```

The NAND flash with `ubinfo -a`

```
Volume ID:   0 (on ubi0)
Type:        dynamic
Alignment:   1
Size:        124 LEBs (31490048 bytes, 30.0 MiB)
State:       OK
Name:        boot
Character device major/minor: 245:1
-----------------------------------
Volume ID:   1 (on ubi0)
Type:        dynamic
Alignment:   1
Size:        620 LEBs (157450240 bytes, 150.1 MiB)
State:       OK
Name:        root_A
Character device major/minor: 245:2
-----------------------------------
Volume ID:   2 (on ubi0)
Type:        dynamic
Alignment:   1
Size:        620 LEBs (157450240 bytes, 150.1 MiB)
State:       OK
```

```
Name:          root_B
Character device major/minor: 245:3
-----------------------------------
Volume ID:    3 (on ubi0)
Type:         dynamic
Alignment:    1
Size:         636 LEBs (161513472 bytes, 154.0 MiB)
State:        OK
Name:         data
```

Why Do we Need a Bootfs Image?

In our implementation the bootfs is mandatory. It contains the following files:

```
root@kontron-mx8mm:~# ls /mnt/
fitImage_active    fitImage_inactive  lost+found         sys_active
```

The U-Boot looks into the file `sys_active` which is the current active Rootfs partition. After an update this file gets altered, so that partitions are switched. Also we update the kernel in the last step and keep the old one (active/inactive) as fallback if something bad happens. Switching back after failure is done automatically from executed U-Boot scripts.

Why Do we Need a Userfs Image?

The userfs image is not mandatory but can be used to install applications and files which should persist during a system update. If you install files during the build into `/usr/local` those files will get extracted and copied into the userfs image. You can see an example in this recipe.

Perform Seamless System Update using SWUpdate

The advantage of having an A/B setup is that you can perform a system update without having to boot into separate rescue/recovery system. This will make the update more comfortable in the end because the update can be downloaded and executed in the background during normal operation.

A disadvantage is the bigger memory footprint of the device. If you are interested in different concepts and in general how it works, please read the SWUpdate documentation.

Creating an SWUpdate Image

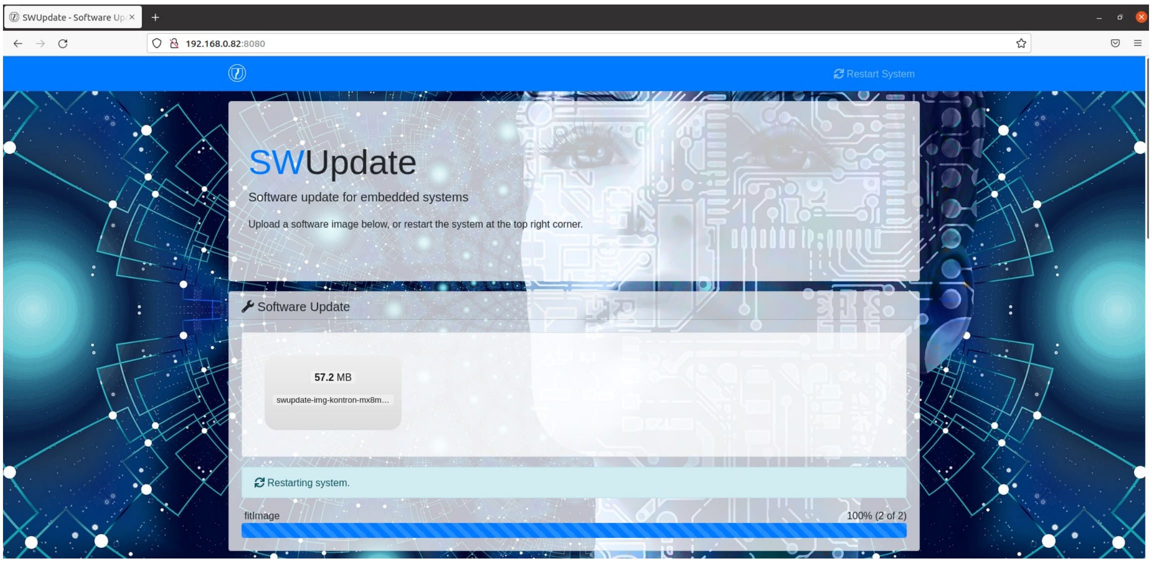
You need a special image type (`*.swu` archive) which can be read and executed by the `swupdate` tool. This file can be created with `bitbake`:

```
bitbake swupdate-img(-qt)
```

This will create an archive containing our default image `image-ktn(-qt)` and includes further files like `sw-description` which `swupdate` can use and install.

Install Update with Integrated Webserver

The default image starts a webserver provided by `swupdate`. It will listen for connections on `http://$DEVICE_IP:8080`. The device will make a DHCP request and fallback to 192.168.1.11 if no DHCP server is available. With the webserver you can download and install a `*.swu` image onto the device.



SWUpdate

Software update for embedded systems

Upload a software image below, or restart the system at the top right corner.

Restarting system.

Hint

This is just an example of a quite handy feature of `swupdate`. In the field you probably want to download the update from an external webserver or from an USB drive. Please look at the file `/usr/lib/swupdate/conf.d/09-swupdate-args` how `swupdate` is invoked. Further parameters are described in the official SWUpdate documentation.

After installing you can reboot the device which will then start into the updated partition.

Using the System

Please also have a look at the "Supported Hardware" section for hardware-specific guides and examples.

Todo

Release Notes

5.0.0 (2022-04)

Supported Hardware

DK i.MX8MM: Demokit with i.MX8MM SoM (N801x-S)

- **Supported peripherals in Linux:** Ethernet 1 & 2, USB Host, USB OTG, SD card, Debug Console, eMMC, SPI NOR, RTC, RS232, CAN, LEDs, GPIOs, PWM-Beeper, HDMI, LVDS, I2C-Touchscreen, RS485, GPU
- **Supported boot devices:** SD card, SPI NOR, eMMC
- **Supported System Boot** A/B partitios on eMMC
- **Not supported or untested:** VPU, Temperature Monitoring, PCIe, Suspend/Sleep

DK i.MX6UL/ULL: Demokit with i.MX6UL/ULL SoM (N6x1x-S)

- **Supported peripherals in Linux:** Ethernet 1 & 2, USB Host, USB OTG, SD card, Debug Console, NAND FLash, SPI NOR, RTC, RS232, CAN, LEDs, GPIOs, PWM-Beeper, RGB, I2C-Touchscreen, RS485
- **Supported boot devices:** SD card, SPI NOR
- **Supported System Boot** A/B partitions on NAND-Flash
- **Not supported or untested:** Suspend/Sleep

Changelog

- Linux: Update to v5.10.103
- U-Boot: Drop extlinux support
- U-Boot: Support A/B System Update
- U-Boot / Linux: Use redundant U-Boot Partition
- General: Support System Update (A/B) using swupdate tool

A/B System Updates

Release 5.0.0 supports seamlessly system updates. This means that you can update your device without booting into a separate partition or device. This requires a higher memory footprint and you must keep it in mind if you are developing a custom rootfs. Other non A/B setups are not supported at the moment.

4.0.0-beta (2020-11-02)

Attention: Beta Release

This is a beta release version of the BSP. Not all parts are verified and tested.

Supported Hardware

DK i.MX8MM: Demokit with i.MX8MM SoM (N801x-S)

- **Supported peripherals in Linux:** Ethernet 1 & 2, USB Host, USB OTG, SD card, Debug Console, eMMC, SPI NOR, RTC, RS232, CAN, LEDs, GPIOs, PWM-Beeper, HDMI *[1]*, LVDS, I2C-Touchscreen, RS485, GPU *[2][3]*
- **Supported boot devices:** SD card, SPI NOR, eMMC
- **Not supported or untested:** VPU, Temperature Monitoring, PCIe, Suspend/Sleep

.

Changelog

- Linux: Update to v5.4.72, update and add further backported patches
- Linux/U-Boot: Simplify board support patches
- U-Boot: Add support for SoM models with 2GB and 4GB RAM
- U-Boot: Drop support for preliminary SoMs with 2GB RAM
- Linux: Preliminary fixes for GPU boot issues (not verified completely) *[3]*
- Yocto: Update to Yocto/OE 3.1.3 + latest patches from dunfell branch

Notes

Please also see the BSP's issue tracker.

[1] HDMI Support

The ADV7535 DSI to HDMI bridge on the i.MX8MM demo board is currently limited to support only CEA861 modes up to 720p. Please consider this if you plan to use the HDMI port.

[2] GPU/Graphics Support

We are currently using the open etnaviv drivers for the GPUs together with downstream NXP drivers for the display subsystem on i.MX8MM. While this works in general, there are known issues (e.g. flickering/jerking of rendered moving objects) in Qt.

[3] Power-Domain Support

We are currently using preliminary backported patches for power domain support in the Linux kernel. There are known problems, especially in relation with the initialization of the GPUs while booting, that might still cause some trouble.

3.0.0-alpha3 (2020-03-23)

Attention: Alpha Release

This is an alpha release version of the BSP. All parts are subject to change without notice.

Supported Hardware

DK i.MX8MM: Demokit with i.MX8MM SoM (Preliminary Development Revision N8010-Rev0 and n8010)

- **Supported peripherals in Linux:** Ethernet 1 & 2, USB Host, USB OTG, SD Card, Debug Console, eMMC, SPI NOR, RTC, RS232, CAN, LEDs, GPIOs, PWM-Beeper, HDMI *[1]*, LVDS, I2C-Touchscreen, RS485, GPU *[2]*
- **Not supported or untested:** VPU, eMMC Boot, SPI NOR Boot, Temperature Monitoring

DK i.MX6UL/ULL: Demokit with i.MX6UL/ULL SoM

- **Supported peripherals in Linux:** SD Card, Debug Console, CAN
- **Not supported or untested:** Ethernet 1 & 2, USB Host, USB OTG, eMMC, SPI NOR, SPI NAND, RTC, RS232, LEDs, GPIOs

Changelog

- U-Boot/Linux i.MX8MM: Create separate devicetrees for the deprecated Rev0 SoM with DA9063 PMIC and the new version. (Please note that the deprecated SoM revision will be removed in the future and is only kept for now until the new hardware is fully available.)
- U-Boot: Add auto detection for the two i.MX8MM SoM variants and connected LVDS panel, to select the best default option in the boot menu.
- Linux: Enable i.MX8MM GPU support with etnaviv drivers.
- Yocto: Update to Yocto/OE 3.0.2

Notes

[1] HDMI Support

The ADV7535 DSI to HDMI bridge on the i.MX8MM demo board is currently limited to support only CEA861 modes up to 720p. Please consider this if you plan to use the HDMI port.

[2] GPU/Graphics Support

We are currently using the open etnaviv drivers for the GPUs together with downstream NXP drivers for the display subsystem on i.MX8MM. While this works in general, there are known issues (e.g. flickering/jerking of rendered moving objects).

3.0.0-alpha2 (2020-01-29)

Attention: Alpha Release

This is an alpha release version of the BSP. All parts are subject to change without notice.

Supported Hardware

DK i.MX8MM: Demokit with i.MX8MM SoM (Preliminary Development Revision)

- **Supported peripherals in Linux:** Ethernet 1 & 2, USB Host, USB OTG, SD Card, Debug Console, eMMC, SPI NOR, RTC, RS232, CAN, LEDs, GPIOs, PWM-Beeper, HDMI *[1]*, LVDS, I2C-Touchscreen
- **Not supported or untested:** RS485, GPU, VPU

DK i.MX6UL/ULL: Demokit with i.MX6UL/ULL SoM

- **Supported peripherals in Linux:** SD Card, Debug Console, CAN
- **Not supported or untested:** Ethernet 1 & 2, USB Host, USB OTG, eMMC, SPI NOR, SPI NAND, RTC, RS232, LEDs, GPIOs

Changelog

- Linux: Add drivers for i.MX8MM graphics and display support (LCDIF, MIPI-DSI, HDMI *[1]* and LVDS).
- Linux: Add support for 2MB SPI-NOR chip MX25V8035F on latest revision of i.MX8MM SoM.
- Linux: Add drivers for PMIC PCA9450 on latest revision of i.MX8MM SoM.

- Linux: Update to latest patch release v5.4.15.
- U-Boot: Update to final v2020.01 and update KTN patches.
- TF-A: Update to v2.2.
- image-ktn-test: Add devregs, libdrm-tests and lmbench.

Notes

[1] HDMI Support

The ADV7535 DSI to HDMI bridge on the i.MX8MM demo board is currently limited to support only CEA861 modes up to 720p. Please consider this if you plan to use the HDMI port.

3.0.0-alpha (Initial Release, 2019-12-19)

Attention: Alpha Release

This is an early alpha release version of the BSP. Large parts of the hardware support were not tested so far. All parts are subject to change without notice.

Version Number

For earlier i.MX Yocto BSPs we already used releases named 1.x.x and 2.x.x. We will keep this scheme by continuing with 3.0.0 and incrementing the first digit for major changes (switch Yocto base release, etc.), the second digit for new features and the last digit for patches and fixes.

Supported Hardware

DK i.MX8MM: Demokit with i.MX8MM SoM (Preliminary Development Revision)

- **Supported peripherals in Linux:** Ethernet 1 & 2, USB Host, USB OTG, SD Card, Debug Console, eMMC, SPI NOR, RTC, RS232, CAN, LEDs
- **Not supported or untested:** HDMI, LVDS, RS485, PWM-Beeper, GPIOs

DK i.MX6UL/ULL: Demokit with i.MX6UL/ULL SoM

- **Supported peripherals in Linux:** SD Card, Debug Console, CAN
- **Not supported or untested:** Ethernet 1 & 2, USB Host, USB OTG, eMMC, SPI NOR, SPI NAND, RTC, RS232, LEDs, GPIOs

Changelog

- Initial Alpha-Release for new i.MX BSP based on Yocto 3.0.1 (Zeus)
- Add limited support for Demokits with i.MX8MM and i.MX6UL/ULL

Issue Tracker

Please have a look at the issue tracker on our GitLab server for known bugs and issues affecting the BSP and also to report any issues you might encounter.

There's also a section in the main documentation about general known issues.

Hardware Overview

SoM Models

Open SoM User Guide PDF
Open SoM Spec Sheet PDF

Name	SoM #	Description
SL i.MX6UL (N6310)	40099 123	i.MX6UL SoM (256MB RAM/NAND)
SL i.MX6UL (N6311)	40099 122	i.MX6UL SoM (512MB RAM/NAND)
SL i.MX6ULL (N6410)	40099 144	i.MX6ULL SoM (256MB RAM/NAND)
SL i.MX6ULL (N6411)	40099 145	i.MX6ULL SoM (512MB RAM/NAND)

Board Models

Open Board Spec Sheet PDF

Name	Kit #	SoM #	Description
DK i.MX6UL	50099 061	40099 122	Demo-Kit
DK 5" i.MX6UL	50099 058	40099 122	Demo-Kit with 5"-Display
DK i.MX6ULL	50099 046	40099 145	Demo-Kit
DK 5" i.MX6ULL	50099 052	40099 145	Demo-Kit with 5"-Display

Using the System

Boot Devices

The i.MX6UL/ULL SoM has the two boot pins on the SoC (BOOT_MODE0 and BOOT_MODE1) unconnected by default, which means the boot mode is set to "00". The BootROM will use the manufacture-mode to look for a bootable image on the SD-card. If none is found, it will fall back to serial loader mode, where it expects an image to be loaded via USB-OTG1 (which is on the micro USB connector on the demo board).

To select other boot devices such as the SPI NOR flash or the eMMC, you need to program the OTP fuses accordingly.

The default setup for production devices from Kontron is that the fuses are set to boot from the SD card (SD1) as primary boot device. The SPI NOR is set as fallback boot device if no SD card is available.

The following OTP register values will be used:

Fuse/Register Name	Offset	Value	Description
BOOT_CFG	0x450	0x49002040	Boot from SD (SD1, 4bit buswidth, 3.3V), enable alternative boot from NOR (SPI2, 3-byte addressing, CS0)
BT_FUSE_SEL	0x460[4]	1	Boot from fuses

If you order SoMs from Kontron and you need a different setup, please supply the necessary information with your order.

Hardware Overview

SoM Models

Open SoM User Guide PDF
Open SoM Spec Sheet PDF

Name	SoM #	Description
SL i.MX8MM (N8010 Rev0)		i.MX8MM SoM (with DA9063 PMIC, Deprectated, Do not use)
SL i.MX8MM (N8010)	40099 175	i.MX8MM SoM (Quad 1,6GHz, 1GB RAM, 8GB eMMC)
SL i.MX8MM (N8011)	40099 185	i.MX8MM SoM (Quad 1,6GHz, 2GB RAM, 8GB eMMC)

Demo Kits

Open Board Spec Sheet PDF

Name	Kit #	SoM #	Description
DK i.MX8MM	50099 059	40099 175	Demo-Kit
DK 7" i.MX8MM	50099 063	40099 175	Demo-Kit with 7"-Display

Getting Started

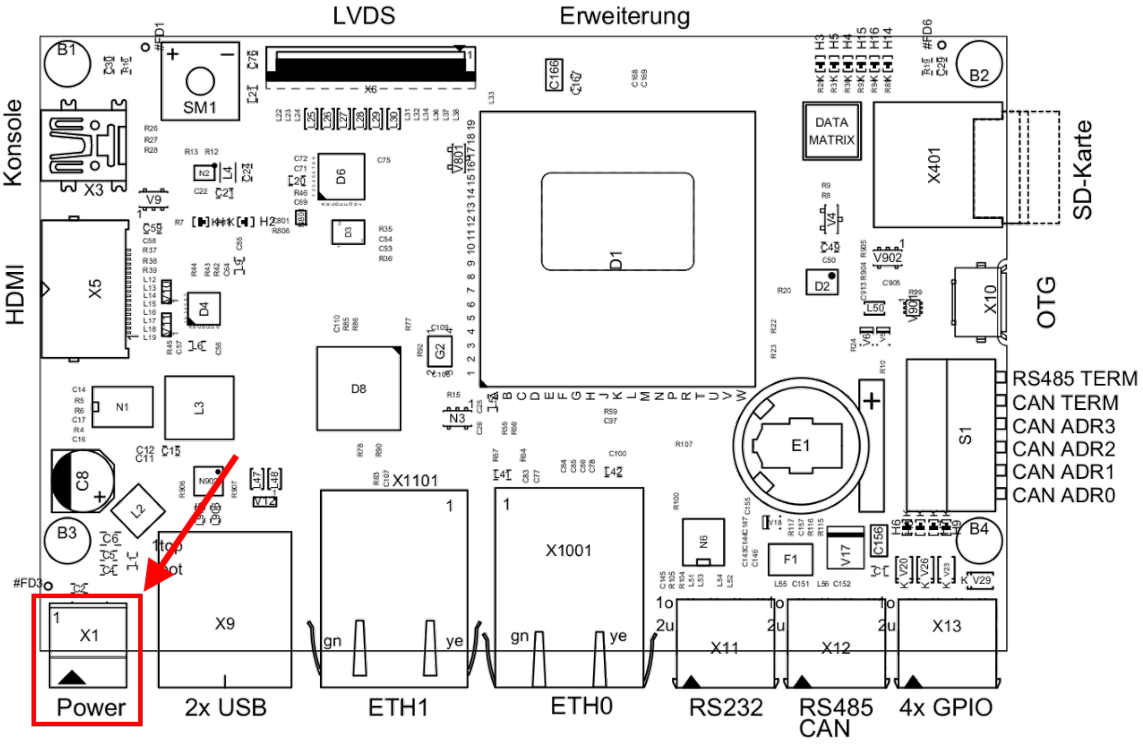
Getting Started with i.MX8MM

This guide is intended for first time users of the i.MX8MM Demo Kits and provides help in getting the board up and running. Each demo kit includes an i.MX8MM board and accessories like a power supply and a USB-to-Serial converter.

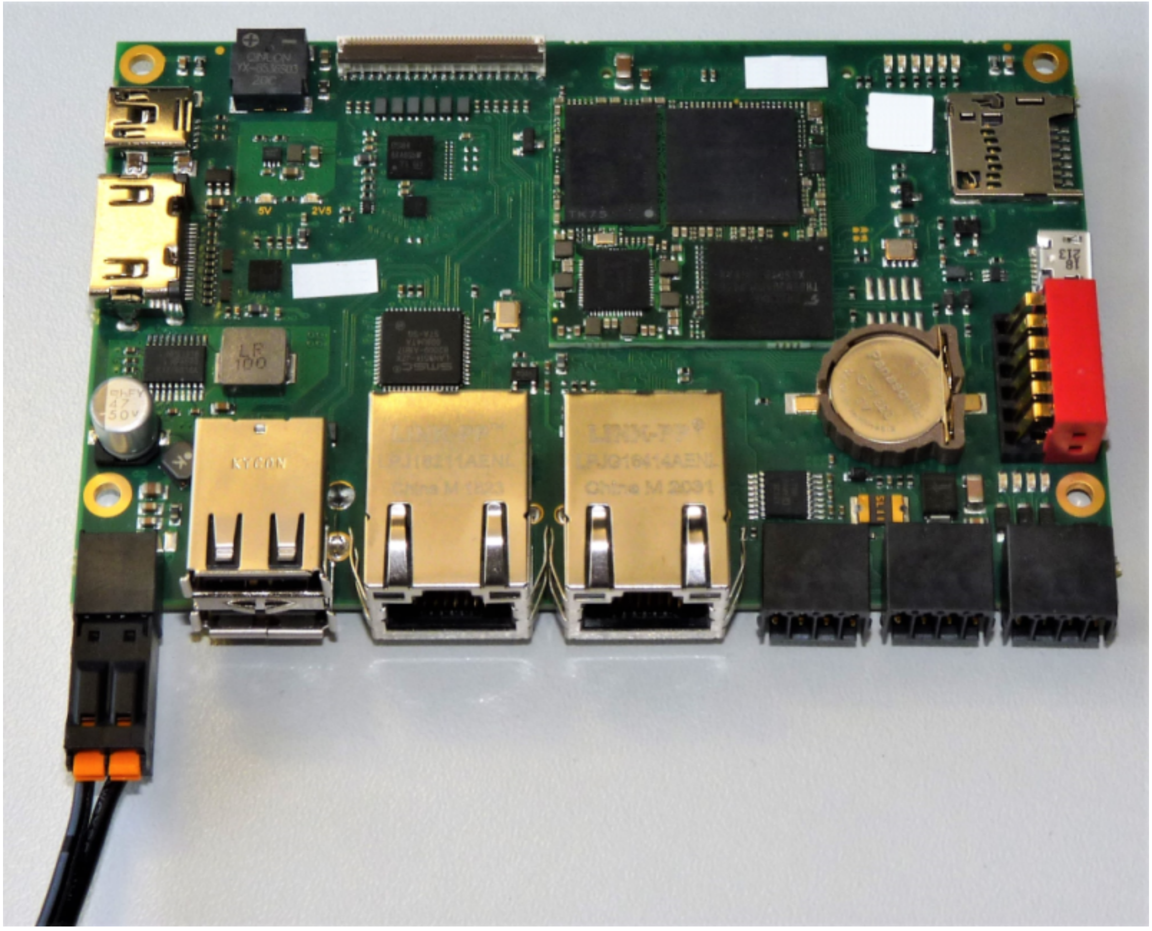
Connecting Power

Connect the power supply that came with the Demo Kit and plug it into the X1 socket on the board. Do not connect the power supply adapter to the mains yet. The power socket on the board is next to the two USB ports and has only two pins.

The following schematic shows the location of the power connector X1 on the BL i.MX8MM board.



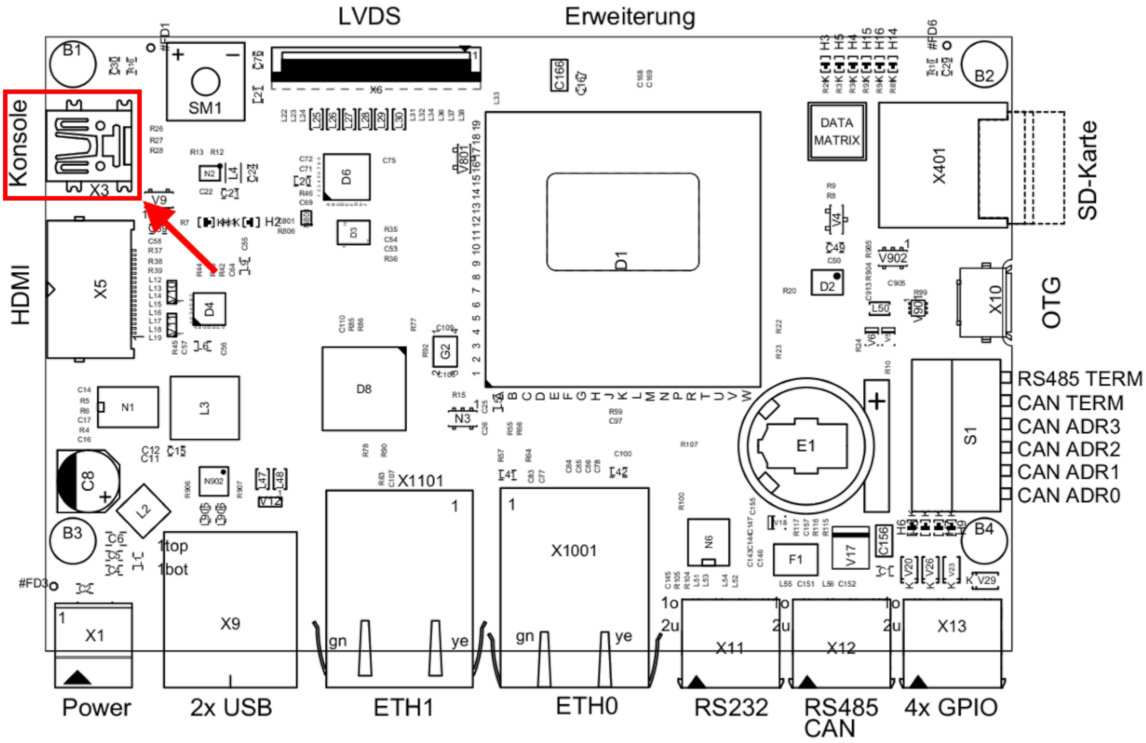
The following image shows the power connection done with an actual BL i.MX8MM board.



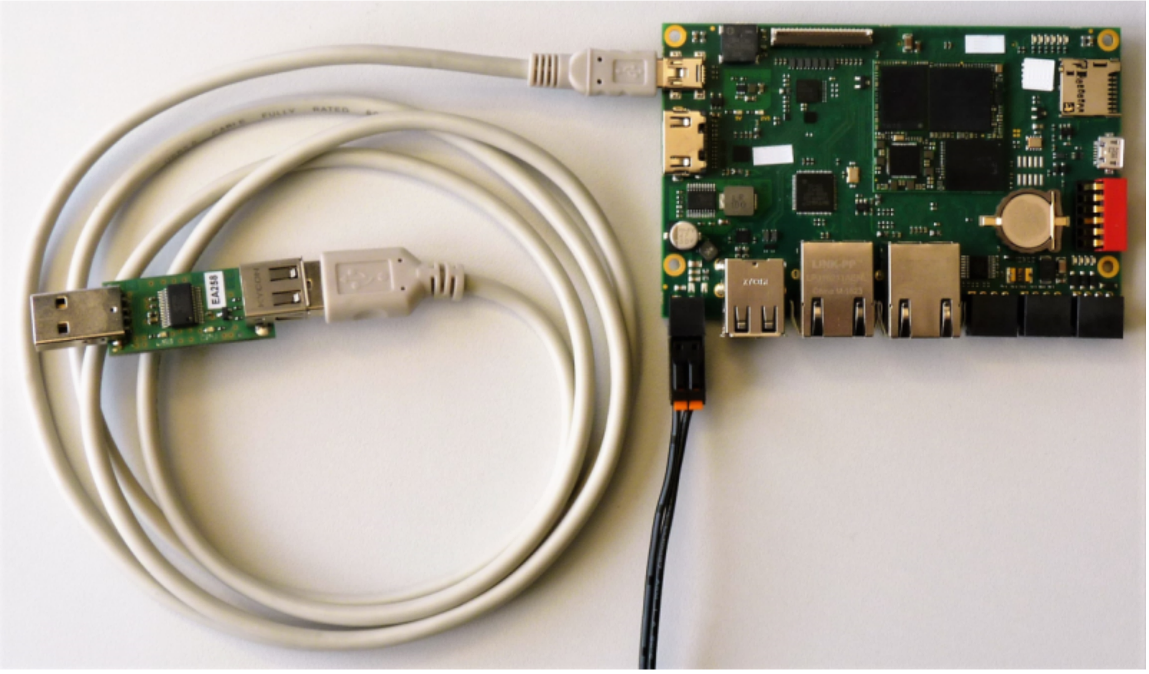
Connecting the Debug Cable

Now take the USB cable from the box and plug the USB-Mini plug into the USB-Mini port, named X3, at the side of the board next to the HDMI port. This connector is *not really a USB port*, but rather a *serial port* which allows the user access to UART 3 (Debug Console) and to communicate with the operating system later on without the need for a network setup. The other end of the cable (USB type A) has to be plugged into the *USB-to-Serial adapter* which came with the Demo Kit. The other end of the USB-to-Serial adapter can now be plugged into a computer.

The following image shows the location of the debug port (X3) on the BL i.MX8MM board.



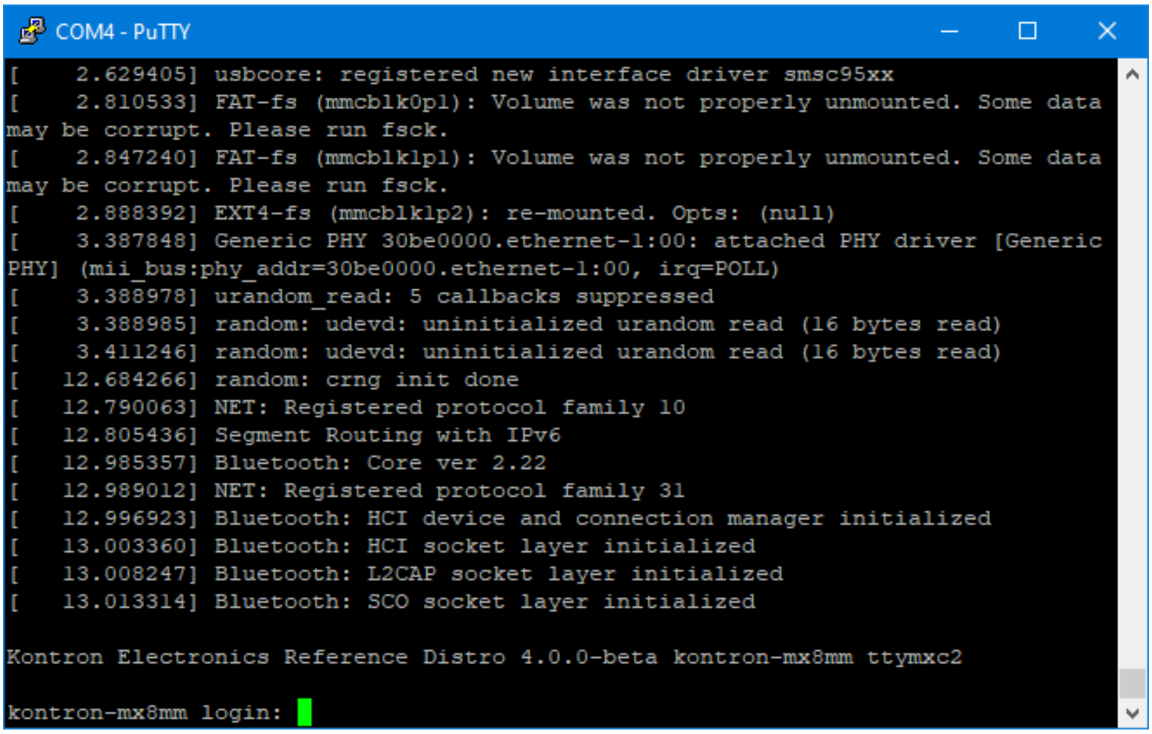
The following image shows the BL i.MX8MM board with the debug cable attached as well as the power cable hooked up.



First Start

Before turning on the power for the BL i.MX8MM, make sure the USB-to-Serial adapter is plugged into the computer and that you have a terminal or console program running on the COM port the USB-to-Serial adapter is registered at. In Linux this will usually be a *ttyUSB* and in Windows a *COM port* device.

Once this is set up, the board can be powered on and text should appear in the terminal program, which means the operating system is starting. After a short wait the text output should stop and a *login prompt* is visible. The image below shows an example what should be shown in the terminal program. The login prompt is right at the end of the screen.



