Copyright 2020 © Embedded Artists AB

Working with Linux and U-boot

Embedded Artists

Embedded Artists AB
Jörgen Ankersgatan 12
SE-211 45 Malmö
Sweden

http://www.EmbeddedArtists.com

Copyright 2020 © Embedded Artists AB. All rights reserved.

No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of Embedded Artists AB.

Disclaimer

Embedded Artists AB makes no representation or warranties with respect to the contents hereof and specifically disclaim any implied warranties or merchantability or fitness for any particular purpose. Information in this publication is subject to change without notice and does not represent a commitment on the part of Embedded Artists AB.

Feedback

We appreciate any feedback you may have for improvements on this document. Send your comments by using the contact form: www.embeddedartists.com/contact.

Trademarks

All brand and product names mentioned herein are trademarks, services marks, registered trademarks, or registered service marks of their respective owners and should be treated as such.

Table of Contents

1 Document Revision History 5

2 Introduction 6

2.1 Conventions 6

3 Boot process 7

3.1 Overview 7

3.2 Processor boot firmware 7

3.2.1 Boot mode register 7

3.2.2 GPIO boot 7

3.2.3 Fuses 8

3.3 SPL 8

3.4 U-boot 8

3.5 Linux 9

3.5.1 Initialization manager – systemd 9

3.6 Frequently asked questions 10

3.6.1 Can I boot from SD/MMC card instead of eMMC? 10

3.6.2 How do I launch an application at startup? 10

4 Device Tree 11

4.1 Introduction 11

4.2 Data structure format 11

4.2.1 Property values 13

4.2.2 Aliases 13

4.3 Source files and compiler 14

4.3.1 Linux 14

4.3.2 U-boot 14

4.4 Pin muxing 15

4.5 Modify device tree from U-boot 16

4.5.1 Usage 16

4.5.2 Test different commands 16

4.5.3 Add to `cmd_custom` 18

4.6 Frequently asked questions 18

4.6.1 Do I need to create my own device tree file? 18

4.6.2 How do I create my own device tree file? 18

4.6.3 How do I find the device driver? 19

4.6.4 How do I find which properties I can use? 20

4.6.5 Can I modify the device tree without re-compiling? 20

4.6.6 Can I delete a node / property? 20

4.6.7 Where do I find more documentation about device tree usage? . 20

5 Customization 21

5.1 U-boot 21

5.1.1 Board specific files 21

Copyright 2020 © Embedded Artists AB

5.1.2 Device tree files 21
5.1.3 Configuration files 22
5.2 Linux 23
5.2.1 Kernel configuration 23
5.2.2 Device drivers 24
6 Miscellaneous 25
6.1 Copy files to / from target using SCP 25
6.1.1 Allow root to login over SSH 25
6.1.2 Get IP address of the target 25
6.1.3 Copy file to target 25
6.1.4 Copy file from target 26
6.1.5 Use SCP on a Windows host 26

1 Document Revision History

Revision	***Date***	***Description***
A	2020-06-08	First release

Copyright 2020 © Embedded Artists AB

2 Introduction

This document provides you with information and instructions for different topics that are related to the U-boot bootloader and the Linux kernel.

Additional documentation you might need is.

- The *Getting Started* document for the board you are using.
- *Working with Yocto*

2.1 Conventions

A number of conventions have been used throughout to help the reader better understand the content of the document.

`Constant width text` – is used for file system paths and command, utility and tool names.

```
$ This field illustrates user input in a terminal running on the
development workstation, i.e., on the workstation where you edit,
configure and build Linux
```

```
# This field illustrates user input on the target hardware, i.e.,
input given to the terminal attached to the COM Board
```

```
This field is used to illustrate example code or excerpt from a
document.
```

Copyright 2020 © Embedded Artists AB

3 Boot process

3.1 Overview

Figure 1 below illustrates the boot process on a high level.

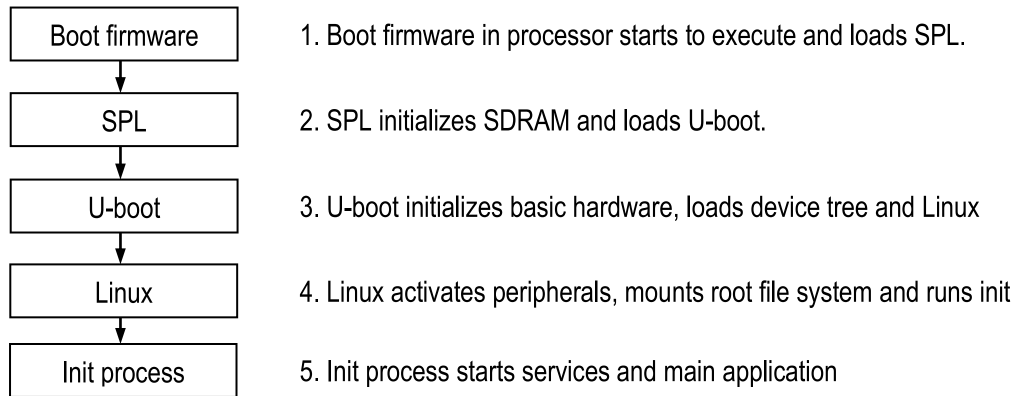

Figure 1 - Overview of boot process

3.2 Processor boot firmware

When power is turned on or when a reset signal is received the processor will start to execute its boot firmware (also called boot ROM code). The main purpose of the boot firmware is to load a program and then run it on the application processor. The firmware determines where to load the program from by looking at the state of the boot mode register, fuses, and/or GPIOs (General Purpose Input Output pins).

The boot firmware is on a high level identical between i.MX processors, but can be slightly different on a more detailed level, for example, different processors might support different boot devices. The details can be found in NXP's user's manual for the processor you are using. For the i.MX 8M Mini application processor this is described in section 6.1 – System Boot (Rev 2 of the manual).

3.2.1 Boot mode register

The boot firmware begins by checking the state of the boot mode register and will continue its execution based on this state. On Embedded Artists COM boards, a boot control mechanism consisting of the signals BOOT_CTRL and ISP_ENABLE has been implemented to configure the boot mode register. The details of the boot control mechanism can be found in the **datasheet** of the COM board you are using, but a short summary is available below.

During the development phase you will most often switch between programming the board via the Universal Update Utility (UUU) and booting from eMMC flash. This is handled by the ISP_ENABLE signal and more specifically the J2 jumper on the COM Carrier board. Putting the J2 jumper in a closed state will enable the serial downloader (USB OTG) boot mode which will allow UUU to program the board. Putting J2 in opened state will instead enable internal boot which by default has been setup to boot from eMMC.