```
[    2.629405] usbcore: registered new interface driver smsc95xx
[    2.810533] FAT-fs (mmcblk0p1): Volume was not properly unmounted. Some data
may be corrupt. Please run fsck.
[    2.847240] FAT-fs (mmcblk1p1): Volume was not properly unmounted. Some data
may be corrupt. Please run fsck.
[    2.888392] EXT4-fs (mmcblk1p2): re-mounted. Opts: (null)
[    3.387848] Generic PHY 30be0000.ethernet-1:00: attached PHY driver [Generic
PHY] (mii_bus:phy_addr=30be0000.ethernet-1:00, irq=POLL)
[    3.388978] urandom_read: 5 callbacks suppressed
[    3.388985] random: udevd: uninitialized urandom read (16 bytes read)
[    3.411246] random: udevd: uninitialized urandom read (16 bytes read)
[   12.684266] random: crng init done
[   12.790063] NET: Registered protocol family 10
[   12.805436] Segment Routing with IPv6
[   12.985357] Bluetooth: Core ver 2.22
[   12.989012] NET: Registered protocol family 31
[   12.996923] Bluetooth: HCI device and connection manager initialized
[   13.003360] Bluetooth: HCI socket layer initialized
[   13.008247] Bluetooth: L2CAP socket layer initialized
[   13.013314] Bluetooth: SCO socket layer initialized

Kontron Electronics Reference Distro 4.0.0-beta kontron-mx8mm ttymxc2

kontron-mx8mm login: 
```

To log in use the user `root` and no password is needed. Once logged in there are many things that can be done with the system, but for a start why not try to turn the digital output 1 (DIO1) on and off. The pre-installed system should have the libgpiod library and its service programs already installed, this means we can use a program called `gpioset` to change the state of DIO1.

For more information on this topic, see the "Main Documentation" here or see section "Using the System" for i.MX8MM based boards.

Configure DIO1 as an output and setting it to 1 (On).

```
root@kontron-mx8mm:/# gpioset gpiochip0 3=1
```

The onboard LED for DIO1 should light up and signal that its output is now active. With this one command the DIO1's corresponding GPIO was configured as an output and then set to 1 in one go.

Turning DIO1 off can be done by simply setting it to 0 with the following command:

```
root@kontron-mx8mm:/# gpioset gpiochip0 3=0
```

The LED for DIO1 should now be off again, meaning the output is no longer active.

This concludes the getting started guide for the BL i.MX8MM board. As mentioned above, also see other sections of this online documentation to learn more about other connection and communication options and how to setup your own Yocto Linux build system to create your own OS.

Troubleshooting

The BL i.MX8MM board in the Demo Kit should have an OS pre-installed on the internal eMMC memory and is ready to go. If this is not the case, you can still get the board working by making an SD card with a pre-build image from our server.

Download this "image" for example to get a base system. Unpack the file and write the *.wic file onto a blank SD card. If you are working under Windows, you might have to rename the file extension from* .wic to *.img to be able to use one of the many SD card writing programs available. Insert the SD card into the BL i.MX8MM boards SD card slot and plug in the power supply. The board should now boot from the SD card automatically and the above mentioned text should appear in the terminal program. You should now be able to log in and try the GPIO example above.

Building your own system

If you want to build your own system for the BL i.MX8MM, you have to use the Yocto based build system to create your own system image. Below you will find a short guide on how to create such a system image, but only the very basic steps are shown. For more information on the Yocto Project visit the projects website where you also find the reference manual.

Installing Prerequisites

As you start from scratch, it is necessary to install some prerequisites on the development PC. First update the package index:

```
sudo apt update
```

Now install these packages:

```
sudo apt install git-core gcc g++ python gawk chrpath texinfo
libsdl1.2-dev gdb-multiarch gcc-multilib g++-multilib
```

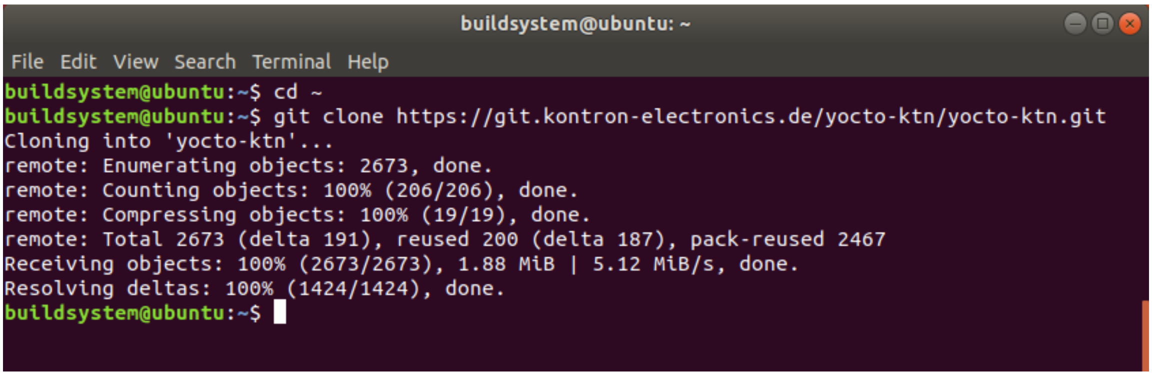
Also see the official Yocto docs for additional packages, that might be needed.

Cloning the Core Repository (yocto-ktn)

To get started you need to clone the core repository to a local folder on the development PC. This repository includes everything needed to make your own system and system image. The drive where the repository is cloned to needs to have at least 50 GB or more space available for extra files that will be downloaded and created during the build process.