If you instead want to boot using fuses you need to put BOOT_CTRL in floating state which is accomplished by setting the J27 jumper in opened state.

3.2.2 GPIO boot

As mentioned in the previous section the default setup for Embedded Artists COM boards is to boot from eMMC. This is accomplished by either using switches or zero-ohm resistors to control pins on the processor so that the boot configuration register is set to select eMMC as boot device. See the **datasheet** for the COM board for more details.

Copyright 2020 © Embedded Artists AB

For the i.MX 8M Mini processor GPIO boot overrides is described in chapter 6.1.3.2 in NXP's User's Manual (Rev 2). Similar chapters exist for other processors in their respective manual.

3.2.3 Fuses

Embedded Artists COM boards are normally delivered without any programmed fuses so you as a customer have full control of these. In an end product it is common, and **NXP recommends**, to control the boot process by programming the fuses.

If you want to use fuses you need to set J27 on the COM carrier board in open state.

For the i.MX 8M Mini processor you will find more information in chapters 6.1.3.1 – Boot eFuse Descriptions and 6.2 - Fusemap in NXP's User's Manual (Rev 2). Similar chapters exist for other processors in their respective manual.

3.3 SPL

As described in section 3.2 a COM board defaults to boot from eMMC flash. The boot firmware will read an Image Vector Table (IVT) from a fixed offset in the selected boot partition of the eMMC flash. The offset can be different for different processors as can be seen in Table 1 below. The user's manual for the processor specifies the offset for different boot devices. For the i.MX 8M Mini processor this is described in chapter 6.1.6.1 – Image Vector Table and Boot data in NXP's User's Manual (Rev 2).

Processor family	IVT offset - eMMC
i.MX6	1 Kbyte
i.MX7	1 Kbyte
i.MX8M and i.MX8M Mini	33 Kbyte
i.MX8M Nano	0, if the image is in boot partition and 32K if it is in user partition

Table 1 - IVT offset for eMMC flash

Information from the IVT will be used to load the remainder of the image and also where it should be loaded (internal RAM in this case).

For the default setup of an Embedded Artists COM board this image will be **SPL** (short for Secondary Program Loader). SPL is part of the **U-boot** source code and can be seen as a small subset of U-boot. The U-boot itself would in normal cases be too big to be loaded to internal RAM and that is why a subset is used. When SPL is built the IVT will also be generated and added at the beginning of the final SPL image.

SPL will be responsible for **initializing the external RAM**, load the U-boot to external RAM and then hand over execution to U-boot.

Board specific SPL code is available in a file called `spl.c` located in the board directory, see link below for the iMX8M Mini uCOM board.

https://github.com/embeddedartists/uboot-imx/blob/ea_v2018.03/board/embeddedartists/mx8mmea-ucom/spl.c

3.4 U-boot

The bootloader used for Embedded Artists COM boards is U-boot, also known as Universal Boot Loader or Das U-Boot. This is an open-source bootloader commonly used on many different architectures and platforms.

http://www.denx.de/wiki/U-Boot

U-boot's main responsibility is to **load the Linux kernel**, select and load the device tree (see Chapter 4 below) and hand it over the device tree to the kernel. In order to do this the U-boot has to do some

Copyright 2020 © Embedded Artists AB

initial hardware initialization such as basic processor (CPU) setup, initialize clocks and timers, initialize console, optionally the display and board specific initialization. Table 2 highlights some of the functions part of the initialization flow.

`_main`	`arm/lib/crt0.S` `arm/lib/crt0_64.S`	Main function called by C runtime.
`board_init_f`	`common/board_f.c`	Prepares the hardware for execution. Will for example call `arch_cpu_init` to initialize CPU, but also the board specific function `board_early_init_f`.
`board_init_r`	`common/board_r.c`	SDRAM is initialized and global variables are available when this function is called. SDRAM is initialized. It will call functions such as `board_init`, `initr_mmc`, `console_init_r` and finally `run_main_loop`.
`main_loop`	`common/main.c`	It is in this function U-boot will start to process commands, such as the commands defined in the U-boot environment. For auto booting U-boot will run the command(s) defined in the configuration variable `CONFIG_BOOTCOMMAND` https://github.com/embeddedartists/uboot-imx/blob/ea_v2018.03/include/configs/mx8mmea-ucom.h#L213

Table 2 - Highlighted initialization functions

3.5 Linux

Linux is the main operating system on Embedded Artists COM boards. It is an open-source kernel widely used on many embedded devices.

The Linux kernel will use the device tree provided by the U-boot to activate peripherals and load device drivers. Finally, it will mount a root file system and hand over execution to the init process. The init process can be seen as the parent of all other processes in Linux. It will for example start background processes, the console, and optionally a main application. All of this is handled via an initialization manager.

At the time of writing this document the default initialization manager used with Embedded Artists Linux distribution is `systemd`. Previously it used to be `SysV` (System V Init).

3.5.1 Initialization manager – systemd

Systemd is a suite of components that is used to initialize and configure a Linux system. There are utility applications used to monitor and control the system and there are init scripts for the different services within the system.

We won't go into any detail of all the aspects of systemd. There are several useful resources available and below are a few of these.

- https://en.wikipedia.org/wiki/Systemd
- https://www.linux.com/training-tutorials/understanding-and-using-systemd/
- https://www.freedesktop.org/wiki/Software/systemd/

Copyright 2020 © Embedded Artists AB

3.6 Frequently asked questions

3.6.1 Can I boot from SD/MMC card instead of eMMC?

It depends on what you mean by booting and what you want to put on the SD/MMC card. There are instructions in the Embedded Artists document "*Working with Yocto*" that show you how to put the **root file system** on an SD/MMC card.

You would have to modify the hardware or burn fuses, see section 3.2.3 if you would like to use something other than eMMC as primary boot device. Our recommendations are however to keep using eMMC as primary boot device.

3.6.2 How do I launch an application at startup?

This is a question related to systemd as described in section 3.5.1 above. You need to create a service file, for example, `myapplication.service` that you put in `/etc/systemd/system`. Below is an example how this file could look like.

```
[Unit]
Description=Launch my application
After=multi-user.target

[Service]
Type=simple
ExecStart=/usr/bin/myapplication

[Install]
WantedBy=multi-user.target
```

Enable the service by using `systemctl` so that it starts at next boot. You can also start it using `systemctl`.

```
# systemctl enable myapplication.service
# systemctl start myapplication.service
```

4 Device Tree

4.1 Introduction

The device tree is a data structure used for describing hardware. Take a Computer-on-Module (COM) such as the *iMX8M Mini uCOM* board as an example. It consists of several hardware devices (peripherals) such as i.MX 8M Mini application processor, LPDDR4 memory, eMMC flash, Ethernet PHY, and Power Management IC (PMIC), see Figure 2 for a block diagram.

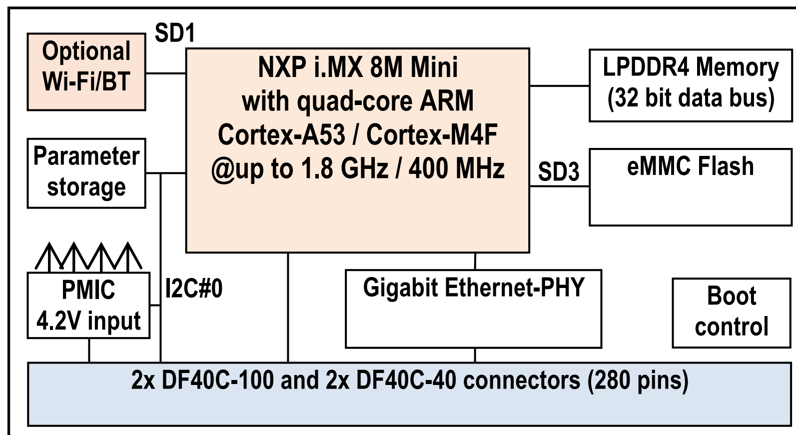

Figure 2 - iMX8M Mini uCOM block diagram

The iMX8M Mini uCOM board is then used in combination with a carrier board such as the *COM Carrier board v2* used in a Developer's Kit. This carrier board adds more peripherals and especially interfaces such as UART, USB, PCIe, SD card, audio codec, and more. To be even more specific the i.MX 8M application processor in itself also contains several peripherals such as I2C, SPI, PCIe, MIPI-DSI, memory buses, and more.

To be able to use a hardware device within an operating system, (we will use Linux kernel as example), a **device driver** is needed. The driver needs to be initialized to work with the specific hardware / board configuration. This could for example be a device address (if attached to a bus), pin configuration, clock to assign to the device, and so on.

Before device trees, the Linux kernel often contained the **board specific code,** such as the address of the device. This meant that the kernel had to be re-compiled when something hardware specific had to be changed. Take the PMIC as an example. This device is attached to the I2C bus at a specific I2C address (0x4B) and if the address had to be changed a new kernel had to be compiled. Another, but related problem, is if you offer different configurations of your product. In this case you had to provide different Linux kernels for each configuration in the case when the kernel contained board specific code.

The device tree solves these problems by moving board / device specific code out of the kernel and into the device tree file. This file can then be maintained and compiled separate from the kernel which will make the kernel more portable across different boards. In the example of using the Linux kernel the device tree will typically be loaded by the U-boot bootloader.

It is important to note that a device tree won't be needed for **discoverable** devices such as devices attached to a USB bus. The USB protocol has been designed to be able to detect if a device is attached and then probe that device for information that is given to the device driver.

4.2 Data structure format

The data structure can be seen as a tree structure with nodes and properties. Each node (except the root node) has one parent in the tree. A node contains a list of properties, which are key-value pairs,

Copyright 2020 © Embedded Artists AB

and can also contain child nodes. This kind of structure makes it quite easy for a human to read and understand how the hardware is organized.

The latest device tree specification can be found at https://www.devicetree.org/specifications/.

Figure 3 shows an example of a node that describes an I2C bus. In this example the node has been assigned a label (`i2c1`). This label can be used to reference the node from other parts of the device tree (instead of referencing the node name). The node name must be unique and should describe the general class of the device. In the example below the node name (`i2c@30a20000`) consists of a name (`i2c`) followed by a unit-address (`@30a20000`). The unit-address is omitted if there is no address associated with the device. In the example below you can see that the unit-address is also specified in the `reg` property.

One important property is the `status` property. Below you can see that it is set to `disabled` which means that the device won't be activated in the kernel.

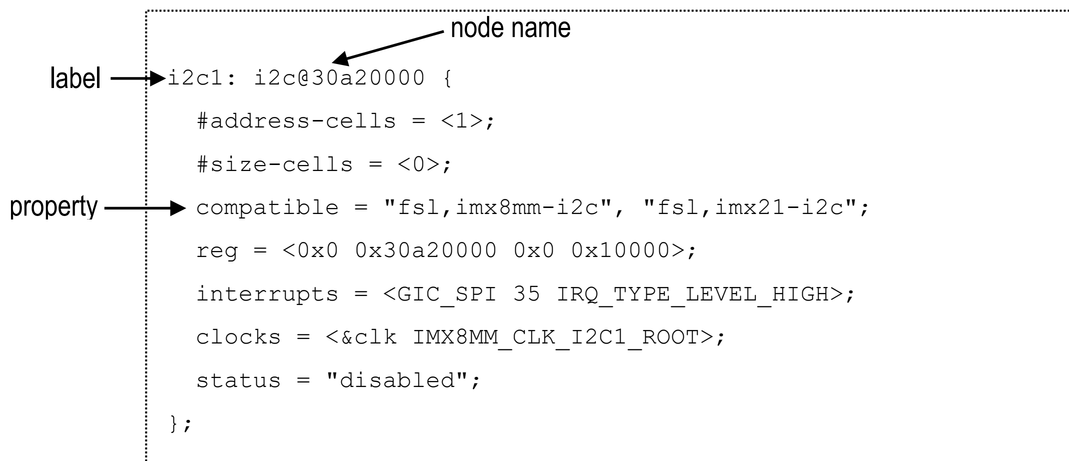

Figure 3 – Example of a device tree node

Figure 4 shows an example where the `i2c1` node is referenced and properties added and changed. The `status` property is for example set to `okay` meaning that the I2C1 bus is activated. A child node (`pmic`) is added to the node since the actual PMIC device is attached to the I2C1 bus on the iMX8M Mini uCOM board. For the PMIC device you can see that the I2C address is 0x4b (both set in the unit-address field and the `reg` property).

Copyright 2020 © Embedded Artists AB

Figure 4 – Continued example of device tree nodes

4.2.1 Property values

As previously mentioned, properties consist of key-value pairs and the key (the property name) is a string of 1 to 31 characters. The value can however be of different types.