```
cd ~
git clone https://git.kontron-electronics.de/sw/yocto/yocto-ktn.git
```

You should see something like this in your terminal:



```
buildsystem@ubuntu:~$ cd ~
buildsystem@ubuntu:~$ git clone https://git.kontron-electronics.de/yocto-ktn/yocto-ktn.git
Cloning into 'yocto-ktn'...
remote: Enumerating objects: 2673, done.
remote: Counting objects: 100% (206/206), done.
remote: Compressing objects: 100% (19/19), done.
remote: Total 2673 (delta 191), reused 200 (delta 187), pack-reused 2467
Receiving objects: 100% (2673/2673), 1.88 MiB | 5.12 MiB/s, done.
Resolving deltas: 100% (1424/1424), done.
buildsystem@ubuntu:~$
```

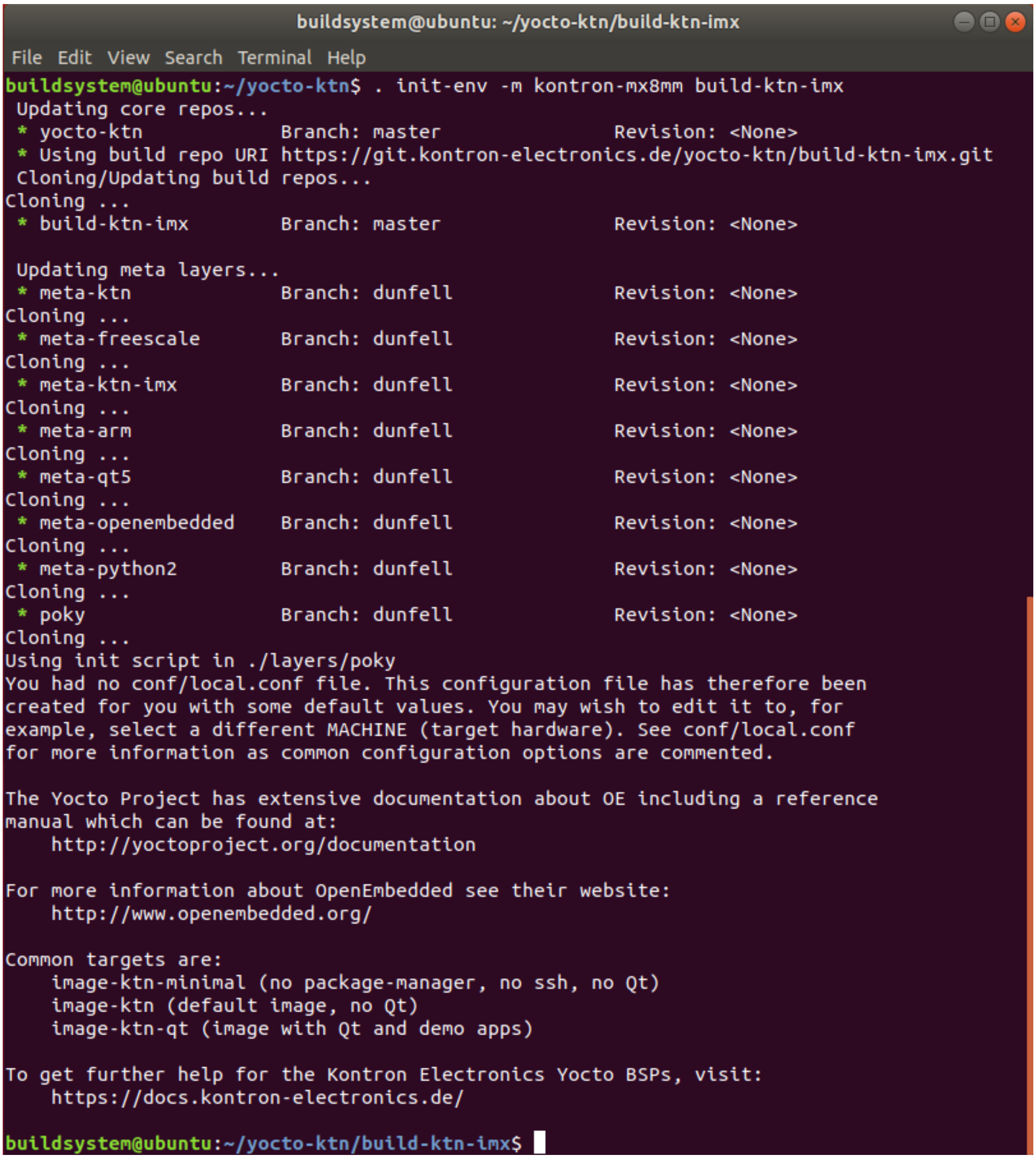
Initializing the Build Environment

Before you can use the build system, it has to be initialized with the parameters for the target machine. You have to change the directory to `yocto-ktn` and then initialize it for the i.MX8MM machine.

```
cd ~/yocto-ktn
. init-env -r latest -m kontron-mx8mm build-ktn-imx
```

Don't miss the dot (= `source` command) at the beginning of the second command or the initialization won't work. Also note the `-r` option, which instructs the initialization script to apply and load the latest release version, which is not necessarily the newest development version. Omit this option if you want to work with the newest development version of the BSP, but this is not recommended.

During the initialization of the Yocto build system, all needed meta data is downloaded from the community servers and the Kontron server automatically. At the end of the output, the command will list three common targets or system images you can create.


buildsystem@ubuntu: ~/yocto-ktn/build-ktn-imx
File Edit View Search Terminal Help

```
buildsystem@ubuntu:~/yocto-ktn$ . init-env -m kontron-mx8mm build-ktn-imx
 Updating core repos...
 * yocto-ktn              Branch: master              Revision: <None>
 * Using build repo URI https://git.kontron-electronics.de/yocto-ktn/build-ktn-imx.git
 Cloning/Updating build repos...
Cloning ...
 * build-ktn-imx          Branch: master              Revision: <None>

 Updating meta layers...
 * meta-ktn               Branch: dunfell             Revision: <None>
Cloning ...
 * meta-freescale         Branch: dunfell             Revision: <None>
Cloning ...
 * meta-ktn-imx           Branch: dunfell             Revision: <None>
Cloning ...
 * meta-arm               Branch: dunfell             Revision: <None>
Cloning ...
 * meta-qt5               Branch: dunfell             Revision: <None>
Cloning ...
 * meta-openembedded      Branch: dunfell             Revision: <None>
Cloning ...
 * meta-python2           Branch: dunfell             Revision: <None>
Cloning ...
 * poky                   Branch: dunfell             Revision: <None>
Cloning ...
Using init script in ./layers/poky
You had no conf/local.conf file. This configuration file has therefore been
created for you with some default values. You may wish to edit it to, for
example, select a different MACHINE (target hardware). See conf/local.conf
for more information as common configuration options are commented.

The Yocto Project has extensive documentation about OE including a reference
manual which can be found at:
    http://yoctoproject.org/documentation

For more information about OpenEmbedded see their website:
    http://www.openembedded.org/

Common targets are:
    image-ktn-minimal (no package-manager, no ssh, no Qt)
    image-ktn (default image, no Qt)
    image-ktn-qt (image with Qt and demo apps)

To get further help for the Kontron Electronics Yocto BSPs, visit:
    https://docs.kontron-electronics.de/

buildsystem@ubuntu:~/yocto-ktn/build-ktn-imx$ 
```

Building the first System Image

To build your first system image you will use `bitbake` to build it, but be careful as this can take a long time to complete.

Step 1: Accept the freescale/NXP EULA in the local.conf file

```
cd conf
nano local.conf
```

Remove the *hash (#)* in front of the line `ACCEPT_FSL_EULA = "1"` and save the change by pressing *CTRL+O* then *Enter* and then press *CTRL+X* to leave the editor. Go one folder up.

```
cd ..
```

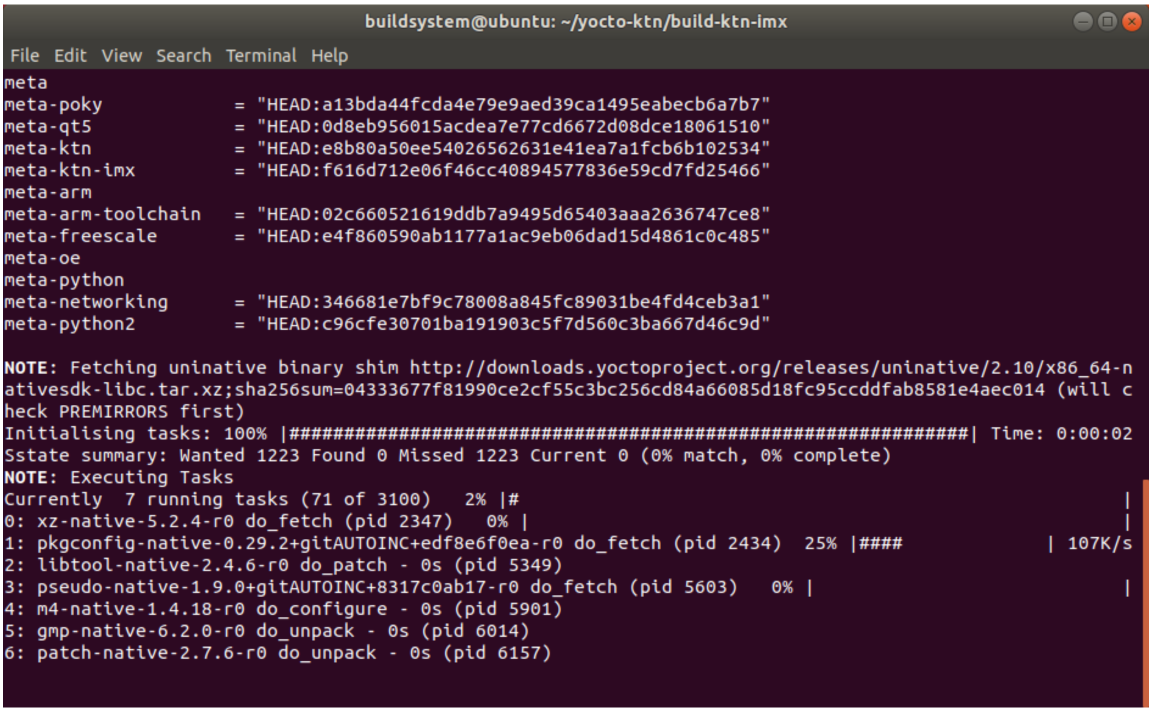
Step 2: Run bitbake to create the default image `image-ktn`

Demand of Resources

Please note, that building from scratch can take a long time (several hours!) and needs a lot of disk space and RAM! To build as much as possible even when a recipe fails you can use the `-k` option.

```
bitbake image-ktn -k
```

Now it is time to wait.



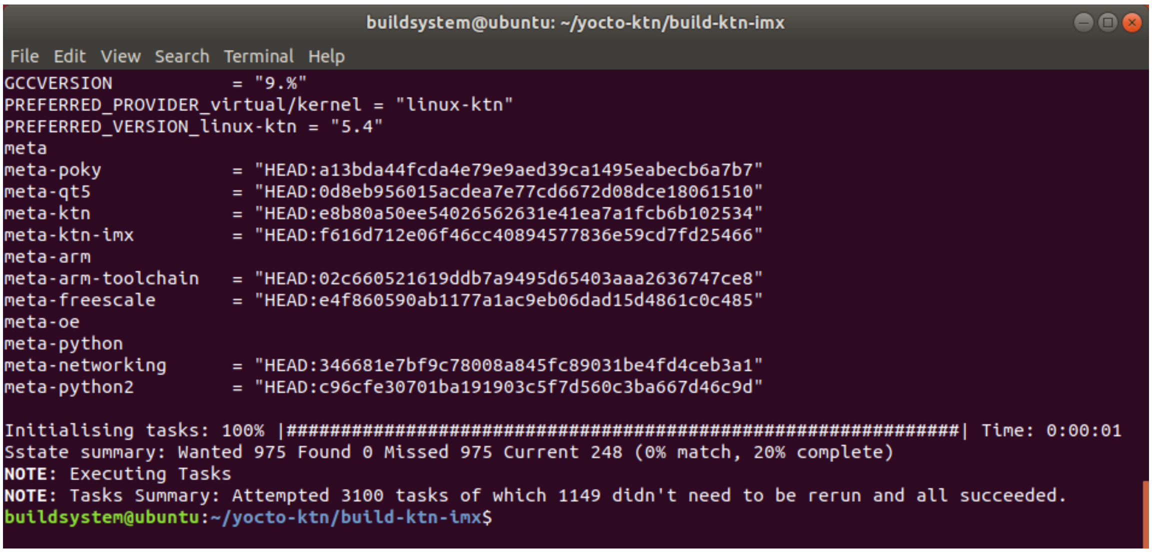
buildsystem@ubuntu: ~/yocto-ktn/build-ktn-imx
File Edit View Search Terminal Help

```
meta
meta-poky            = "HEAD:a13bda44fcda4e79e9aed39ca1495eabecb6a7b7"
meta-qt5             = "HEAD:0d8eb956015acdea7e77cd6672d08dce18061510"
meta-ktn             = "HEAD:e8b80a50ee54026562631e41ea7a1fcb6b102534"
meta-ktn-imx         = "HEAD:f616d712e06f46cc40894577836e59cd7fd25466"
meta-arm
meta-arm-toolchain   = "HEAD:02c660521619ddb7a9495d65403aaa2636747ce8"
meta-freescale       = "HEAD:e4f860590ab1177a1ac9eb06dad15d4861c0c485"
meta-oe
meta-python
meta-networking      = "HEAD:346681e7bf9c78008a845fc89031be4fd4ceb3a1"
meta-python2         = "HEAD:c96cfe30701ba191903c5f7d560c3ba667d46c9d"

NOTE: Fetching uninative binary shim http://downloads.yoctoproject.org/releases/uninative/2.10/x86_64-n
ativesdk-libc.tar.xz;sha256sum=04333677f81990ce2cf55c3bc256cd84a66085d18fc95ccddfab8581e4aec014 (will c
heck PREMIRRORS first)
Initialising tasks: 100% |##############################################################| Time: 0:00:02
Sstate summary: Wanted 1223 Found 0 Missed 1223 Current 0 (0% match, 0% complete)
NOTE: Executing Tasks
Currently  7 running tasks (71 of 3100)   2% |#                                                      |
0: xz-native-5.2.4-r0 do_fetch (pid 2347)   0% |                                                     |
1: pkgconfig-native-0.29.2+gitAUTOINC+edf8e6f0ea-r0 do_fetch (pid 2434)  25% |####             | 107K/s
2: libtool-native-2.4.6-r0 do_patch - 0s (pid 5349)
3: pseudo-native-1.9.0+gitAUTOINC+8317c0ab17-r0 do_fetch (pid 5603)   0% |                           |
4: m4-native-1.4.18-r0 do_configure - 0s (pid 5901)
5: gmp-native-6.2.0-r0 do_unpack - 0s (pid 6014)
6: patch-native-2.7.6-r0 do_unpack - 0s (pid 6157)
```

Checking the Build

Once the building process completes, you have to check if there were any error. Usually *bitbake* prints out information about the build process during and at the end when all steps have been completed. If there is any mention of an *error* or a *failed package* that could not be retrieved or compiled, try to run the build process again with the same command as before and see if it completes now.

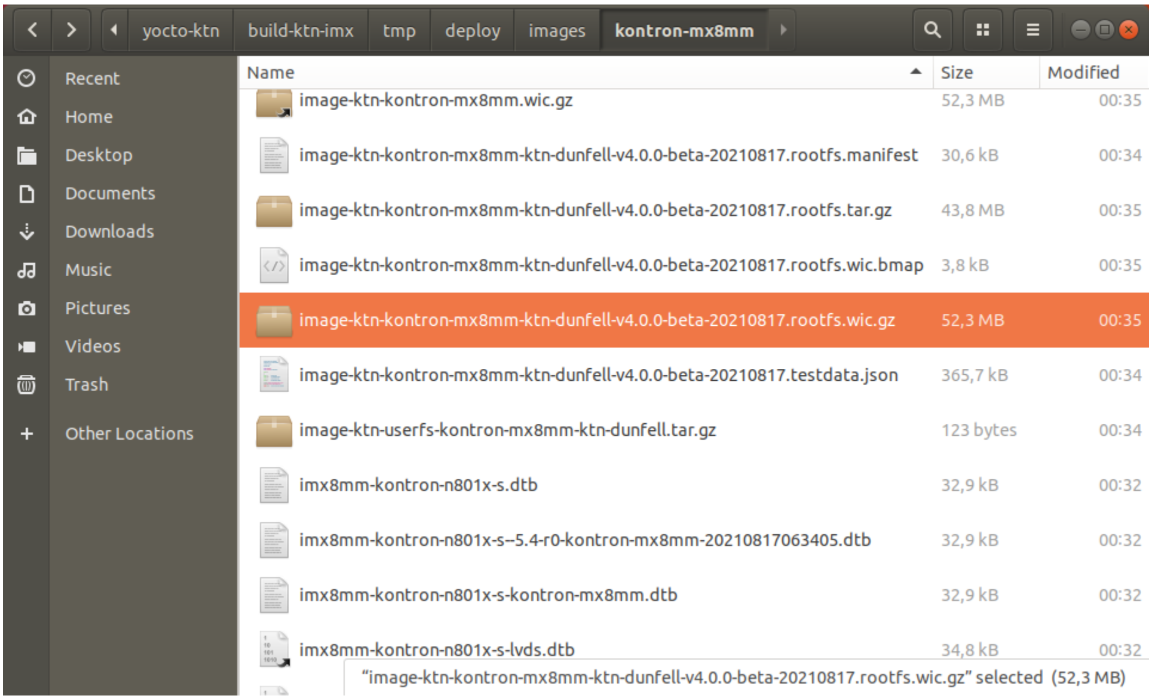
If there were no errors, you have successfully compiled your first system image.



```
GCCVERSION            = "9.%"
PREFERRED_PROVIDER_virtual/kernel = "linux-ktn"
PREFERRED_VERSION_linux-ktn = "5.4"
meta
meta-poky             = "HEAD:a13bda44fcda4e79e9aed39ca1495eabecb6a7b7"
meta-qt5              = "HEAD:0d8eb956015acdea7e77cd6672d08dce18061510"
meta-ktn              = "HEAD:e8b80a50ee54026562631e41ea7a1fcb6b102534"
meta-ktn-imx          = "HEAD:f616d712e06f46cc40894577836e59cd7fd25466"
meta-arm
meta-arm-toolchain    = "HEAD:02c660521619ddb7a9495d65403aaa2636747ce8"
meta-freescale        = "HEAD:e4f860590ab1177a1ac9eb06dad15d4861c0c485"
meta-oe
meta-python
meta-networking       = "HEAD:346681e7bf9c78008a845fc89031be4fd4ceb3a1"
meta-python2          = "HEAD:c96cfe30701ba191903c5f7d560c3ba667d46c9d"

Initialising tasks: 100% |##############################################################| Time: 0:00:01
Sstate summary: Wanted 975 Found 0 Missed 975 Current 248 (0% match, 20% complete)
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 3100 tasks of which 1149 didn't need to be rerun and all succeeded.
buildsystem@ubuntu:~/yocto-ktn/build-ktn-imx$
```

Booting the System Image

The newly compiled system image can be found in the folder "*yocto-ktn/build-ktn-imx/tmp/deploy/images/kontron-mx8mm/*". The image file, usually named *image-ktn-kontron-mx8mm-ktn-YOCTORELEASE-VERSION-DATE.rootfs.wic.gz*, is a compressed Gzip file and needs to be unpacked before it can be written to an SD card. Furthermore, the file type is called *.wic* which is an SD card image. If you are working under Windows for example, you can rename the file extension to *.img* which can make handling the file easier. The screenshot below shows the location of the image file.



Name	Size	Modified
image-ktn-kontron-mx8mm.wic.gz	52,3 MB	00:35
image-ktn-kontron-mx8mm-ktn-dunfell-v4.0.0-beta-20210817.rootfs.manifest	30,6 kB	00:34
image-ktn-kontron-mx8mm-ktn-dunfell-v4.0.0-beta-20210817.rootfs.tar.gz	43,8 MB	00:35
image-ktn-kontron-mx8mm-ktn-dunfell-v4.0.0-beta-20210817.rootfs.wic.bmap	3,8 kB	00:35
image-ktn-kontron-mx8mm-ktn-dunfell-v4.0.0-beta-20210817.rootfs.wic.gz	52,3 MB	00:35
image-ktn-kontron-mx8mm-ktn-dunfell-v4.0.0-beta-20210817.testdata.json	365,7 kB	00:34
image-ktn-userfs-kontron-mx8mm-ktn-dunfell.tar.gz	123 bytes	00:34
imx8mm-kontron-n801x-s.dtb	32,9 kB	00:32
imx8mm-kontron-n801x-s--5.4-r0-kontron-mx8mm-20210817063405.dtb	32,9 kB	00:32
imx8mm-kontron-n801x-s-kontron-mx8mm.dtb	32,9 kB	00:32
imx8mm-kontron-n801x-s-lvds.dtb	34,8 kB	00:32

“image-ktn-kontron-mx8mm-ktn-dunfell-v4.0.0-beta-20210817.rootfs.wic.gz” selected (52,3 MB)

Once the SD card is inserted into the BL i.MX8MM, you can go to the top of this guide and follow the steps there to see your own system booting off of the SD card.

Modifying the System

After creating a system image for the BL i.MX8MM the next step is to make changes to the BSP. The following steps will show how to create your own layer and how to add it to the BSP so it gets included in the bitbake build process. This new layer will add additional packages to the system image, which will be needed for some examples in later sections.

Of course it is possible to modify the BSP directly and make changes to the files which are already there, but if there is a problem it can be very difficult to get help and if you want to start from scratch all changes are lost. This is why for this guide we utilize Yocto's layering system. For more detailed information about layers and how to create them visit the Yocto documentation.

Creating a new Layer

Using a separate layer you will add Python 3 and other Python 3 modules to the system image, which are needed later for some examples to finish off this guide, but could also be useful in your own projects.

Open a terminal window and go to the *yocto-ktn* folder. In this folder initialize the build environment if this is not already the case from the previous section.

```
cd ~/yocto-ktn
. init-env -r latest -m kontron-mx8mm build-ktn-imx
```

Then go back to your home folder.

```
cd ~
```

Now create a new folder which will hold your layers and enter the folder.

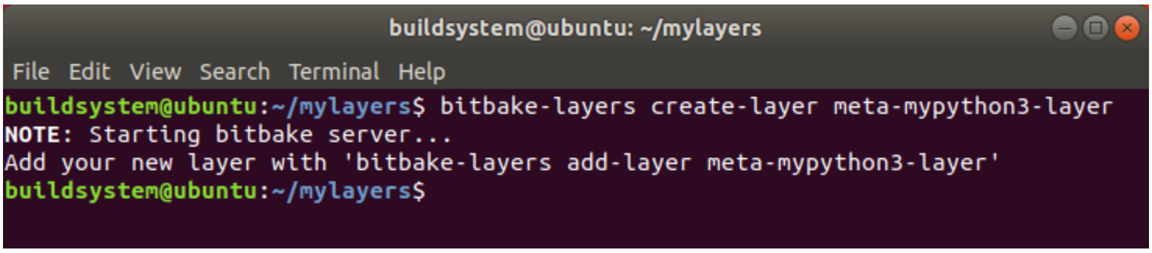
```
mkdir mylayers
cd mylayers
```

Now use bitbake to create a new layer together with a basic layer folder structure.

```
bitbake-layers create-layer meta-mypython3-layer
```

Note: The layer name *meta-mypython3-layer* is just an example, you can give your layer a different name. However the following examples and commands in this guide will use this layer name from here on. If you choose a different name, remember to adapt the shown commands.

In your terminal you should see something similar as shown in the following image:


buildsystem@ubuntu: ~/mylayers
File Edit View Search Terminal Help

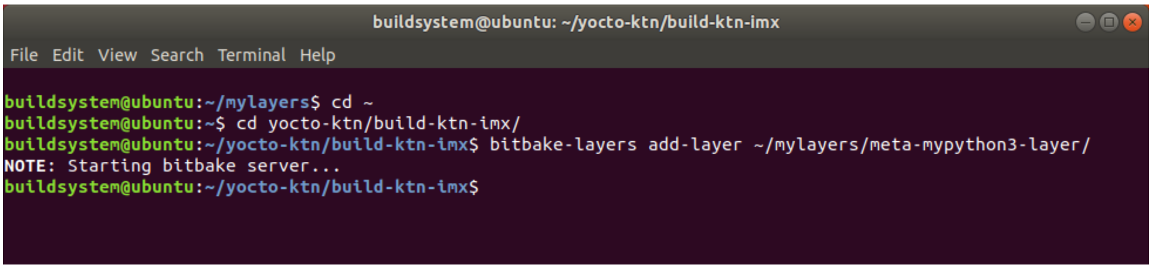
```
buildsystem@ubuntu:~/mylayers$ bitbake-layers create-layer meta-mypython3-layer
NOTE: Starting bitbake server...
Add your new layer with 'bitbake-layers add-layer meta-mypython3-layer'
buildsystem@ubuntu:~/mylayers$
```

Once the layer has been created, you will find a new folder in your *mylayers* folder named *meta-mypython3-layer* which contains a basic layer structure and the needed files to make it work. Feel free to explore the files and the folders inside.

In the next step this new layer is added to the build system and becomes part of your system image. Go back to the build folder and run bitbake again to add your layer like this:

```
cd ~
cd yocto-ktn/build-ktn-imx/
bitbake-layers add-layer ~/mylayers/meta-mypython3-layer/
```

The result should look like this:

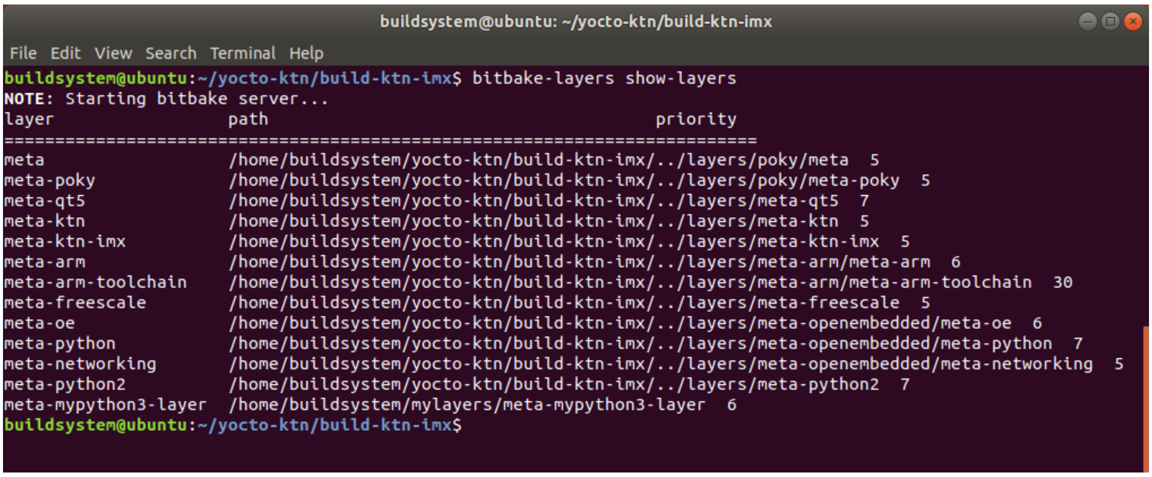

buildsystem@ubuntu: ~/yocto-ktn/build-ktn-imx
File Edit View Search Terminal Help

```
buildsystem@ubuntu:~/mylayers$ cd ~
buildsystem@ubuntu:~$ cd yocto-ktn/build-ktn-imx/
buildsystem@ubuntu:~/yocto-ktn/build-ktn-imx$ bitbake-layers add-layer ~/mylayers/meta-mypython3-layer/
NOTE: Starting bitbake server...
buildsystem@ubuntu:~/yocto-ktn/build-ktn-imx$
```

You can verify that your layer has been added to the system by issuing this command:

```
bitbake-layers show-layers
```

You should now see all active layers. Your layer should be at the bottom of the list, showing it's name and path.

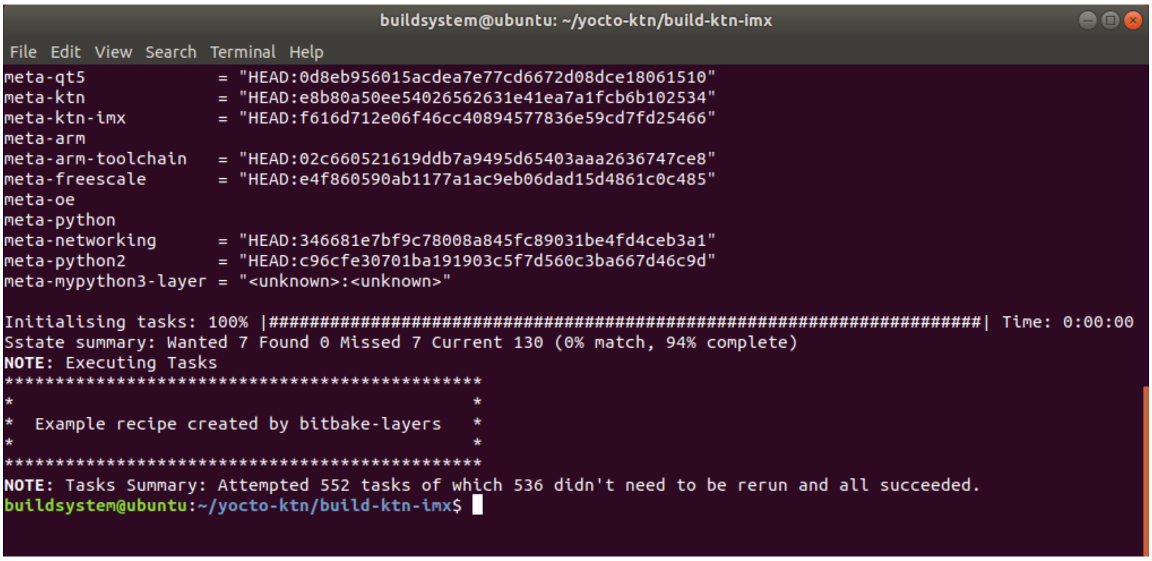

buildsystem@ubuntu: ~/yocto-ktn/build-ktn-imx
File Edit View Search Terminal Help

```
buildsystem@ubuntu:~/yocto-ktn/build-ktn-imx$ bitbake-layers show-layers
NOTE: Starting bitbake server...
layer                 path                                      priority
==========================================================================
meta                  /home/buildsystem/yocto-ktn/build-ktn-imx/../layers/poky/meta  5
meta-poky             /home/buildsystem/yocto-ktn/build-ktn-imx/../layers/poky/meta-poky  5
meta-qt5              /home/buildsystem/yocto-ktn/build-ktn-imx/../layers/meta-qt5  7
meta-ktn              /home/buildsystem/yocto-ktn/build-ktn-imx/../layers/meta-ktn  5
meta-ktn-imx          /home/buildsystem/yocto-ktn/build-ktn-imx/../layers/meta-ktn-imx  5
meta-arm              /home/buildsystem/yocto-ktn/build-ktn-imx/../layers/meta-arm/meta-arm  6
meta-arm-toolchain    /home/buildsystem/yocto-ktn/build-ktn-imx/../layers/meta-arm/meta-arm-toolchain  30
meta-freescale        /home/buildsystem/yocto-ktn/build-ktn-imx/../layers/meta-freescale  5
meta-oe               /home/buildsystem/yocto-ktn/build-ktn-imx/../layers/meta-openembedded/meta-oe  6
meta-python           /home/buildsystem/yocto-ktn/build-ktn-imx/../layers/meta-openembedded/meta-python  7
meta-networking       /home/buildsystem/yocto-ktn/build-ktn-imx/../layers/meta-openembedded/meta-networking  5
meta-python2          /home/buildsystem/yocto-ktn/build-ktn-imx/../layers/meta-python2  7
meta-mypython3-layer  /home/buildsystem/mylayers/meta-mypython3-layer  6
buildsystem@ubuntu:~/yocto-ktn/build-ktn-imx$
```

To verify that the layer is recognized by the build system, you can build the *example* recipe which was automatically created alongside the new layer. Run the following command to build the *example*:

```
bitbake example
```

Right now nothing is really build as the *example* only includes instructions to display text during the build process, but now it is clear the layer works and is included in the build process correctly.


buildsystem@ubuntu: ~/yocto-ktn/build-ktn-imx
File Edit View Search Terminal Help

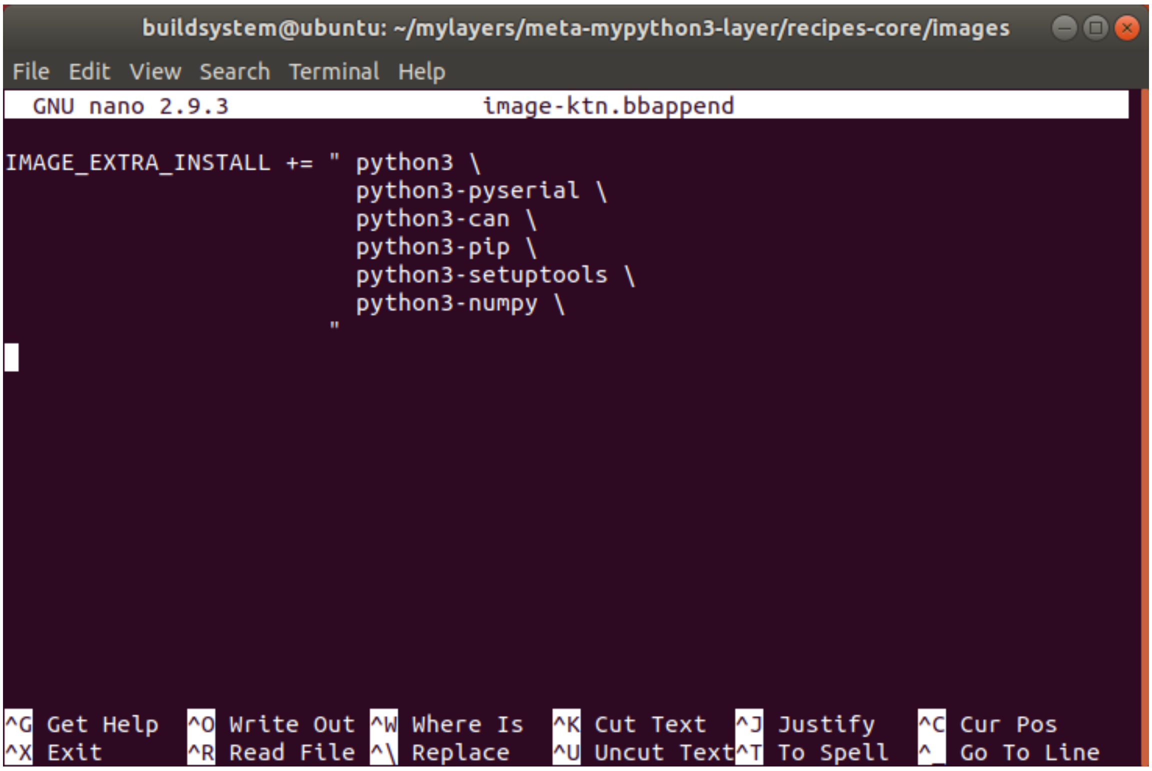
```
meta-qt5             = "HEAD:0d8eb956015acdea7e77cd6672d08dce18061510"
meta-ktn             = "HEAD:e8b80a50ee54026562631e41ea7a1fcb6b102534"
meta-ktn-imx         = "HEAD:f616d712e06f46cc40894577836e59cd7fd25466"
meta-arm
meta-arm-toolchain   = "HEAD:02c660521619ddb7a9495d65403aaa2636747ce8"
meta-freescale       = "HEAD:e4f860590ab1177a1ac9eb06dad15d4861c0c485"
meta-oe
meta-python
meta-networking      = "HEAD:346681e7bf9c78008a845fc89031be4fd4ceb3a1"
meta-python2         = "HEAD:c96cfe30701ba191903c5f7d560c3ba667d46c9d"
meta-mypython3-layer = "<unknown>:<unknown>"

Initialising tasks: 100% |######################################################################| Time: 0:00:00
Sstate summary: Wanted 7 Found 0 Missed 7 Current 130 (0% match, 94% complete)
NOTE: Executing Tasks
***********************************************
*                                             *
*  Example recipe created by bitbake-layers   *
*                                             *
***********************************************
NOTE: Tasks Summary: Attempted 552 tasks of which 536 didn't need to be rerun and all succeeded.
buildsystem@ubuntu:~/yocto-ktn/build-ktn-imx$ █
```

To get the build system to include Python 3 and other Python 3 modules in the system image, some changes are needed to make this work. Use the following commands to adapt your layer:

```
cd ~/mylayers/meta-mypython3-layer/
mv recipes-example recipes-core
cd recipes-core/
mv example images
cd images/
mv example_0.1.bb image-ktn.bbappend
nano image-ktn.bbappend
```

In the nano editor delete everything that is already there and replace it with this:

```
IMAGE_EXTRA_INSTALL += " python3 \
                         python3-pyserial \
                         python3-can \
                         python3-pip \
                         python3-setuptools \
                         python3-numpy \
                       "
```


buildsystem@ubuntu: ~/mylayers/meta-mypython3-layer/recipes-core/images
File Edit View Search Terminal Help

```
  GNU nano 2.9.3                    image-ktn.bbappend

IMAGE_EXTRA_INSTALL += " python3 \
                         python3-pyserial \
                         python3-can \
                         python3-pip \
                         python3-setuptools \
                         python3-numpy \
                       "

^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text^T To Spell    ^_ Go To Line
```

Save the changes with *CTRL+O* and *Enter* and then leave the editor with *CTRL+X*.

Explanation: In this Yocto system or BSP, the building of the system image and which packages go into it, is defined in the core recipes named *recipes-core/*

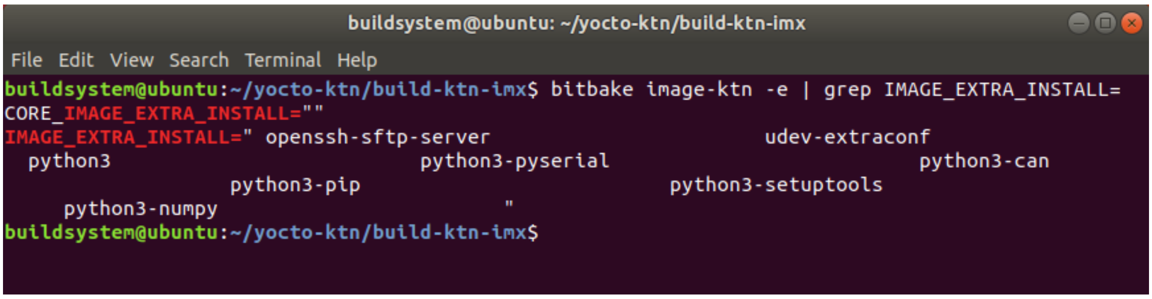
images. The *bb* file name used is *image-ktn.bb* and with all the above changes, your layer now extends (appends) this base recipe to also include Python 3 in the final system image.

If you want to include more or other Python 3 modules, visit the OpenEmbedded Layer Index and go to the *Recipes* tab. If you enter "python3-" into the search bar, all available Python 3 packages will be listed. However note, that you can only include packages which are available for the *Dunfell* and later releases of the Yocto Project. If a desired recipe requires a newer Yocto version then this recipe cannot be included in the system image at this point.

To check that the changes have been applied correctly, go back to the build folder and run bitbake again like this:

```
bitbake image-ktn -e | grep IMAGE_EXTRA_INSTALL=
```

The option `-e` instructs bitbake to only look at the global environment and which variables are defined. In conjunction with *grep* only the information for the variable *IMAGE_EXTRA_INSTALL* is shown. The `=` sign at the end helps to narrow down the search. You should see something like this:



buildsystem@ubuntu: ~/yocto-ktn/build-ktn-imx
File Edit View Search Terminal Help

```
buildsystem@ubuntu:~/yocto-ktn/build-ktn-imx$ bitbake image-ktn -e | grep IMAGE_EXTRA_INSTALL=
CORE_IMAGE_EXTRA_INSTALL=""
IMAGE_EXTRA_INSTALL=" openssh-sftp-server                       udev-extraconf
  python3                          python3-pyserial                          python3-can
                   python3-pip                          python3-setuptools
     python3-numpy                        "
buildsystem@ubuntu:~/yocto-ktn/build-ktn-imx$
```

Now that everything is ready, you can go ahead and build a new system image. Use bitbake as before:

```
bitbake image-ktn
```

The building of the new system image should go fairly fast, as most packages were build in previous steps. You should mainly see Python 3 packages and their dependencies being build. After the build process completes, retrieve the system image file, unpack it and write it to an SD card.

Once the SD card is done and inserted into the BL i.MX8MM, you can go to the top of this guide and follow the steps there to see this new system image booting.

Examples using Python 3

Now that the system image includes Python 3 and some additional modules to work with the BL i.MX8MMs hardware, why not try it out.

Serial Communication

In this example you will use the `pySerial` module to communicate with a computer. First wire up the RS232 port of the BL i.MX8MM to a computer directly if it has an RS232 port or via a USB-to-Serial adapter. You can find the connector pin-outs of the BL i.MX8MM in the board overview. Remember to switch the Rx and Tx lines when connecting the wires. Once done, turn on the BL i.MX8MM and log in.

Try to run Python 3 and print something to the console:

```
root@kontron-mx8mm:~# python3
Python 3.8.2 (default, Feb 25 2020, 10:39:28)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> print("Hello World!")
Hello World!
>>> quit()
root@kontron-mx8mm:~#
```

Now it's time to create a small Python 3 serial test program. Open the *nano* editor:

```
root@kontron-mx8mm:~# nano serial_test.py
```

and enter the following program code or download a copy of the file serial_test.py

```
#!/usr/bin/python3

import serial, time
# Configure the serial driver
ser = serial.Serial()
# Communicate via RS232
ser.port = "/dev/ttymxc0"
ser.baudrate = 115200
ser.bytesize = serial.EIGHTBITS
ser.parity = serial.PARITY_NONE
ser.stopbits = serial.STOPBITS_ONE

# Try to open the port
try:
    ser.open()
except Exception as e:
    print ("Error while opening the serial port: " + str(e))
    exit()

# If the port is open, write Hello World and wait
# for a reply which must be terminated by an LF.
if ser.isOpen():
    try:
        # Clear the buffers
        ser.flushInput()
        ser.flushOutput()
        while True:
            # Write to the serial port
            ser.write("Hello World!\n".encode())
            # Write it immediately
            ser.flush()
            print("Write data!")
            try:
                # Now wait for the other side to send something
                myvar = ser.read_until() # Wait for LF
            except:
                # If there was an error during receiving, print an
error
                myvar = b'Error receiving!'

            # Display the received data, which must be ASCII
characters in this example
            print("Read data! " + myvar.decode('ascii'))
            time.sleep(0.05)
        ser.close()
    except Exception as e1:
        print ("Error while communicating...:" + str(e1))
```

```
else:
    print ("Cannot open serial port!")
```

Save the changes with *CTRL+O* and *Enter* and then leave the editor with *CTRL+X*.

On the computer start a terminal program and open the serial port using these parameters:

- Baud rate: 115200
- Data bits: 8
- Stop bits: 1
- Parity bit: None

Run the Python 3 program on the BL i.MX8MM with this command:

```
python3 serial_test.py
```

The terminal windows on the computer should now display the text:

> Hello World!

While on the BL i.MX8MM the Python 3 program printed the text "*Write data!*" to the console and now waits for the computer to send something. If you now enter a text like "*Hello World 2!*" in the terminal program on the computer and send it to the BL i.MX8MM, the Python 3 program will print the following to the console:

> Read data! Hello World 2!

Note: When sending data from the computer to the BL i.MX8MM the last byte or character has to be a linefeed character, also referred to as LF, \n, ASCII decimal value 10 or hex value A. Some terminal or serial programs on the computer do this automatically, but there are exceptions which do not. If there is no linefeed character in the data sent to the BL i.MX8MM, the Python 3 program will wait indefinitely until the linefeed character arrives, therefore during this wait the program might appear frozen. There are other function in pySerial to receive data which do not wait (block the program), the function `read_until` was chosen in this example for practical reasons.

If you want to quit or terminate the program, press and hold the key combination *CTRL+C* until the program has ended. Feel free to modify the program or make your own and experiment with the serial port of the BL i.MX8MM sending and receiving data.

CAN-Bus Communication

This example focuses on the Python 3 `pyCan` module to communicate with another CAN-Bus node. The other device can be a BL i.MX8MM, one of our Raspberry Pi based products or a PC with a CAN-Bus adapter. The commands shown below are run on a device with Debian Linux and the `can-utils` package installed. If your setup differs from this, you will have to adapt the procedures below accordingly.

Both devices are wired together so that CAN High (CAN_H) from one device is connected to CAN High on the other device, the same is done with CAN Low (CAN_L) and Ground (GND). For the pin layout of the BL i.MX8MM have a look at the board overview section.

After turning both devices on and letting them boot, you can try to perform a simple CAN-Bus communication test, before moving on to the Python 3 example.

On the second device run the `candump` command on the connected CAN port, in this example *can0*, to listen to traffic on the CAN-Bus. Configure the *can0* interface with 125000 bps beforehand.

```
user@linux:~# candump can0
```

On the BL i.MX8MM configure the *can0* interface also with 125000 bps and then run the `cansend` command to send out a CAN message:

```
root@kontron-mx8mm: ifconfig can0 down
root@kontron-mx8mm: ip link set can0 type can bitrate 125000
root@kontron-mx8mm: ifconfig can0 up
root@kontron-mx8mm: cansend can0 123#DEADBEEF
```

On the second device the console should show the received message:

```
user@linux:~# candump can0
  can0  123   [4]  DE AD BE EF
```

If you get the same result, the communication works as expected, but you could now swap the commands and send a message from the second device to the BL i.MX8MM to test the reverse direction.

The next step is to build a Python 3 program which receives CAN messages and sends them back like an echo. However the message ID will be changed to make it easier to distinguish between the messages sent from the second device and those the BL i.MX8MM echoed back.

The following Python 3 program uses the *pyCan* or *can* module to receive and send messages via the BL i.MX8MM's CAN-Bus interface *can0*.

Download a copy of the file can_test.py.

This example program sends out one message at the start and then waits for incoming messages which it will then echo back to the CAN-Bus, but changing the message ID to 321h before doing so. To leave the program press *CTRL+C* while still messages are arriving otherwise simply reboot the device or turn off power. The function "*bus.recv()*" is a blocking call and waits until a new message arrives.

On the second device run the *candump* program again, if it is not already running, and configure the CAN interface with 125000 bps:

```
user@linux:~# candump can0
```

Configure the *can0* interface of the BL i.MX8MM and then run the Python 3 program like this:

```
root@kontron-mx8mm: ifconfig can0 down
root@kontron-mx8mm: ip link set can0 type can bitrate 125000
root@kontron-mx8mm: ifconfig can0 up
root@kontron-mx8mm: python3 can_test.py
First message sent on socketcan channel 'can0'
```

The second device should show in the console:

```
can0  234   [4]  DE AD BE EF
```

On the same device press *CTRL+C* to quit the *candump* program and instead run the *cangen* program to generate random CAN messages on the CAN-Bus continously. You can run the program like this:

```
user@linux:~# cangen can0 -g 1000
```

The parameter `-g 1000` means *gap* or *pause* for 1000 ms between messages.

The BL i.MX8MM starts to display a new message in the console every second which should look like this:

```
Message:
Timestamp: 1630326139.336095        ID: 04a6    S
DLC:  3    b5 4d 86                    Channel: can0
Echo message sent on socketcan channel 'can0'
Message:
Timestamp: 1630326140.335648        ID: 00e4    S
DLC:  0                                Channel: can0
Echo message sent on socketcan channel 'can0'
Message:
Timestamp: 1630326141.336454        ID: 0415    S
DLC:  8    dc fd 9c 71 77 d1 25 07     Channel: can0
Echo message sent on socketcan channel 'can0'
Message:
Timestamp: 1630326142.336584        ID: 047d    S
DLC:  8    f7 23 4c 43 20 78 70 0f     Channel: can0
Echo message sent on socketcan channel 'can0'
Message:
Timestamp: 1630326169.304173        ID: 0704    S
DLC:  8    09 18 4d 56 92 20 d2 2f     Channel: can0
Echo message sent on socketcan channel 'can0'
...
```

If you are connected to the second device via SSH or you are working in a terminal program on a desktop PC, you can open a second terminal window or a second SSH connection and start the *candump* program again in parallel to the *cangen* program. This way you can see the replies (echos) from the BL i.MX8MM coming back to the second device.

Example output from *candump* running in parallel (separate terminal window) to *cangen* on the second device:

```
user@linux:~# candump can0
  can0  4E1   [8]  11 61 62 62 76 25 4B 1D
  can0  321   [8]  11 61 62 62 76 25 4B 1D
  can0  1D6   [8]  7B B2 06 67 B5 3E CE 18
  can0  321   [8]  7B B2 06 67 B5 3E CE 18
  can0  06A   [1]  37
  can0  321   [1]  37
  can0  1BA   [8]  1B 34 16 41 46 FD D6 67
  can0  321   [8]  1B 34 16 41 46 FD D6 67
  can0  599   [4]  D8 2A 0B 61
  can0  321   [4]  D8 2A 0B 61
  can0  64A   [7]  A1 03 FB 7B 46 DF A9
  can0  321   [7]  A1 03 FB 7B 46 DF A9
  can0  386   [8]  13 1E A5 10 DF 43 F2 0D
  can0  321   [8]  13 1E A5 10 DF 43 F2 0D
  can0  357   [8]  B2 0B D8 3A 38 F0 06 3F
  can0  321   [8]  B2 0B D8 3A 38 F0 06 3F
...
```

First comes the CAN-Bus interface, then the message ID followed by the message DLC and the raw data in hex values. Every message always appears twice. The first message is sent by the second device and has a random message ID, for the second message the ID is always 321h. This message was echoed back from the BL i.MX8MM.

Writing the System Image to eMMC Memory

Danger

Since Release 5.0.0 this is no longer valid please refer to chapter Flash layout.

The BL i.MX8MM has an eMMC memory chip on the SoM, which can be used to boot a self made System Image and in the process freeing up the SD card slot which can then be used as data storage for example.

Note: If you received the BL i.MX8MM with a Demo Kit, then you should be good to go and you can go to the top of this guide and follow the steps there to get the device up and running. If you received or bought the BL i.MX8MM standalone, have a look at the next section.

Preparations

To write the System Image to the eMMC memory, a few things have to be prepared.

- Find and unpack your System Image as described here Booting the System Image further up in this guide.
- Place the extracted *wic* file on a FAT (FAT32) formatted USB stick. Consider renaming the file to something shorter, like rootfs.wic. This filename will used instead of the default name.
- Boot the BL i.MX8MM from SD card and insert the USB stick which contains the System Image (*wic*) file.
- The eMMC memory can be accessed through the device */dev/mmcblk0*.

Writing the System Image to eMMC Memory

Once the system is running and the USB stick is inserted into a USB port, there should be some text in the console that a USB stick was detected and which device it is.

```
root@kontron-mx8mm:~# [   87.081061] usb 1-1.4: new high-speed USB
device number 4 using ci_hdrc
[   87.385051] usb 1-1.4: device descriptor read/64, error -71
[   87.607943] usb-storage 1-1.4:1.0: USB Mass Storage device
detected
[   87.614512] scsi host0: usb-storage 1-1.4:1.0
[   88.638468] scsi 0:0:0:0: Direct-Access     Generic- Multiple
Reader  1.07 PQ: 0 ANSI: 4
[   88.648633] scsi 0:0:0:1: Direct-Access     Generic- MicroSD/
```

```
M2       1.08 PQ: 0 ANSI: 4
[   89.050075] sd 0:0:0:1: [sdb] 31116288 512-byte logical blocks:
(15.9 GB/14.8 GiB)
[   89.058307] sd 0:0:0:0: [sda] Attached SCSI removable disk
[   89.061203] sd 0:0:0:1: [sdb] Write Protect is off
[   89.071104] sd 0:0:0:1: [sdb] Write cache: disabled, read
cache: disabled, doesn't support DPO or FUA
[   89.113221]  sdb: sdb1
[   89.122718] sd 0:0:0:1: [sdb] Attached SCSI removable disk
```

If the USB stick was already plugged-in during boot, you can use the command `dmesg` to see the system messages or try the command `lsblk` to see if there is a new block device available and if it is already mounted.

```
root@kontron-mx8mm:~# lsblk
NAME         MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sdb            8:16   1  14.9G  0 disk
`-sdb1         8:17   1  14.9G  0 part /run/media/sdb1
mtdblock0     31:0    0     2M  0 disk
mmcblk0      179:0    0  29.1G  0 disk
|-mmcblk0p1  179:1    0  83.2M  0 part /run/media/mmcblk0p1
`-mmcblk0p2  179:2    0 664.3M  0 part
mmcblk0boot0 179:32   0     4M  1 disk
mmcblk0boot1 179:64   0     4M  1 disk
mmcblk1      179:96   0  14.9G  0 disk
|-mmcblk1p1  179:97   0  83.2M  0 part /run/media/mmcblk1p1
`-mmcblk1p2  179:98   0   356M  0 part /
```

In this example the USB stick was detected during boot and the first partition of the media was mounted to "*/run/media/sdb1*". Navigating in to this folder and listing the available files should show you the System Image (*wic*) file. If the mount point is different on your device, adjust the following commands accordingly.

```
root@kontron-mx8mm:~# cd /run/media/sdb1
root@kontron-mx8mm:/run/media/sdb1# ls -la
drwxrwx---    2 root     disk         32768 Jan  1  1970 .
drwxr-xr-x    5 root     root           100 Sep  1 09:08 ..
-rwxrwx---    1 root     disk     465599488 Aug 18 12:20 rootfs.wic
```

You can now write the System Image to the eMMC memory using the program *dd* and the device name */dev/mmcblk0*:

```
root@kontron-mx8mm:~# dd if=rootfs.wic of=/dev/mmcblk0 bs=1M
conv=sync
```

Now wait for a few minutes until the complete image has been transferred. Once this is done, *dd* will print out the amount of blocks read and written and you are back at the command prompt.

Unplug the USB stick and then reboot the BL i.MX8MM from SD card again, there is one more change needed.

```
root@kontron-mx8mm:~# reboot
```

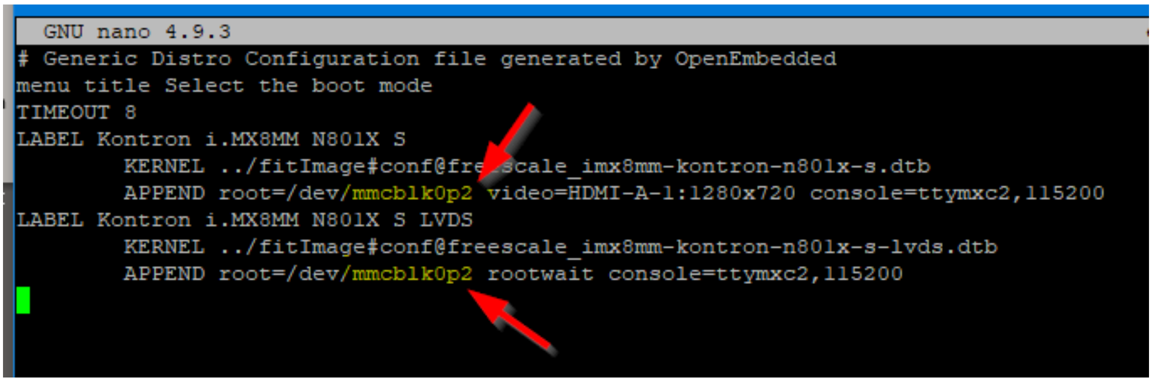
After the reboot check with `lsblk` that the FAT partition of the eMMC memory is mounted at */run/media/mmcblk0p1*.