- Empty value. This indicates a true / false property and only contains the property name.
  - `enable-active-high;`
- String value.
  - `status = "okay";`
- String list. Strings are separated with comma (‘,’).
  - `compatible = "rohm,bd71840", "rohm,bd71837";`
- Cell property array. The format is specific to the property. Below are two examples where the `reg` property contains one hexadecimal integer value, while the `gpio_intr` property contains one reference to another node and two integer values.
  - `reg = <0x4b>;`
  - `gpio_intr = <&gpio1 3 GPIO_ACTIVE_LOW>;`
- Binary data. The data is delimited with square brackets.
  - `local-mac-address = [00 1A F1 01 04 15];`

4.2.2 Aliases

There is a special node in the device tree called `aliases`. This node can be used to assign a short alias to a full node path. This alias can then be used within the Linux kernel or U-boot when accessing a node instead of using the full path to the node. Aliases **are not used** within the device tree source (`.dts`) to reference a node. Instead, labels are used within the device tree source.

The example below, which is from `imx8mm.dtsi` can look quite confusing since for example the alias `i2c0` reference `i2c1`. The right-hand side is however in this case a label as illustrated in Figure

2 and means that the alias `i2c0` will have the value `/i2c@30a20000`, i.e., the full path to the I2C node.

```
    aliases {
            ethernet0 = &fec1;
            i2c0 = &i2c1;
            i2c1 = &i2c2;
            i2c2 = &i2c3;
            i2c3 = &i2c4;
            serial0 = &uart1;
...
```

The complete alias section for the iMX8M Mini uCOM can be seen by following the link below.

https://github.com/embeddedartists/linux-imx/blob/ea_4.14.98/arch/arm64/boot/dts/freescale/fsl-imx8mm.dtsi#L30

4.3 Source files and compiler

There are two types of source files used for device trees.

- `.dtsi` – includable device tree file. This file is not considered a standalone device tree, but instead it is included in either another `.dtsi` file or in a .dts file that will complete the device tree. These types of files are typically used when describing the application processor, such as `fsl-imx8mm.dtsi` for the i.MX 8M Mini application processor.
- `.dts` – This is a standalone device tree that can be compiled into a binary file (`.dtb`). Please note that a `.dts` file can include another `.dts` file. The file to include doesn't have to be a `.dtsi` file.

A device tree source file is compiled into a binary device tree (`.dtb`) by using the device tree compiler (`dtc`). Most often you won't use this compiler directly, but instead build the `.dtb` files within the Linux kernel. In this case you will use the `dtbs` target.

```
$ make dtbs
```

4.3.1 Linux

Within the Linux kernel the device tree files are available under the `arch` directory. Below you can see the exact location (for version 4.14.98) depending on which target you are using.

iMX6 and iMX7:

https://github.com/embeddedartists/linux-imx/tree/ea_4.14.98/arch/arm/boot/dts

iMX8:

https://github.com/embeddedartists/linux-imx/tree/ea_4.14.98/arch/arm64/boot/dts/freescale

4.3.2 U-boot

Within the U-boot bootloader the device tree files are available under the `arch/dts` directory. Below you can see the exact location (for version 2018.03).

https://github.com/embeddedartists/uboot-imx/tree/ea_v2018.03/arch/arm/dts

4.4 Pin muxing

A pin on an i.MX application processor may have more than one function, that is, it can be connected to more than one internal peripheral (but only one at a time). The selection of pin function is handled by an input-output multiplexer usually called IOMUX. Besides selecting which function to use the multiplexer is also used to configure other characteristics such as drive strength, hysteresis, open drain, pull-up/pull-down, and so on.

Pin muxing is handled in the device tree in a node called `iomuxc`. In this node you configure a pin within a child node that must contain the property `fsl,pins` with a value consisting of several cells.

If we take the I2C node in Figure 4 as example you can see that it has a property called `pinctrl-0` that has a reference to a node called `pinctrl_i2c1`. The node `pinctrl_i2c1` is a child node to `iomuxc`.

```
pinctrl-0 = <&pinctrl_i2c1>;
```

Below is the `pinctrl_i2c1` node with the `fsl,pins` property and a list of two pins being configured. The first part of a row is a pre-processor macro that will be expanded to several cells. For the i.MX 8M Mini this macro is defined in `include/dt-bindings/pinctrl/pins-imx8mm.h`. For i.MX6 and i.MX7 corresponding files are located in `arch/arm/boot/dts`. In general, you don't need to know the exact values set in the macro, but only how to interpret the name of the macro since a specific naming convention is being used.

The name consists of three parts:

- `MX8MM_IOMUX`: A prefix which should be unique and usually identifies the processor.
- `I2C1_SCL`: The pad name (in the NXP manual it is usually referred to as a pad instead of a pin) on the processor which normally is the same as the main function of the pad.
- `I2C1_SCL`: The function the pad should get. In this example this is the same as the pad name, but if you look in `pins-imx8mm.h` you can see that it could have been set to for example `GPIO5_IO14` if you wanted it to be configured to be a GPIO.

```
pinctrl_i2c1: i2c1grp {
  fsl,pins = <
    MX8MM_IOMUXC_I2C1_SCL_I2C1_SCL             0x400001c3
    MX8MM_IOMUXC_I2C1_SDA_I2C1_SDA             0x400001c3
  >;
};
```

The second part of a row is a hexadecimal value that sets the pad control registers – the characteristics of the pad. The user's manual for the processor must be consulted to interpret this value. If we continue with the I2C example for the i.MX 8M Mini processor and look in the manual (Rev 2 was used when writing this document) we can find a description of the control register in section 8.2.5.283 – Pad Control Register (IOMUXC_SW_PAD_CTL_PAD_I2C1_SCL).

The table from that section is replicated below and a third column has been added with a description of the specific value used for the I2C1_SCL pad.