```
NAME         MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
mtdblock0     31:0    0     2M  0 disk
mmcblk0      179:0    0  29.1G  0 disk
|-mmcblk0p1  179:1    0  83.2M  0 part /run/media/mmcblk0p1
`-mmcblk0p2  179:2    0 664.3M  0 part
mmcblk0boot0 179:32   0     4M  1 disk
mmcblk0boot1 179:64   0     4M  1 disk
mmcblk1      179:96   0  14.9G  0 disk
|-mmcblk1p1  179:97   0  83.2M  0 part /run/media/mmcblk1p1
`-mmcblk1p2  179:98   0   356M  0 part /
```

Enter the folder */run/media/mmcblk0p1* and then go into the folder *extlinux* and edit the file *extlinux.conf*:

```
root@kontron-mx8mm:~# cd /run/media/mmcblk0p1
root@kontron-mx8mm:~# cd extlinux
root@kontron-mx8mm:~# nano extlinux.conf
```

Change all references of *mmcblk1* to *mmcblk0*, there should be 2 places to edit.



```
  GNU nano 4.9.3
# Generic Distro Configuration file generated by OpenEmbedded
menu title Select the boot mode
TIMEOUT 8
LABEL Kontron i.MX8MM N801X S
        KERNEL ../fitImage#conf@freescale_imx8mm-kontron-n801x-s.dtb
        APPEND root=/dev/mmcblk0p2 video=HDMI-A-1:1280x720 console=ttymxc2,115200
LABEL Kontron i.MX8MM N801X S LVDS
        KERNEL ../fitImage#conf@freescale_imx8mm-kontron-n801x-s-lvds.dtb
        APPEND root=/dev/mmcblk0p2 rootwait console=ttymxc2,115200
```

Note: Do not delete the "*p2*" part from the device name, only change the *1* to *0*.

Press *CTRL+O* and *Enter* to save the changes and then *CTRL+X* to leave the editor. Everything is now ready. Power off the BL i.MX8MM and then take out the SD card and power the device back on. It should now boot from the onboard eMMC memory. You can go to the top of this guide and follow the steps there to see your system image booting from eMMC memory.

Troubleshooting:

If nothing happens and the BL i.MX8MM does not boot from eMMC, then probably the SPI NOR flash is not programed with a bootloader or the bootconfiguration fuses do not have the correct settings. Boot the BL i.MX8MM again from SD card and follow the steps from the next section.

Writing the Bootloader to NOR Flash

Info

Since Release 5.0.0 this is no longer necesarry please refer to chapter Flash layout.

This section is intended for users who have a single BL i.MX8MM and not the Demo Kit and want to be able to boot the device from eMMC memory instead, freeing up the SD card for other uses. To boot the BL i.MX8MM from eMMC a bootloader stored in the NOR flash of the SoM is needed, like *U-Boot*. In addition the bootconfig has to include booting from NOR flash, at least as fallback option. More information about the *bootconfig* can be found on the page Using the System.

First check if any action is needed. The `fwinfo` command prints out the current system status.

```
root@kontron-mx8mm:~# fwinfo
Fuses: SOM SerialNo: xxxxxxxx
Fuses: Board SerialNo: xxxxxxx
Fuses: Bootconfig: 0x18001410
Fuses: MAC1: xxxxxxxxxxxx
Fuses: MAC2: xxxxxxxxxxxx
Fuses: Lock: xxxxxxxxxxxx
Version: Firmware Version: xxxxxxxxxxxxxxxxx
Log: Devicetree info: xxxxxxxxxxxxxxx
Linux version xxxxxxxxxxxxxxx
U-Boot SPL xxxxxxxxxxxxx
U-Boot xxxxxxxxxxx
```

A bootloader is present in the NOR flash, no action is needed and the bootconfig supports booting from NOR flash as well.

Note: If you ordered a custom SL i.MX8MM or BL i.MX8MM, the bootconfig might be different, however you might still be able to boot from eMMC. Check the page "Using the System" to see which setting you have.

If the last 2 lines are missing, then no bootloader is present and needs to be written to the NOR flash for the device to be able to boot from eMMC memory.

```
root@kontron-mx8mm:~# fwinfo
Fuses: SOM SerialNo: xxxxxxxx
Fuses: Board SerialNo: xxxxxxx
Fuses: Bootconfig: 0x18001410
Fuses: MAC1: xxxxxxxxxxxx
Fuses: MAC2: xxxxxxxxxxxx
Fuses: Lock: xxxxxxxxxxxx
Version: Firmware Version: xxxxxxxxxxxxxxxxx
Log: Devicetree info: xxxxxxxxxxxxxxx
Linux version xxxxxxxxxxxxxxx
```

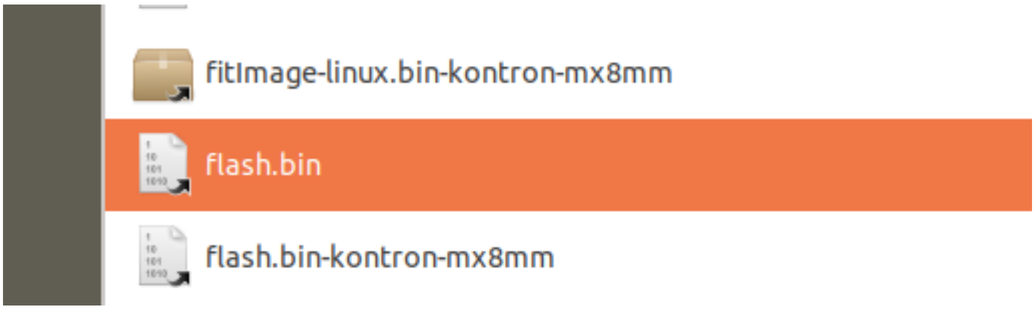
Locating needed files

For the *U-Boot* bootloader to work, 2 files are needed. These files are named *flash.bin* and *uboot.bin*. The files are located in the BSP output folder where you also find the system image (wic) file.

The *flash.bin* file is most likely a symbolic link, follow it or look at the file properties and note the name of the original file or copy it to a save location. The same goes for the *uboot.bin*, but here the file is named *u-boot-kontron-mx8mm.bin*, which also is most likely a symbolic link which you have to follow to find the original file. Note it down or copy it also to a save location.

Following are some images from a Ubuntu Linux system showing the files as an example. On your system the files can be named differently, so it is best to double check the files names before coping them.

The *flash.bin* file symbolic link

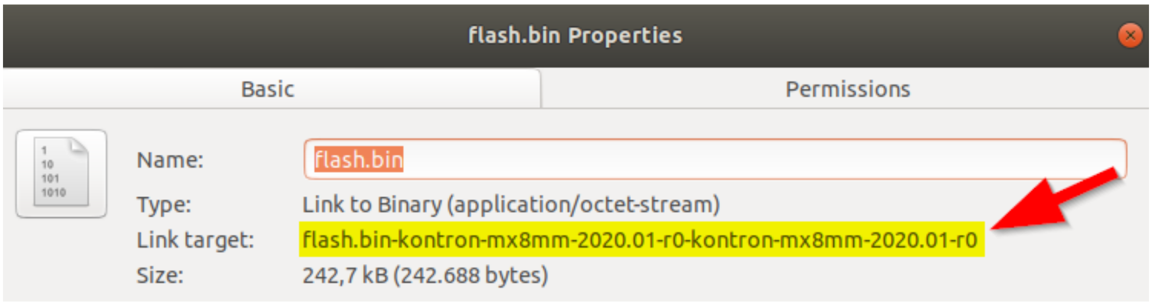


fitImage-linux.bin-kontron-mx8mm

flash.bin

flash.bin-kontron-mx8mm

File properties window of the *flash.bin* file revealing the actual file



flash.bin Properties

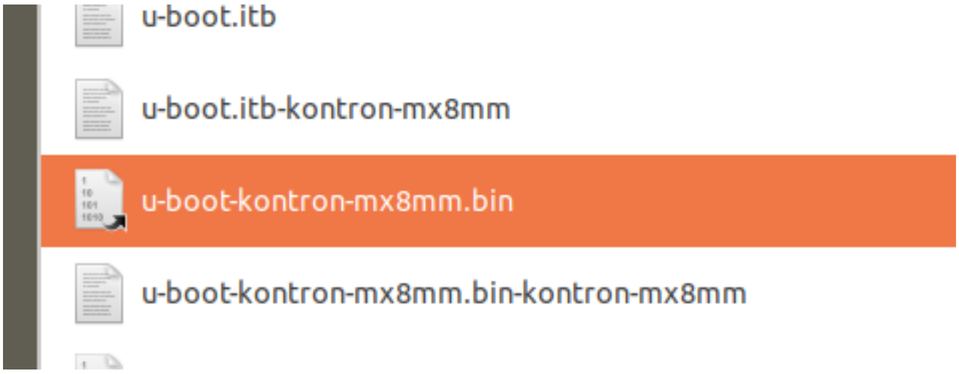
Name: flash.bin

Type: Link to Binary (application/octet-stream)

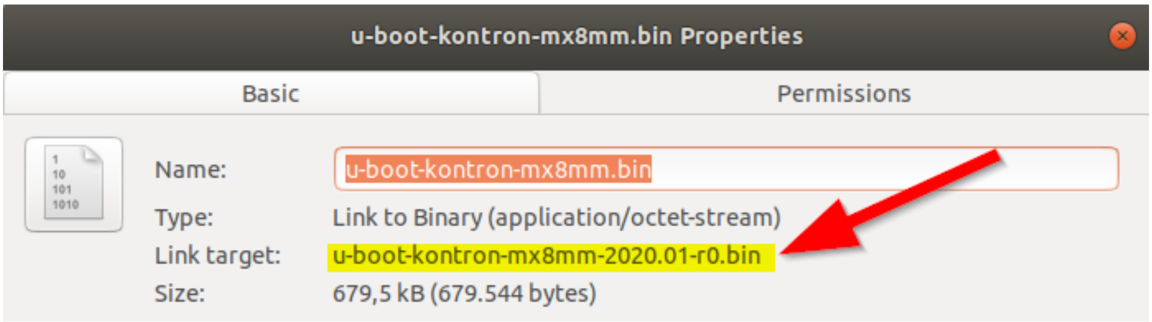
Link target: flash.bin-kontron-mx8mm-2020.01-r0-kontron-mx8mm-2020.01-r0

Size: 242,7 kB (242.688 bytes)

The *u-boot-kontron-mx8mm.bin* symbolic link



File properties window of the *u-boot-kontron-mx8mm.bin* file revealing the actual file



Name: u-boot-kontron-mx8mm.bin
Type: Link to Binary (application/octet-stream)
Link target: u-boot-kontron-mx8mm-2020.01-r0.bin
Size: 679,5 kB (679.544 bytes)

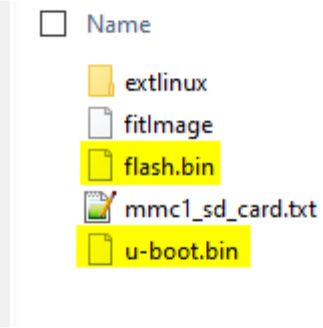
Preparations

Now that you have the files, a few more steps are need before you can write the files to the NOR flash of the BL i.MX8MM.

- Insert the SD card you have been using so far to boot the BL i.Mx8MM into a computer using a card reader. You can also create a new SD card, for this have a look at the previous topics further up in this guide first and do not continue with the preparations until you have a booting SD card for the BL i.MX8MM.
- Copy both files on to the first partition of the SD card. Don't place the files in a folder or subfolder, leave them in the root folder.
- Rename each file accordingly, in this example the file *flash.bin-kontron-mx8mm-2020.01-r0-kontron-mx8mm-2020.01-r0* is renamed to *flash.bin*.
- and the *U-Boot* file, in this case *u-boot-kontron-mx8mm-2020.01-r0.bin* is renamed to *uboot.bin*.

- You can also create an empty text file called *mmc1_sd_card.txt* for example, so you can identify the SD card later on in the *U-Boot* bootloader, because the onboard eMMC memory also identifies as an SD card.

The files on the SD card should look something like in the following image:



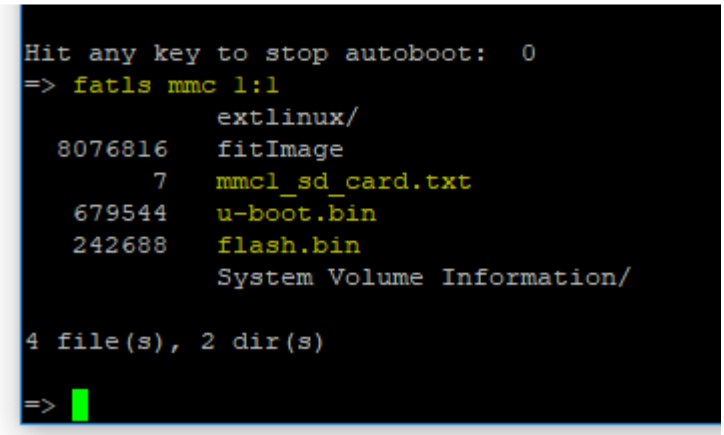
□ Name

extlinux
fitImage
flash.bin
mmc1_sd_card.txt
u-boot.bin

Writing the Bootloader

Everything has been prepared, now you can write the *U-Boot* files to the NOR flash of the BL i.MX8MM.

- Insert the SD card into the BL i.MX8MM but have the serial console ready to press the space bar when the 3 second timeout appears. You have to press the space bar within this timeframe to enter the bootloader stored on the SD card.
    - The message in the console will say: *Hit any key to stop autoboot*
- Once you are in the bootloader, issue the command *fatls mmc 1:1* to list the files of the SD card's first partition. You should see a listing like the one in the image.

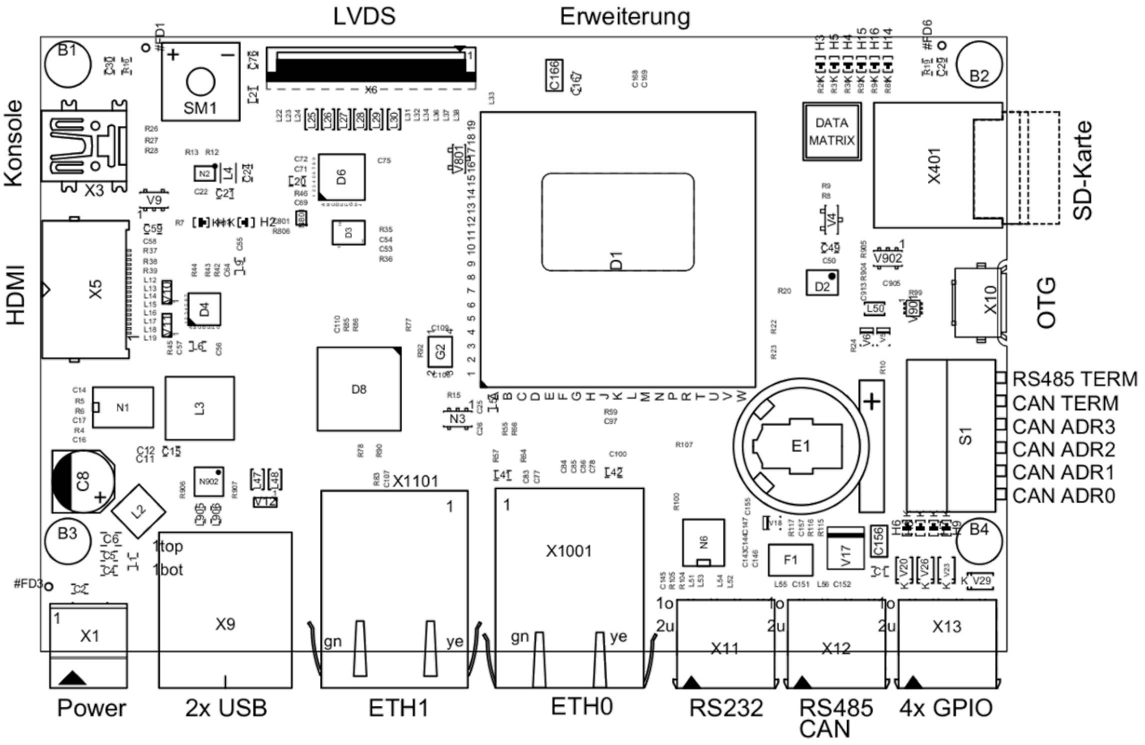


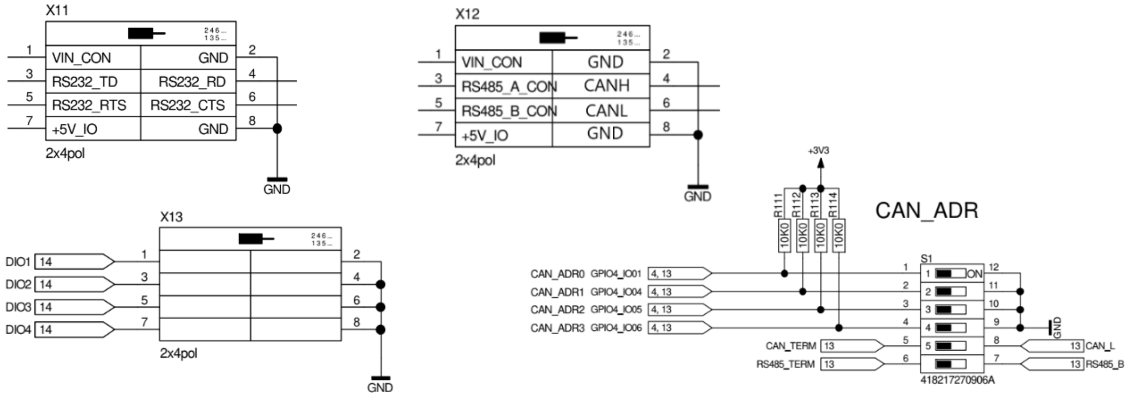
```
Hit any key to stop autoboot:  0
=> fatls mmc 1:1
            extlinux/
  8076816   fitImage
        7   mmc1_sd_card.txt
   679544   u-boot.bin
   242688   flash.bin
            System Volume Information/

4 file(s), 2 dir(s)