Field	Description	I2C1_SCL value
31-9 -	This field is reserved	-
8 PE	Pull resistors enable field 0 = Disable pull resistors	1 = Enable pull resistors

	1 = Enable pull resistors	
7 HYS	Hysteresis enable field 0 = Select CMOS input 1 = Select Schmitt input	1 = Schmitt input
6 PUE	0 = Select pull-down resistors 1 = Select pull-up resistors	1 = Pull-up resistors enabled
5 ODE	Open drain enable field 0 = Disable open drain mode 1 = Enable open drain mode	0 = Open drain disabled
4-3 FSEL	Slew rate field (lsb field not used hence the X below) 0X – Select slow slew rate (SR=1) 1X – Select fast slew rate (SR=0)	00 = Slow slew rate
2-0	Drive strength field (lsb field not used hence the X below) 00X – Drive strength X1 10X – Drive strength X2 01X – Drive strength X4 11X – Drive strength X6	011 = Drive strength X4

Table 3 - I2C1_SCL pad control register

4.5 Modify device tree from U-boot

The U-boot is responsible for loading the device tree and providing it to the Linux kernel. The U-boot also has the `fdt` command that can be used to parse and modify the device tree before it is provided to Linux. By using the `fdt` command, you can make temporary changes to the device tree without having to modify and re-compile the `.dts` file.

4.5.1 Usage

From within the U-boot console run the command below to get a description of which `fdt` commands that are available.

```
=> help fdt

Usage:
fdt addr [-c]  <addr> [<length>]   - Set the [control] fdt location to
<addr>
fdt boardsetup                      - Do board-specific set up
fdt systemsetup                     - Do system-specific set up
fdt move   <fdt> <newaddr> <length> - Copy the fdt to <addr> and make it
active
...
```

The commands that you will most often use are print, set, rm and possibly mknod.

4.5.2 Test different commands

This section describes how you can test some of the `fdt` commands.

Load the device tree

The device tree must be loaded before you can modify it. Do this by setting `skip_booting` and then running `boot`.

Copyright 2020 © Embedded Artists AB

```
=> setenv skip_booting yes
=> boot
switch to partitions #0, OK
mmc1(part 0) is current device
1492 bytes read in 7 ms (208 KiB/s)
Running bootscript from mmc ...
## Executing script at 40480000
39658 bytes read in 9 ms (4.2 MiB/s)
!!!! Selected to skip booting !!!!
!!!! Unset skip_booting variable to enable booting again !!!!
```

Print the device tree

When the device tree has been loaded you can inspect it by using `fdt print`. You enter a path to the part of the tree you want to print. If you want to see the entire tree you print the root ('/'). You can also use `fdt list` if you just want to print one level of nodes, for example, all children just beneath the root, but not any of the child nodes children.

Since the device tree is usually quite large it is better to only print the part you are interested in. You need to specify the complete path to the node or if an alias exist use that alias. If we take the I2C node described in section 4.2 and list its content it looks like below. The I2C node is available directly under the root node.

```
=> fdt list /i2c@30a20000
i2c@30a20000 {
        #address-cells = <0x00000001>;
        #size-cells = <0x00000000>;
        compatible = "fsl,imx8mm-i2c", "fsl,imx21-i2c";
        reg = <0x00000000 0x30a20000 0x00000000 0x00010000>;
        interrupts = <0x00000000 0x00000023 0x00000004>;
        clocks = <0x00000004 0x000000a4>;
        status = "okay";
        clock-frequency = <0x00061a80>;
        pinctrl-names = "default";
        pinctrl-0 = <0x0000001b>;
        bd71837@4b {
        };
        at24@55 {
        };
        wm8731@1a {
        };
};
```

You can get the same result by using the alias `i2c0`. See section 4.2.2 for information about aliases.

```
=> fdt list i2c0
...
```

Change the value of a property

Use `fdt set` to change the value of a property. In this example we will deactivate the audio codec (wm8731) which we could see as a child node to `i2c@30a20000` in the example above.

```
=> fdt set /i2c@30a20000/wm8731 status "disabled"
```

Boot Linux with the modified device tree

If you want to test the modifications you have done you cannot run just `boot` since this would mean that the device tree would be re-loaded. Instead you need to run some of the individual commands part of the boot process. In most cases this involves loading the image, setting the mmc arguments and then issuing the boot command. Which boot command to use can differ (`booti` or `bootz`) for different boards, but you can find it by inspecting the `bootcmd` variable. For the iMX8M Mini uCOM it looks like below.

```
=> run loadimage
=> run mmcargs
=> booti ${loadaddr} - ${fdt_addr}
```

4.5.3 Add to `cmd_custom`

If you want the modification of the device tree to be more permanent, that is, it should be done for consecutive boots without having to manually enter the `fdt` commands, you can add the changes to the `cmd_custom` variable.

```
=> setenv cmd_custom fdt set /i2c@30a20000/wm8731 status "disabled"
=> saveenv
```

If you need to run several `fdt` commands you can separate them with a semicolon (';').

4.6 Frequently asked questions

4.6.1 Do I need to create my own device tree file?

If you are developing your own product by using one of Embedded Artists COM boards you will most likely need to develop your own carrier board. Your carrier board will contain the specific peripherals and interfaces needed by your product. This mean that you have to use your own `.dts` file that defines the peripherals you are using.

4.6.2 How do I create my own device tree file?

The recommendation is that you start with one of the files (there can be more than one) that has been created for the Embedded Artists Developer's Kit you are using, for example, `fsl-imx8mm-ea-ucom-kit_v2.dts` if you are using iMX8M Mini uCOM.

1. Copy our `.dts` file and give it a new name for your product.
2. Modify your file so it matches your hardware. This usually involves removing nodes that you don't need, but also adding new nodes for peripherals that are new and specific to your hardware.
3. Add the new `.dts` file to the `Makefile` so that it will be built. Below is a link to the `Makefile` used for `fsl-imx8mm-ea-ucom-kit_v2.dts`
    a. https://github.com/embeddedartists/linux-imx/blob/ea_4.14.98/arch/arm64/boot/dts/freescale/Makefile#L142
4. The U-boot environment has a variable named `fdt_file` that defines which `.dtb` file to load. You have to either update this variable dynamically using `setenv` or change the default setting in the U-boot source code.
    a. Change dynamically in U-boot environment:

```
=> setenv fdt_file fsl-imx8mm-my_device.dtb
=> saveenv
```

Copyright 2020 © Embedded Artists AB

b. Change default values statically in U-boot source. The link below shows where and how it is configured for iMX8M Mini uCOM. It is done in a similar way for the other COM boards.

https://github.com/embeddedartists/uboot-imx/blob/ea_v2018.03/include/configs/mx8mmea-ucom.h#L166

5. If you want your new file to be included in a Yocto image you must update the machine file. The link below goes to the machine file for the iMX8M Mini uCOM board. It is similar for the other COM boards.

   https://github.com/embeddedartists/meta-ea/blob/ea-4.14.98/conf/machine/imx8mmea-ucom.conf#L22