=> 
```

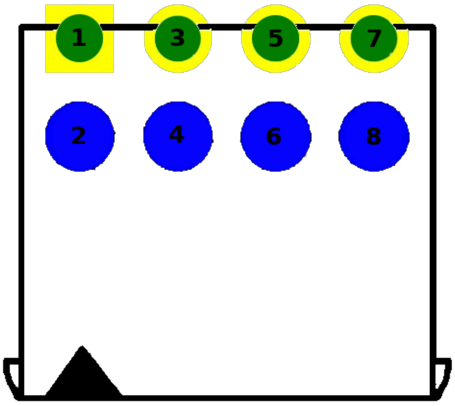
- Now follow the steps from the page Using the System in the section *SPI NOR Boot*
  - **Remark:** If it comes to writing to the NOR flash, you will see this term *${filesize}* in the commands, this is correct. Do not enter any numbers or anything in its place, use the commands as they are.
- Once done with the steps enter the command: *boot*
- The system still boots from SD card, but now enter the command *fwinfo* again and see if it says "*U-Boot xxxx*" at the bottom of the text.
- If so, power off the device, take out the SD card and power the BL i.MX8MM back on and look at the serial console if the bootloader now gets loaded from the NOR flash and then boots your system image from the eMMC. If you have no system installed in the eMMC memory, take a look at the section *Writing the System Image to eMMC Memory*.

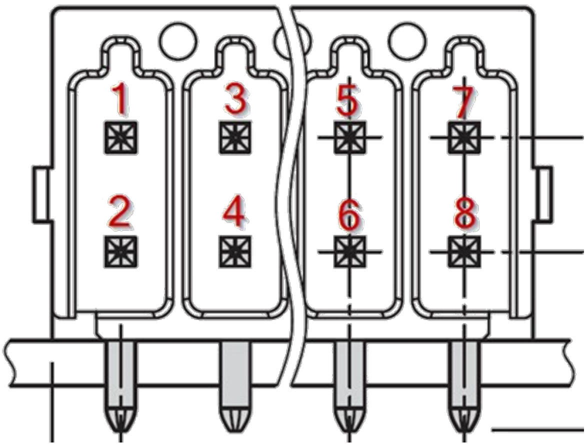
BL i.MX8MM (N801x S)

Board Layout and Connectors









IOs

Four digital inputs/outputs (either or) are available.
The table below shows number and function of available DIOs and their associated GPIOs.

The input/output voltage is coupled to the power supply voltage (normally 24 volts).

DIO Name	Direction	GPIO Name	GPIO Device	GPIO Offset	Connector
DIO1	output	`GPIO1_IO03`	gpiochip0	3	X13 - Pin 1
DIO1	input	`GPIO1_IO06`	gpiochip0	6	X13 - Pin 1
DIO2	output	`GPIO1_IO07`	gpiochip0	7	X13 - Pin 3
DIO2	input	`GPIO1_IO08`	gpiochip0	8	X13 - Pin 3
DIO3	output	`GPIO1_IO09`	gpiochip0	9	X13 - Pin 5
DIO3	input	`GPIO1_IO10`	gpiochip0	10	X13 - Pin 5
DIO4	output	`GPIO1_IO011`	gpiochip0	11	X13 - Pin 7
DIO4	input	`GPIO5_IO02`	gpiochip4	2	X13 - Pin 7

For an example on how to use or work with the DIOs see section "Using the System" for more information.

Serial Devices

Below you will find an overview which serial device has which name in the OS and belongs to which UART device of the iMX8 CPU.

UART	Type	Accessible via	Connector
UART 1	RS232	`/dev/ttymxc0`	X11
UART 2	RS485	`/dev/ttymxc1`	X12
UART 3	UART/TTL (Debug UART)	`/dev/ttymxc2`	X3 (USB Mini-B)

For an example on how to use the serial devices see section "Using the System" for more information.

CAN-Bus Interface

The BL i.MX8MM (N801x S) has one CAN-Bus interface. The device is present after boot, but is not enabled by default.

Name	Accessible via	Connector
CAN	`can0`	X12

For an example on how to use the CAN-Bus see section "Using the System" for more information.

Using the System

Boot Devices

The i.MX8MM SoM has the two boot pins on the SoC (BOOT_MODE0 and BOOT_MODE1) unconnected by default, which means the boot mode is set to "00". The BootROM will use the manufacture-mode to look for a bootable image on the SD-card. If none is found, it will fall back to serial loader mode, where it expects an image to be loaded via USB-OTG1 (which is on the micro USB connector on the demo board).

To select other boot devices such as the SPI NOR flash or the eMMC, you need to program the OTP fuses accordingly.

The default setup for production devices from Kontron is that the fuses are set to boot from the SD card (SD2) as primary boot device. The SPI NOR is set as fallback boot device if no SD card is available.

If you order SoMs from Kontron and you need a different setup, please supply the necessary information with your order.

The following table shows the default configuration from Kontron in the first row and below other possibilities for boot device configurations as example.

Fuse/Register Name	Offset	Value	Description
BOOT_CFG **BOOT_CFG_PARAMETER**	**0x470** **0x480**	**0x18001410** **0x03000010**	**Boot from SD (SD2, 4bit buswidth, 3.3V), set BT_FUSE_SEL,** **Use SPI NOR as fallback (eCSPI1, CS0)**
BOOT_CFG BOOT_CFG_PARAMETER	0x470 0x480	0x18000060 0x00000000	Boot from SPI NOR (eCSPI1, CS0), set BT_FUSE_SEL, No fallback
BOOT_CFG BOOT_CFG_PARAMETER	0x470 0x480	0x18002020 0x03000010	Boot from eMMC (SD1, 8bit buswidth, 3.3V), set BT_FUSE_SEL, Use SPI NOR as fallback (eCSPI1, CS0)
BOOT_CFG BOOT_CFG_PARAMETER	0x470 0x480	0x18002020 0x00000000	Boot from eMMC (SD1, 8bit buswidth, 3.3V), set BT_FUSE_SEL, No fallback

SPI NOR Boot

The U-Boot configuration for the Kontron i.MX8MM SoM creates two image: `flash.bin` for the SPL and `u-boot.itb` for TF-A and U-Boot proper (see also "Boot Chain Overview").

Here is an example of how to write the two images to the SPI NOR from within the bootloader (running from SD-card). The two image files are expected to be stored on a FAT partition of the SD-card.

> **Info**
>
> It is recommended to flash the Bootloader with ptool or swupdate because you don't have to take care of memory offsets. This is explained in the chapter Flash-Layout

Probing the SPI NOR

```
=> sf probe
SF: Detected mx25r1635f with page size 256 Bytes, erase size 4 KiB,
total 2 MiB
```

Erasing the whole flash

```
=> sf erase 0 0x200000
SF: 2097152 bytes @ 0x0 Erased: OK
```

Loading the SPL image

```
=> fatload mmc 1:1 0x40000000 flash.bin
242688 bytes read in 23 ms (10.1 MiB/s)
```

Writing the SPL image (offset: 1 KiB)

```
=> sf write 0x40000000 0x400 ${filesize}
device 0 offset 0x400, size 0x3b400
SF: 242688 bytes @ 0x400 Written: OK
```

Loading the U-Boot image

```
=> fatload mmc 1:1 0x40000000 u-boot.bin
679544 bytes read in 43 ms (15.1 MiB/s)
```

Writing the U-Boot image (offset: 320 KiB)

```
=> sf write 0x40000000 0x50000 ${filesize}
device 0 offset 0x50000, size 0xa5e78
SF: 679544 bytes @ 0x50000 Written: OK
```

Qt Applications and Backends

To make use of the GPU for your QtQuick applications, we recommend to use the `eglfs_kms` backend.

Using the DIOs

The 4 DIOs of the BL i.MX8MM (N801x S) are shown below. They can be accessed in the operating system using commands from the `libgpiod` library. The GPIOs in the operating system are grouped by so called `gpiochip` devices.

Note: The counting in the OS starts at *0* whereas the GPIO names/numbers (column "Accessible via") start at *1*. This does not apply to latter, the "IOxx" part. The number behind "IO" denotes the GPIO offset number within each gpiochip device when using *libgpiod*. See below for examples.

DIO - GPIO overview:

DIO Name	Direction	GPIO Name	GPIO Device	GPIO Offset	Connector
DIO1	output	`GPIO1_IO03`	gpiochip0	3	X13 - Pin 1
DIO1	input	`GPIO1_IO06`	gpiochip0	6	X13 - Pin 1
DIO2	output	`GPIO1_IO07`	gpiochip0	7	X13 - Pin 3
DIO2	input	`GPIO1_IO08`	gpiochip0	8	X13 - Pin 3
DIO3	output	`GPIO1_IO09`	gpiochip0	9	X13 - Pin 5
DIO3	input	`GPIO1_IO10`	gpiochip0	10	X13 - Pin 5
DIO4	output	`GPIO1_IO011`	gpiochip0	11	X13 - Pin 7
DIO4	input	`GPIO5_IO02`	gpiochip4	2	X13 - Pin 7

Examples

DIO as Outputs

With the `gpioset` command from the libgpiod library package, a GPIO can be configured as an output and its state can be set at the same time. The command needs the gpiochip number or name and then the GPIO offset number that should be changed.

Configure DIO1 (GPIO1_IO03) as an output and setting it to 1 (High, On).

```
root@kontron-mx8mm:/# gpioset gpiochip0 3=1
```

Setting DIO1 (GPIO1_IO03) to 0 (Low, Off).

```
root@kontron-mx8mm:/# gpioset gpiochip0 3=0
```

Configure DIO4 (GPIO1_IO011) as an output and setting it to 1 (High, On).

```
root@kontron-mx8mm:/# gpioset gpiochip0 11=1
```

Setting DIO4 (GPIO1_IO011) to 0 (Low, Off).

```
root@kontron-mx8mm:/# gpioset gpiochip0 11=0
```

DIO as Inputs

To read the current value of an DIO and configuring it as an input at the same time, the `gpioget` command from the libgpiod library package can be used.

Configure DIO1 (GPIO1_IO06) as an input and read the current value.

```
root@kontron-mx8mm:/# gpioget gpiochip0 6
```

Configure DIO2 (GPIO1_IO08) as an input and read the current value.

```
root@kontron-mx8mm:/# gpioget gpiochip0 8
```

Configure DIO3 (GPIO1_IO10) as an input and read the current value.

```
root@kontron-mx8mm:/# gpioget gpiochip0 10
```

Configure DIO4 (GPIO5_IO02) as an input and read the current value.

```
root@kontron-mx8mm:/# gpioget gpiochip4 2
```

If you want to see the change of an input, the command *watch* can help. In the following example we are watching DIO4 using the `gpioget` command to read the current value of the input. With "-n 1" we can set the update interval to 1 second.

```
root@kontron-mx8mm:/# watch -n 1 gpioget 4 2
```

To see all available gpiochip devices and known GPIO offsets (lines) the command `gpioinfo` prints out all available devices.

```
root@kontron-mx8mm:/# gpioinfo
gpiochip0 - 32 lines:
        line   0:      unnamed       unused   input  active-high
        line   1:      unnamed       unused   input  active-high
....

gpiochip1 - 32 lines:
        line   0:      unnamed       unused   input  active-high
        line   1:      unnamed       unused   input  active-high
        line   2:      unnamed       unused   input  active-high
....

gpiochip2 - 32 lines:
        line   0:      unnamed       unused   input  active-high
        line   1:      unnamed       unused   input  active-high
....

gpiochip3 - 32 lines:
        line   0:      unnamed       unused   input  active-high
        line   1:      unnamed       unused   input  active-high
....


gpiochip4 - 32 lines:
        line   0:      unnamed       unused   input  active-high
        line   1:      unnamed       unused   input  active-high
...
```

If you just want to see the GPIO info of a particular gpiochip device, adding the name of the device behind the `getinfo` command prints out only the information for the given device.

For example to display only the information for `gpiochip0` device:

```
root@kontron-mx8mm:/# gpioinfo gpiochip0
gpiochip0 - 32 lines:
        line   0:      unnamed       unused   input  active-high
        line   1:      unnamed       unused   input  active-high
        line   2:      unnamed       unused   input  active-high
        line   3:      unnamed       unused   input  active-high
...
```

Using Serial Devices

The serial devices on the i.MX8 usually have a name like `ttymxc` and the number at the end corresponds to a UART device. Below is a table with the different UARTs of the i.MX8, what type they are and how each serial device can be accessed, meaning which tty device corresponds to which UART.

UART	Type	Accessible via	Connector
UART 1	RS232	`/dev/ttymxc0`	X11
UART 2	RS485	`/dev/ttymxc1`	X12
UART 3	UART/TTL (Debug UART)	`/dev/ttymxc2`	X3 (USB Mini-B)

To be able to use UART1 and and UART2, they have to be configured before their first use. They have to be switched from *terminal emulator* (canonical mode) to "*raw*". Only this way the user can access the devices and send and receive data.

Note: UART3 does not need to be configured, this is done automatically at the start through the bootloader. Do not try to reconfigure this interface, otherwise it might not be possible to establish any debug connection with the board / OS.

Configure UART1 (RS232)

After logging in via the Debug Console the following command needs to be executed in order to get the RS232 (ttymxc0) ready for use.

```
root@kontron-mx8mm:/# stty -F /dev/ttymxc0 raw -echo -echoe -echok 9600
```

This command switches UART1 but also sets a baud rate of 9600 at the same time.

Configure UART2 (RS485)

After logging in via the Debug Console the following command needs to be executed in order to get the RS485 (ttymxc1) ready for use. Remember when using this interface, that communication can only be half duplex. Only one device is allowed to communicate at any given time, otherwise data will be lost.

```
root@kontron-mx8mm:/# stty -F /dev/ttymxc1 raw -echo -echoe -echok 9600
```

This command switches UART1 but also sets a baud rate of 9600 at the same time.

Examples

Following are some examples on how to use the serial ports on the BL i.MX8MM (N801x S) and how to configure them.

RS232 communication

This example assumes that the RS232 interface of the BL i.MX8MM (N801x S) is wired to another devices RS232 interface, remember to swap the Rx and Tx lines and also connect up GND, otherwise the received data can look very strange.

After connecting the RS232 to the other device, it can be configured and is ready to send and received data.

```
root@kontron-mx8mm:/# stty -F /dev/ttymxc0 raw -echo -echoe -echok 19200
root@kontron-mx8mm:/# cat /dev/ttymxc0
```

```
Hello World!
Hello World!
Hello World!
Hello World!
```

Sending out data from the console, the `echo` command can be used and the output is forwarded to the serial device.

```
root@kontron-mx8mm:/# echo "Hello World!" > /dev/ttymxc0
```

The PC now also displays the the received text "Hello World!".

RS485 communication

This example assumes that the RS485 interface of the BL i.MX8MM (N801x S) is wired to a PC USB-to-RS485 adapter and that the lines A, B and GND are connected appropriately. Of course any other device with an RS485 interface is just as good.

After wiring the RS485, it can be configured and is ready to receive data.

```
root@kontron-mx8mm:/# stty -F /dev/ttymxc1 raw -echo -echoe -echok 9600
root@kontron-mx8mm:/# cat /dev/ttymxc1
Hello World!
Hello World!
Hello World!
Hello World!
```

Sending out data from the console, the `echo` command can be used and the output is forwarded to the serial device.

```
root@kontron-mx8mm:/# echo "Hello World!" > /dev/ttymxc1
```

The PC should now display the received text "Hello World!".

Using CAN-Bus

The CAN-Bus interface is present in the OS after boot, but it is not activated by default. Following is a table with the name of the CAN interface and for an example on how to configure and use it, see below.

Name	Accessible via	Connector
CAN	`can0`	X12

Checking CAN availability

To be sure that the CAN device or interface is really present in the system, the command `ifconfig` with the `-a` parameter can be used to list all socket based communication devices. The `can0` interface should be right at the top.

```
root@kontron-mx8mm:~# ifconfig -a
can0      Link encap:UNSPEC  HWaddr
00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          NOARP  MTU:16  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:10
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

eth0      Link encap:Ethernet  HWaddr 82:73:E0:1D:85:45
....

eth1      Link encap:Ethernet  HWaddr 42:04:8D:DC:2E:DF
....

lo        Link encap:Local Loopback
....
```

Configure and activate the CAN interface

In order for the CAN interface to work, it has to be configured first and then activated to become available for use.

Use the following commands to ready the interface:

```
root@kontron-mx8mm:~# ifconfig can0 down
root@kontron-mx8mm:~# ip link set can0 type can bitrate 125000
root@kontron-mx8mm:~# ifconfig can0 up
```

The first command disables the interface, the second command configures the `can0` interface as can with a bitrate of 125 kbits. The third command enables or re-enables the CAN interface with the new settings and the device is now ready for use.

Examples

To make sure the CAN interface is really available, entering the *ifconfig* command on its own will show all currently active network or socket based devices. In the following examples the programs `candump` and `cansend` are used to send and receive CAN messages.

Receiving data

To receive data from the CAN-Bus and print it to the console, we can use the `candump` command.

```
root@kontron-mx8mm:/# candump can0
interface = can0, family = 29, type = 3, proto = 1
<0x134> [4] de ad be ef
<0x134> [4] de ad be ef
<0x134> [4] de ad be ef
<0x134> [4] de ad be ef
```

The CAN-Bus master sent out 4 messages each containing 4 bytes of data. The id or CAN identifier of each message is 0x134 (hexadecimal) in this example.

Sending data

To send data via the CAN-Bus, we can use the `cansend` command.