6. If you want the new file to be copied to the target when running the UUU tool you must update the uuu scripts (can be downloaded from http://imx.embeddedartists.com). The example below is from the `full_tar.uuu` file for the iMX8M Mini uCOM board.

```
# Copy kernel and dtb files
FBK: ucp  files/Image-imx8mmea-ucom.bin t:/mnt/fat/Image
FBK: ucp  files/fsl-imx8mm-ea-ucom-kit_v2.dtb t:/mnt/fat
FBK: ucp  files/fsl-imx8mm-ea-ucom-kit_v2-1mw.dtb t:/mnt/fat
FBK: ucp  files/fsl-imx8mm-ea-ucom-kit_v2-m4.dtb t:/mnt/fat
FBK: ucp  files/fsl-imx8mm-ea-ucom-kit_v2-pcie.dtb t:/mnt/fat
FBK: ucp  files/boot.scr t:/mnt/fat
FBK: ucmd umount /mnt/fat
```

4.6.3 How do I find the device driver?

Given a node in the device tree, how do I find the associated device driver in the Linux kernel? The short answer is that you need to look at the compatible property and find a driver that match one of the strings in the string list.

Let’s take the PMIC node shown in Figure 4 as an example. The compatible property looks like below.

```
compatible = "rohm,bd71840", "rohm,bd71837";
```

There are two strings in the string list indicating that the same driver could be used for several versions of the PMIC. In the Linux kernel source directory search for one of these strings. As you can see below the string can be found in `mfd/bd7183.c`.

```
$ cd linux-imx/drivers
$ grep -r "rohm,bd71840" *
mfd/bd71837.c:  { .compatible = "rohm,bd71840", .data = (void *)0},
```

If you open this file you can see the device table below which lists both `"rohm,bf71837"` and `"rohm,bd71840"` as compatible devices. Note that only one of the strings in the compatible property must match a string in the device table. In this example both string were included in the table.

```
static struct of_device_id bd71837_of_match[] = {
        { .compatible = "rohm,bd71837", .data = (void *)0},
        { .compatible = "rohm,bd71840", .data = (void *)0},
        { },
};
```

4.6.4 How do I find which properties I can use?

When you need to add a new node in the device tree for a new hardware device you also need to know which properties to use.

The first step is to try to find already existing device tree files using the same kind of device. Search for the device in the kernel sources. If you find existing examples you can use this as a starting point, but should also double-check the actual device driver since the examples could be out-dated.

Find the device driver as described in section 4.6.3 and open that file. Look for functions beginning with `of_` such as `of_property_read_u32` or `of_get_named_gpio`.

For `mfd/bd71837.c` that was given as an example in section 4.6.3 you can find such functions in `bd71837_parse_dt`, see below for an excerpt where the `gpio_intr` property is retrieved.

```
board_info->gpio_intr = of_get_named_gpio(np, "gpio_intr", 0);
if (!gpio_is_valid(board_info->gpio_intr)) {
  dev_err(&client->dev, "no pmic intr pin available\n");
  goto err_intr;
}
```

4.6.5 Can I modify the device tree without re-compiling?

See section 4.4 for a way to do this from within the U-boot console.

4.6.6 Can I delete a node / property?

Yes, if you want to delete an already defined node or a property from a new `.dts` file you can do this by `/delete-node/` or `/delete-property/`, see below for examples.

```
&gpio1 {
        /delete-node/ sd1_vselect_gpio;
};
```

```
&gpio_buff {
    hog_DIR_WL_DEV_WAKE {
          gpio-hog;
          gpios = <9 0>;
          /delete-property/ output-low;
          output-high;
    };
};
```

4.6.7 Where do I find more documentation about device tree usage?

The links below contain a lot of useful information about device trees.

- https://elinux.org/Device_Tree_Usage
- https://elinux.org/Device_Tree_Mysteries
- https://elinux.org/Device_Tree_Source_Undocumented

Copyright 2020 © Embedded Artists AB

5 Customization

5.1 U-boot

Most projects can use the Embedded Artists U-boot with little or no modification. The most common modification is to change the default `.dtb` file that will be loaded. Some projects might need additional early initialization of hardware and will put this in the board specific file of the U-boot.

5.1.1 Board specific files

If modification is needed it can in most cases be limited to changes in the board specific file which is located in a sub-directory to `<uboot-dir>/board/embeddedartists`. Table 4 lists board specific files for the different COM boards.

https://github.com/embeddedartists/uboot-imx/tree/ea_v2018.03/board/embeddedartists

COM board	Board specific file
iMX6 UltraLite COM	mx6ulea-com/mx6ulea-com.c
iMX6 SoloX COM	mx6sxea-com/mx6sxea-com.c
iMX6 Quad	mx6qea-com/mx6qea-com.c
iMX6 DualLite COM	mx6qea-com/mx6qea-com.c
iMX7 Dual COM	mx7dea-com/mx7dea-com.c
iMX7 Dual uCOM	mx7dea-com/mx7dea-com.c
iMX7ULP uCOM	mx7ulpea-ucom/mx7ulpea-ucom.c
iMX8M Nano uCOM	mx8mmea-ucom/mx8mmea-ucom.c
iMX8M Mini uCOM	mx8mnea-ucom/mx8mnea-ucom.c
iMX8M COM	mx8mqea-com/mx8mqea-com.c

Table 4 - U-boot board specific files

5.1.2 Device tree files

The U-boot is also utilizing device tree files like the Linux kernel. The support for device trees is however (for version 2018.03) not as extensive as it is for the Linux kernel. Not all drivers have full support for device trees. If you need to change initialization of a peripheral or add a new peripheral that should be initialized in the U-boot you might also need to modify the device tree file used in the U-boot.

Device tree files are located in the directory `<uboot-dir>/arch/arm/dts`.

https://github.com/embeddedartists/uboot-imx/tree/ea_v2018.03/arch/arm/dts

Table 5 lists the device trees used for Embedded Artists COM boards.

COM board	Device tree file
iMX6 UltraLite COM	imx6ulea-com-kit.dts
iMX6 SoloX COM	imx6sxea-com-kit.dts
iMX6 Quad	imx6qea-com-kit.dts
iMX6 DualLite COM	imx6dlea-com-kit.dts
iMX7 Dual COM	imx7dea-com-kit.dts
iMX7 Dual uCOM	imx7dea-ucom-kit.dts

Copyright 2020 © Embedded Artists AB

iMX7ULP uCOM	imx7ulpea-ucom-kit_v2.dts
iMX8M Nano uCOM	fsl-imx8mn-ea-ucom-kit_v2.dts
iMX8M Mini uCOM	fsl-imx8mm-ea-ucom-kit_v2.dts
iMX8M COM	fsl-imx8mq-ea-com-kit_v2.dts

Table 5 - Device tree files for Embedded Artists COM boards

5.1.3 Configuration files

The configuration of a U-boot is divided into two files. Historically it used to be an include file that contained all configurations, but more and more are moved to a defconfig file utilizing the same kind of configuration infrastructure (Kconfig) as in the Linux kernel. You will typically modify the defconfig file if you would like to add support for more U-boot commands or drivers. Defconfig files are located in <uboot-dir>/configs/, see Table 6 for the files that are used by Embedded Artists COM boards.

Follow the link below to see the defconfig file for the iMX8M Mini uCOM board.

https://github.com/embeddedartists/uboot-imx/blob/ea_v2018.03/configs/mx8mmea-ucom_defconfig

COM board	Defconfig
iMX6 UltraLite COM	configs/mx6ulea-com_defconfig
iMX6 SoloX COM	configs/mx6sxea-com_defconfig
iMX6 Quad	configs/mx6qea-com_defconfig
iMX6 DualLite COM	configs/mx6dlea-com_defconfig
iMX7 Dual COM	configs/mx7dea-com_defconfig
iMX7 Dual uCOM	configs/mx7dea-ucom_defconfig
iMX7ULP uCOM	configs/mx7ulpea-ucom_defconfig
iMX8M Nano uCOM	configs/mx8mnea-ucom_defconfig
iMX8M Mini uCOM	configs/mx8mmea-ucom_defconfig
iMX8M COM	configs/mx8mqea-com_defconfig

Table 6 - Defconfig files for Embedded Artists COM boards

If you instead want to modify for example the default U-boot environment, such as change the default dtb file that is loaded, you need to do this change in the include file which is located in <uboot-dir>/include/configs/. See Table 7 for a list of all files used with Embedded Artists COM boards.

Follow the link below to see how the dtb file is defined in the U-boot environment for the iMX8M Mini uCOM board.

https://github.com/embeddedartists/uboot-imx/blob/ea_v2018.03/include/configs/mx8mmea-ucom.h#L166

COM board	Configuration file (include)
iMX6 UltraLite COM	include/configs/mx6ulea-com.h
iMX6 SoloX COM	include/configs/mx6sxea-com.h
iMX6 Quad	include/configs/mx6qea-com.h

Copyright 2020 © Embedded Artists AB

iMX6 DualLite COM	`include/configs/mx6qea-com.h`
iMX7 Dual COM	`include/configs/mx7dea-com.h`
iMX7 Dual uCOM	`include/configs/mx7dea-com.h`
iMX7ULP uCOM	`include/configs/mx7ulpea-ucom.h`
iMX8M Nano uCOM	`include/configs/mx8mnea-ucom.h`
iMX8M Mini uCOM	`include/configs/mx8mmea-ucom.h`
iMX8M COM	`include/configs/mx8mqea-com.h`

Table 7 - Configuration files for Embedded Artists COM boards

5.2 Linux

Customizing Linux for your product mostly involves creating or modifying device tree files, read chapter 4 for more information about device tree files. In some cases, you might also need to modify the kernel configuration and / or add new device drivers. Our Linux kernel repository is available at GitHub: https://github.com/embeddedartists/linux-imx.

5.2.1 Kernel configuration

The default Linux kernel configuration for Embedded Artists COM boards is stored in the kernel sources. There are two different files; one for iMX6 / iMX7 based COM boards and another for iMX8 based COM boards.

COM board family	Configuration file
iMX6 and iMX7	`<kernel-dir>/arch/arm/configs/ea_imx_defconfig`
iMX8	`<kernel-dir>/arch/arm64/configs/ea_imx8_defconfig`

Table 8 - Default configuration for Linux kernel

If you need to modify the default kernel configuration it is recommended to use a tool, such as `make menuconfig`, to do this instead of editing the configuration file manually. The Embedded Artists document "*Working with Yocto*" has instructions of how you can run `make menuconfig` from within Yocto or if you build the kernel directly from source code.

In the example below we want to enable dynamic printk support (`CONFIG_DYNAMIC_DEBUG`) in the kernel.

1. Run the menuconfig tool.

```
$ make menuconfig
```

2. In the menu go to Kernel hacking → printk and dmesg options → Enable dynamic printk() support
3. Click Exit a number of times and when you are asked to save the new configuration click Yes.
4. You can now build the kernel with the new configurations to make sure there are no errors.
5. When you are satisfied with the configuration changes, generate a new defconfig file.

```
$ make savedefconfig
```

6. The file will be called `defconfig` and stored in the root of the kernel sources. You can replace the current default configurations by copying this file to the location of the default configuration file as shown below (for iMX8).

Copyright 2020 © Embedded Artists AB

```
$ cp defconfig arch/arm64/configs/ea_imx8_defconfig
```

5.2.2 Device drivers

Peripherals that you need access to from Linux needs a device driver. The kernel already contains device drivers for many peripherals so the first step is to search in the kernel sources for the peripheral. Device drivers are located in `<kernel-dir>/drivers`.

In the example below we search for the Atmel MXT1664 touch controller and finds it in the `input/touchscreen` directory.

```
$ cd <kernel-dir/drivers
$ grep -r mxt1664 *
input/touchscreen/Kconfig:        module will be called atmel_mxt1664_ts.
input/touchscreen/Makefile:obj-$(CONFIG_TOUCHSCREEN_ATMEL_MXT1664) +=
atmel_mxt1664_ts.o
input/touchscreen/atmel_mxt1664_ts.c:struct mxt1664_ts {
...
```

If the device driver isn't available in the kernel you need to add it. It is out-of-scope for this document to describe how you develop a new device driver and it is actually not that common that you need to develop it from scratch anyways. The most common approach to add a missing device driver to the kernel is as follows.

1. Go to the manufacturer of the peripheral and see if they have a device driver available.
2. If the manufacturer doesn't have a device driver look in newer versions of the Linux kernel you are using. If it has been added you need to back-port it to the version of the kernel you are using.
3. Besides looking in a newer version of the kernel you are using you could look in other kernel source repositories such as the main line kernel or NXP's community kernel.
   a. https://github.com/torvalds/linux
   b. https://github.com/Freescale/linux-fslc
4. If you find a device driver you need to add it to your kernel sources which involves several steps (the exact number of steps can be different for different types of device drivers).
   a. Copy the device driver file(s).
   b. Add the new file to the relevant Makefile.
   c. Add a new configuration to relevant Kconfig file.
   d. Make sure the new configuration is enabled by adding it to the default configuration file.
   e. Make sure everything builds.