```
root@kontron-mx8mm:/# cansend can0 -v -i 0x123 0xDE 0xAD 0xBE 0xEF
interface = can0, family = 29, type = 3, proto = 1
id: 291 dlc: 4
0xde 0xad 0xbe 0xef
```

The parameters `-v` means verbose output and with `-i` we can set our own message id. The default message id *cansend* uses is "1" if the "-i" parameter is not specified. The data or byte(s) which we want to send, have to be supplied individually in the format 0x*NM*, where *NM* has to be a hexadecimal number in the range of 0x00 to 0xFF which equals the decimal range of 0 to 255. In this example we send 4 bytes containing the data "*deadbeef*".

Using the Cortex M4 Core

Using the Cortex M4 Core on Kontron i.MX8MM Boards

The following guide uses U-Boot or Linux to load a binary compiled with the NXP MCUXpresso SDK into the M4 core of the i.MX8MM and start it. The application on the M4 is able to communicate with the Linux system through a custom `rpmsg` client driver kernel module.

1. Compiling an Application for the M4 Core

Please follow the guide in this blog post from i.MX guru Detlev Zundel to setup the SDK and compile a demo application.

The demo applications for the NXP i.MX8MM Evaluation Kit (EVK) can also be used on the Kontron boards. You might want to adjust the the peripherals and pinmux settings to match your hardware.

In the following we will use two different example apps:

1. **Example 1**: The *"hello_world"* example for creating a simple UART console. We will load this app **via U-Boot**.
2. **Example 2**: The *"rpmsg_lite_pingpong_rtos"* example to demonstrate the intercore communication. We will load this app **via Linux**.

In order to receive log messages from the M4 core, we need access to an additional UART (not the one used by Linux). In our example we will use the RS232 interface (UART1) on the Kontron board as the M4 debug console.

To switch from UART4 used on the NXP i.MX8MM EVK to UART1, the following changes in the source code are required.

✎ **Demo Source Code Changes for using UART1 as M4 console**

```
--- a/evkmimx8mm/demo_apps/hello_world/pin_mux.c
+++ b/evkmimx8mm/demo_apps/hello_world/pin_mux.c
@@ -55,14 +55,14 @@ BOARD_InitPins:
  *
  * END
*********************************************************************************
 void BOARD_InitPins(void) {                                /*!< Function
assigned for the core: Cortex-M4[m4] */
-    IOMUXC_SetPinMux(IOMUXC_UART4_RXD_UART4_RX, 0U);
-    IOMUXC_SetPinConfig(IOMUXC_UART4_RXD_UART4_RX,
+    IOMUXC_SetPinMux(IOMUXC_SAI2_RXC_UART1_RX, 0U);
+    IOMUXC_SetPinConfig(IOMUXC_SAI2_RXC_UART1_RX,
                         IOMUXC_SW_PAD_CTL_PAD_DSE(6U) |
                         IOMUXC_SW_PAD_CTL_PAD_FSEL(2U));
-    IOMUXC_SetPinMux(IOMUXC_UART4_TXD_UART4_TX, 0U);
-    IOMUXC_SetPinConfig(IOMUXC_UART4_TXD_UART4_TX,
+    IOMUXC_SetPinMux(IOMUXC_SAI2_RXFS_UART1_TX, 0U);
+    IOMUXC_SetPinConfig(IOMUXC_SAI2_RXFS_UART1_TX,
                         IOMUXC_SW_PAD_CTL_PAD_DSE(6U) |
                         IOMUXC_SW_PAD_CTL_PAD_FSEL(2U));

--- a/evkmimx8mm/multicore_examples/rpmsg_lite_pingpong_rtos/linux_remote/
pin_mux.c
+++ b/evkmimx8mm/multicore_examples/rpmsg_lite_pingpong_rtos/linux_remote/
pin_mux.c
@@ -55,14 +55,14 @@ BOARD_InitPins:
  *
  * END
*********************************************************************************
 void BOARD_InitPins(void) {                                /*!< Function
assigned for the core: Cortex-M4[m4] */
-    IOMUXC_SetPinMux(IOMUXC_UART4_RXD_UART4_RX, 0U);
-    IOMUXC_SetPinConfig(IOMUXC_UART4_RXD_UART4_RX,
+    IOMUXC_SetPinMux(IOMUXC_SAI2_RXC_UART1_RX, 0U);
+    IOMUXC_SetPinConfig(IOMUXC_SAI2_RXC_UART1_RX,
                         IOMUXC_SW_PAD_CTL_PAD_DSE(6U) |
                         IOMUXC_SW_PAD_CTL_PAD_FSEL(2U));
-    IOMUXC_SetPinMux(IOMUXC_UART4_TXD_UART4_TX, 0U);
-    IOMUXC_SetPinConfig(IOMUXC_UART4_TXD_UART4_TX,
+    IOMUXC_SetPinMux(IOMUXC_SAI2_RXFS_UART1_TX, 0U);
+    IOMUXC_SetPinConfig(IOMUXC_SAI2_RXFS_UART1_TX,
                         IOMUXC_SW_PAD_CTL_PAD_DSE(6U) |
                         IOMUXC_SW_PAD_CTL_PAD_FSEL(2U));
 }

--- a/boards/evkmimx8mm/board.c
+++ b/boards/evkmimx8mm/board.c
@@ -24,7 +24,7 @@
 void BOARD_InitDebugConsole(void)
 {
     uint32_t uartClkSrcFreq = BOARD_DEBUG_UART_CLK_FREQ;
-    CLOCK_EnableClock(kCLOCK_Uart4);
+    CLOCK_EnableClock(kCLOCK_Uart1);
     DbgConsole_Init(BOARD_DEBUG_UART_INSTANCE, BOARD_DEBUG_UART_BAUDRATE,
BOARD_DEBUG_UART_TYPE, uartClkSrcFreq);
 }
 /* Initialize MPU, configure non-cacheable memory */

--- a/boards/evkmimx8mm/board.h
+++ b/boards/evkmimx8mm/board.h
```

```
@@ -19,13 +19,13 @@
 /* The UART to use for debug messages. */
 #define BOARD_DEBUG_UART_TYPE     kSerialPort_Uart
 #define BOARD_DEBUG_UART_BAUDRATE 115200u
-#define BOARD_DEBUG_UART_BASEADDR UART4_BASE
-#define BOARD_DEBUG_UART_INSTANCE 4U
+#define BOARD_DEBUG_UART_BASEADDR UART1_BASE
+#define BOARD_DEBUG_UART_INSTANCE 1U
 #define
BOARD_DEBUG_UART_CLK_FREQ
\
-    CLOCK_GetPllFreq(kCLOCK_SystemPll1Ctrl) /
(CLOCK_GetRootPreDivider(kCLOCK_RootUart4)) / \
-        (CLOCK_GetRootPostDivider(kCLOCK_RootUart4)) / 10
-#define BOARD_UART_IRQ         UART4_IRQn
-#define BOARD_UART_IRQ_HANDLER UART4_IRQHandler
+    CLOCK_GetPllFreq(kCLOCK_SystemPll1Ctrl) /
(CLOCK_GetRootPreDivider(kCLOCK_RootUart1)) / \
+        (CLOCK_GetRootPostDivider(kCLOCK_RootUart1)) / 10
+#define BOARD_UART_IRQ         UART1_IRQn
+#define BOARD_UART_IRQ_HANDLER UART1_IRQHandler

 #define GPV5_BASE_ADDR        (0x32500000)
 #define FORCE_INCR_OFFSET     (0x4044)

--- a/boards/evkmimx8mm/clock_config.c
+++ b/boards/evkmimx8mm/clock_config.c
@@ -99,8 +99,8 @@ void BOARD_BootClockRUN(void)
     //    CLOCK_SetRootDivider(kCLOCK_RootAxi, 1U, 2);
     //    CLOCK_SetRootMux(kCLOCK_RootAxi, kCLOCK_AxiRootmuxSysPll1); /*
switch AXI to SYSTEM PLL1 800MHZ */

-    CLOCK_SetRootMux(kCLOCK_RootUart4, kCLOCK_UartRootmuxSysPll1Div10); /*
Set UART source to SysPLL1 Div10 80MHZ */
-    CLOCK_SetRootDivider(kCLOCK_RootUart4, 1U, 1U);                     /*
Set root clock to 80MHZ/ 1= 80MHZ */
+    CLOCK_SetRootMux(kCLOCK_RootUart1, kCLOCK_UartRootmuxSysPll1Div10); /*
Set UART source to SysPLL1 Div10 80MHZ */
+    CLOCK_SetRootDivider(kCLOCK_RootUart1, 1U, 1U);                     /*
Set root clock to 80MHZ/ 1= 80MHZ */

     CLOCK_EnableClock(kCLOCK_Rdc); /* Enable RDC clock */
     /* The purpose to enable the following modules clock is to make sure the
M4 core could work normally when A53 core
```

2. Modify the TF-A code to assign peripherals to the correct RDC domain

The Ressource Domain Controller (RDC) is used to assign peripherals to either the A53 domain or the M4 domain. Using a peripheral from a domain that it is not assigned to usually leads to failures like system lockups.

The following code changes in `imx-atf` assign the UART1 used as debug console for the M4 core to the M4 domain. Other peripherals need to be added as required for the application.

TF-A code changes to assign UART1 to M4 domain

```
--- a/plat/imx/imx8m/imx8mm/imx8mm_bl31_setup.c
+++ b/plat/imx/imx8m/imx8mm/imx8mm_bl31_setup.c
@@ -58,7 +58,7 @@ static const struct imx_rdc_cfg rdc[] = {
        RDC_MDAn(RDC_MDA_M4, DID1),

        /* peripherals domain permission */
-       RDC_PDAPn(RDC_PDAP_UART4, D1R | D1W),
+       RDC_PDAPn(RDC_PDAP_UART1, D1R | D1W),
        RDC_PDAPn(RDC_PDAP_UART2, D0R | D0W),

        /* memory region */
```

3. Modify the Linux Devicetree

3.1 Adding the Devicetree Nodes

Step only needed for Example 2

This step is only needed for "Example 2". It can be skipped if only "Example 1" is used (loaded from U-Boot).

DDR addresses

The example from NXP is configured for the NXP EVK with 2GB of DDR RAM. If your hardware has less RAM available, you might have to adjust the memory mapping of the M4 app and change the devicetree accordingly. For 1GB of DDR using 0x77000000 as base for the M4 app and 0x78000000 for the shared resources should work. In the M4 app `MEMORY` in `MIMX8MM6xxxxx_cm4_ddr_ram.ld` and `VDEV0_VRING_BASE` in `board.h` needs to be changed.

Add the following nodes to your board devicetree's root node in order to set up the memory for the M4 core and the `remoteproc` driver.

Devicetree Nodes for the M4 Core

```
/ {
    [...]

    reserved-memory {
        #address-cells = <2>;
        #size-cells = <2>;
        ranges;

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

        vdev0vring0: vdev0vring0@b8000000 {
            reg = <0 0xb8000000 0 0x8000>;
            no-map;
        };

        vdev0vring1: vdev0vring1@b8008000 {
            reg = <0 0xb8008000 0 0x8000>;
            no-map;
        };

        rsc_table: rsc_table@b80ff000 {
            reg = <0 0xb80ff000 0 0x1000>;
            no-map;
        };

        vdevbuffer: vdevbuffer@b8400000 {
            compatible = "shared-dma-pool";
            reg = <0 0xb8400000 0 0x100000>;
            no-map;
        };
    };

    imx8mm-cm4 {
        compatible = "fsl,imx8mm-cm4";
        clocks = <&clk IMX8MM_CLK_M4_DIV>;
        mbox-names = "tx", "rx", "rxdb";
        mboxes = <&mu 0 1
              &mu 1 1
              &mu 3 1>;
        memory-region = <&vdevbuffer>, <&vdev0vring0>, <&vdev0vring1>,
<&rsc_table>;
        syscon = <&src>;
    };

    [...]
};
```

3.2 Disable M4 Peripherals in Linux

At this point we also need to make sure, that no peripherals are probed by Linux that are assigned to M4 domain. In our case we need to disable UART1:

```
--- a/arch/arm64/boot/dts/freescale/imx8mm-kontron-n801x-s.dts
+++ b/arch/arm64/boot/dts/freescale/imx8mm-kontron-n801x-s.dts
@@ -307,7 +307,7 @@ &uart1 {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_uart1>;
        uart-has-rtscts;
-       status = "okay";
+       status = "disabled";
 };
```

4. Modify the Kernel Configuration

Step only needed for Example 2

This step is only needed for "Example 2". It can be skipped if only "Example 1" is used (loaded from U-Boot).

In order to build the drivers used in this example, we enable the following in our `defconfig`.

```
+CONFIG_REMOTEPROC=y
+CONFIG_REMOTEPROC_CDEV=y
+CONFIG_IMX_REMOTEPROC=y
+CONFIG_RPMSG_CHAR=m
+CONFIG_RPMSG_CTRL=m
+CONFIG_RPMSG_VIRTIO=m
+CONFIG_IMX_RPMSG_PINGPONG=m
```

5. Example 1: Loading and Starting through U-Boot

The compiled application's BIN file is copied to the DDR via TFTP (or alternatively from some storage device).

```
=> tftp 0x42000000 hello_world.bin
Using ethernet@30be0000 device
TFTP from server 192.168.1.10; our IP address is 192.168.1.11
Filename 'hello_world.bin'.
Load address: 0x42000000
Loading: #
         2.7 MiB/s
```

```
done
Bytes transferred = 14260 (37b4 hex)
```

Next, we copy the executable from DDR to the internal TCML memory, where we want to start it from.

```
=> cp.b 0x42000000 0x7e0000 ${filesize}
```

At last we will use the `bootaux` command to start the app.

```
=> bootaux 0x7e0000
## No elf image at address 0x007e0000
## Starting auxiliary core stack = 0x20020000, pc = 0x1FFE02CD...
```

At this point you should closely watch the UART console attached to the M4. It will print "hello world" ans start echoing all characters it receives.

⚠ **Linux Kernel Clock Gating**

To make sure that the app running on the M4 core will continue to be executed and not freeze when Linux is booted, the kernel needs to be told to not gate the system clocks. This can be done by adding the parameter `clk-imx8mm.mcore_booted=1` to the kernel commandline. Usually this can be done by appending the value to the `bootargs_base` variable in the U-Boot environment.

6. Example 2: Loading and Starting through Linux

ℹ **Required Kernel Version**

While "Example 1" works fine with the v5.10-ktn kernel branch, this example requires additional patches for the `remoteproc` and `rpmsg` frameworks from later kernel versions. To make it work you either need a recent mainline kernel (tested on 5.19-rc5) or you need to integrate the backport for v5.10 provided here.

After booting the system with the modifications applied as described before, we need to copy the compiled M4 application's ELF file to the root filesystem. In our example we copy the `rpmsg_lite_pingpong_rtos_linux_remote.elf` to `/lib/firmware` .

Next we can load the executable into the M4 core using the `remoteproc` sysfs interface:

```
root@kontron-mx8mm:~# echo -n
rpmsg_lite_pingpong_rtos_linux_remote.elf > /sys/class/remoteproc/
remoteproc0/firmware
```

Once loaded we can start the execution of the M4 application:

```
root@kontron-mx8mm:~# echo start > /sys/class/remoteproc/
remoteproc0/state
[ 7348.685563] remoteproc remoteproc0: powering up imx-rproc
[ 7348.692688] remoteproc remoteproc0: Booting fw image
rpmsg_lite_pingpong_rtos_linux_remote.elf, size 409576
[ 7348.702909]  remoteproc0#vdev0buffer: assigned reserved memory
node vdevbuffer@b8400000
[ 7348.712718] virtio_rpmsg_bus virtio0: rpmsg host is online
[ 7348.718388]  remoteproc0#vdev0buffer: registered virtio0 (type
7)
[ 7348.724550] remoteproc remoteproc0: remote processor imx-rproc
is now up
[ 7349.714126] virtio_rpmsg_bus virtio0: creating channel rpmsg-
openamp-demo-channel addr 0x1e
```

From the kernel log we can see that the M4 core was brought up and the `rpmsg` application on the M4 already bound itself to the kernel and announced the available communication channels.

At this point you should also see some messages printed to the M4 console:

```
RPMSG Ping-Pong FreeRTOS RTOS API Demo...
RPMSG Share Base Addr is 0xb8000000
Link is up!
Nameservice announce sent.
```

7. Loading the `rpmsg` Client Driver Kernel Module

Now the last part of the demo is to load the `rpmsg` client driver that uses the message bus to communicate with the already running M4 application.

NXP provides a demo kernel driver `imx_rpmsg_pingpong` that communicates with the `rpmsg_lite_pingpong_rtos` app on the M4.

```
root@kontron-mx8mm:~# modprobe imx_rpmsg_pingpong
```

As soon as we load the driver module we will see the output of the messages being sent and received in the kernel log and on the M4 console.

Pingpong Demo Messages

```
root@kontron-mx8mm:~# modprobe imx_rpmsg_pingpong
[ 8073.867508] 90:init
[ 8073.869731] 42:rpmsg_pingpong_probe
[ 8073.873232] imx_rpmsg_pingpong virtio0.rpmsg-openamp-demo-channel.-1.30:
new channel: 0x400 -> 0x1e!
[ 8073.885947] get 1 (src: 0x1e)
[ 8073.892351] get 3 (src: 0x1e)
[ 8073.897069] get 5 (src: 0x1e)
[ 8073.901646] get 7 (src: 0x1e)
[ 8073.906147] get 9 (src: 0x1e)
[ 8073.910645] get 11 (src: 0x1e)
[ 8073.915247] get 13 (src: 0x1e)
[ 8073.919844] get 15 (src: 0x1e)
[ 8073.924436] get 17 (src: 0x1e)
[ 8073.929131] get 19 (src: 0x1e)
[ 8073.933734] get 21 (src: 0x1e)
[ 8073.938321] get 23 (src: 0x1e)
[ 8073.942935] get 25 (src: 0x1e)
[ 8073.947551] get 27 (src: 0x1e)
[ 8073.952152] get 29 (src: 0x1e)
[ 8073.956758] get 31 (src: 0x1e)
[ 8073.961358] get 33 (src: 0x1e)
[ 8073.965946] get 35 (src: 0x1e)
[ 8073.970533] get 37 (src: 0x1e)
[ 8073.975128] get 39 (src: 0x1e)
[ 8073.979737] get 41 (src: 0x1e)
[ 8073.984334] get 43 (src: 0x1e)
[ 8073.988941] get 45 (src: 0x1e)
[ 8073.993539] get 47 (src: 0x1e)
[ 8073.998127] get 49 (src: 0x1e)
[ 8074.002713] get 51 (src: 0x1e)
[ 8074.007308] get 53 (src: 0x1e)
[ 8074.011917] get 55 (src: 0x1e)
[ 8074.016512] get 57 (src: 0x1e)
[ 8074.021120] get 59 (src: 0x1e)
[ 8074.025718] get 61 (src: 0x1e)
[ 8074.030304] get 63 (src: 0x1e)
[ 8074.034902] get 65 (src: 0x1e)
[ 8074.039505] get 67 (src: 0x1e)
[ 8074.044105] get 69 (src: 0x1e)
[ 8074.048713] get 71 (src: 0x1e)
[ 8074.053332] get 73 (src: 0x1e)
[ 8074.057918] get 75 (src: 0x1e)
[ 8074.062506] get 77 (src: 0x1e)
[ 8074.067102] get 79 (src: 0x1e)
[ 8074.071710] get 81 (src: 0x1e)
[ 8074.076308] get 83 (src: 0x1e)
[ 8074.080916] get 85 (src: 0x1e)
[ 8074.085514] get 87 (src: 0x1e)
[ 8074.090103] get 89 (src: 0x1e)
[ 8074.094690] get 91 (src: 0x1e)
[ 8074.099285] get 93 (src: 0x1e)
[ 8074.103889] get 95 (src: 0x1e)
[ 8074.108488] get 97 (src: 0x1e)
[ 8074.113095] get 99 (src: 0x1e)
[ 8074.117698] get 101 (src: 0x1e)
[ 8074.120850] imx_rpmsg_pingpong virtio0.rpmsg-openamp-demo-channel.-1.30:
goodbye!
[ 8074.229325] imx-rproc imx8mm-cm4: imx_rproc_kick: failed (0, err:-62)
```

```
RPMSG Ping-Pong FreeRTOS RTOS API Demo...
RPMSG Share Base Addr is 0xb8000000
Link is up!
Nameservice announce sent.
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
```

```
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
Sending pong...
Waiting for ping...
```

```
Sending pong...
Ping pong done, deinitializing...
Looping forever...
```

References

- *AN5317: "Loading Code on Cortex-M from U-Boot/Linux for the i.MX Asymmetric Multi-Processing Application Processors"*
- *ST Wiki: "Linux RPMsg framework overview"*
- *Detlev Zundel: "Using the M4 MCU on the i.MX8M Mini"*
- *Detlev Zundel: "Basic AMP on the i.MX8M Mini with Rpmsg"*
- *NXP MCUXpresso SDK*
- *Linux Kernel Docs: "Remote Processor Framework"*
- *Linux Kernel Docs: "Remote Processor Messaging (rpmsg) Framework"*