If you inspect the commit below and look for mxt1664 you can see how Atmel's touch controller driver was added.

https://github.com/embeddedartists/linux-imx/commit/5549a9a2bf234fe61db2eec5f25676e7d97fe091

6 Miscellaneous

6.1 Copy files to / from target using SCP

If you want to transfer files to a running target you can use the Secure Copy (SCP) tool. It is for example convenient to use SCP to copy the kernel or device tree files from your host computer to the target if you are doing your own kernel / device tree development.

Note: Using SCP requires that you have a **network** connection to the target.

6.1.1 Allow root to login over SSH

These instructions are based on using a default `ea-image-base` image where root is the only user on the target. By default, the user root is not allowed to login to the target over an SSH connection so we have to begin by allowing this.

1. Boot into Linux and login using user: `root`, password: `pass`.

2. Open the SSH config file.

```
# nano /etc/ssh/sshd_config
```

3. Search for `PermitRootLogin` and uncomment that line and then save the file. If you are using nano editor you search by CTRL+W and you save by CTRL+O.

```
#LoginGraceTime 2m
PermitRootLogin yes
#StrictModes yes
```

6.1.2 Get IP address of the target

Use ifconfig to get the IP address of the target. In this example the IP address is **192.168.1.92**.

```
# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:1A:F1:10:27:B8
          inet addr:192.168.1.92  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::21a:f1ff:fe10:27b8/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:15308 errors:0 dropped:8740 overruns:0 frame:0
          TX packets:198 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1501699 (1.4 MiB)  TX bytes:22795 (22.2 KiB)
```

6.1.3 Copy file to target

On your Linux based host computer run the command below to copy the file `myfile.txt` to the user root's home directory (`/home/root`). If this is the first time you transfer a file to this target you will be asked if you want to continue. Answer `yes`. Then you will also be asked for root's password which by default is `pass`.

```
$ scp myfile.txt root@192.168.1.92:/home/root

The authenticity of host '192.168.1.92 (192.168.1.92)' can't be
established.
ECDSA key fingerprint is
SHA256:S5YI1fIzYiw2f1DdOWCz73//ABpjfxwLst/aqE1f6Qp.
Are you sure you want to continue connecting (yes/no)?
```

Copyright 2020 © Embedded Artists AB

```
root@192.168.1.92's password:
myfile.txt                                    100%    6     0.0KB/s   00:00
```

6.1.4 Copy file from target

On your Linux based host computer run the command below to copy the file `myfile.txt` from the user root's home directory (`/home/root`) to your host computer. You will be asked for root's password which by default is `pass`. In the example below we copy the file to the working directory, that is, the directory where we run `scp`. You can also specify a complete path if you want to copy it to a different location.

```
$ scp root@192.168.1.92:/home/root/myfile.txt myfile.txt
root@192.168.1.92's password:
myfile.txt                                    100%   18     0.0KB/s   00:00
```

6.1.5 Use SCP on a Windows host

If you are working on a Windows host you can use an application called WinSCP to transfer files to / from the target. This is an application with a user interface and not a command line based application.

https://winscp.net/eng/index.php

When you start the application, you are asked to login. In this dialog you enter the host name, user name, and password as can be seen in Figure 5 below.

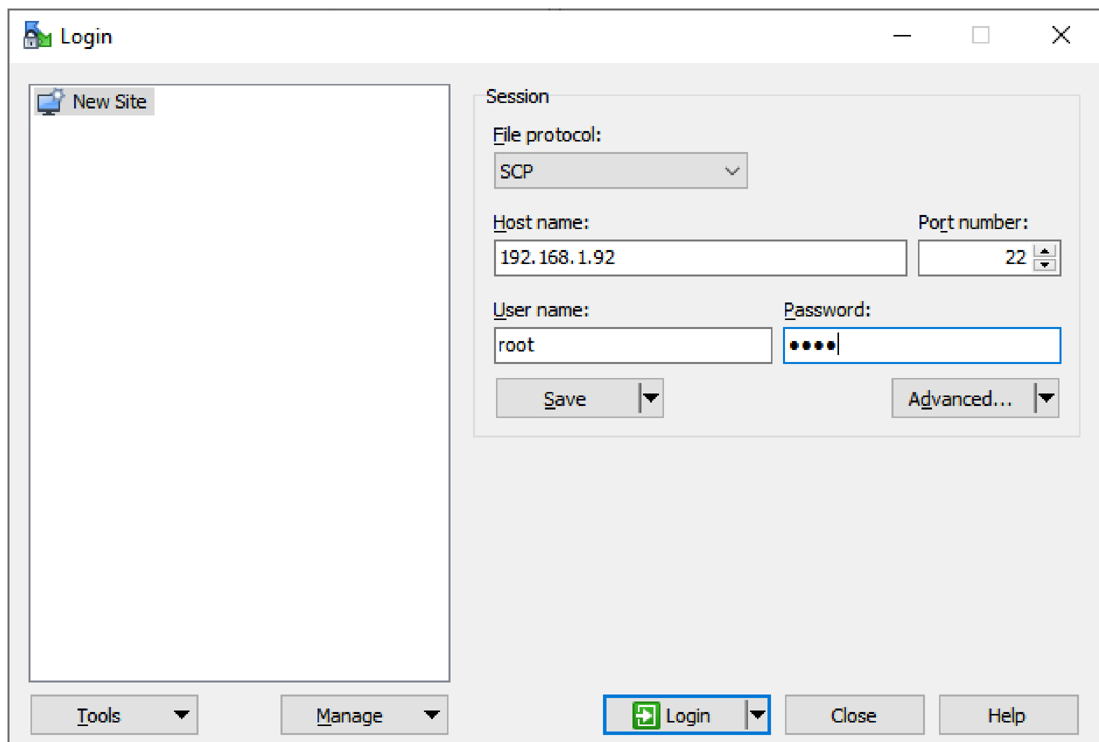

Figure 5 - WinSCP login dialog

Once you have logged in your will have view of the target file system in one panel (the right panel in the Figure 5 below). You can see the local file system in the other panel.

More information about how you use this tool can be found on the WinSCP website.

https://winscp.net/eng/docs/introduction

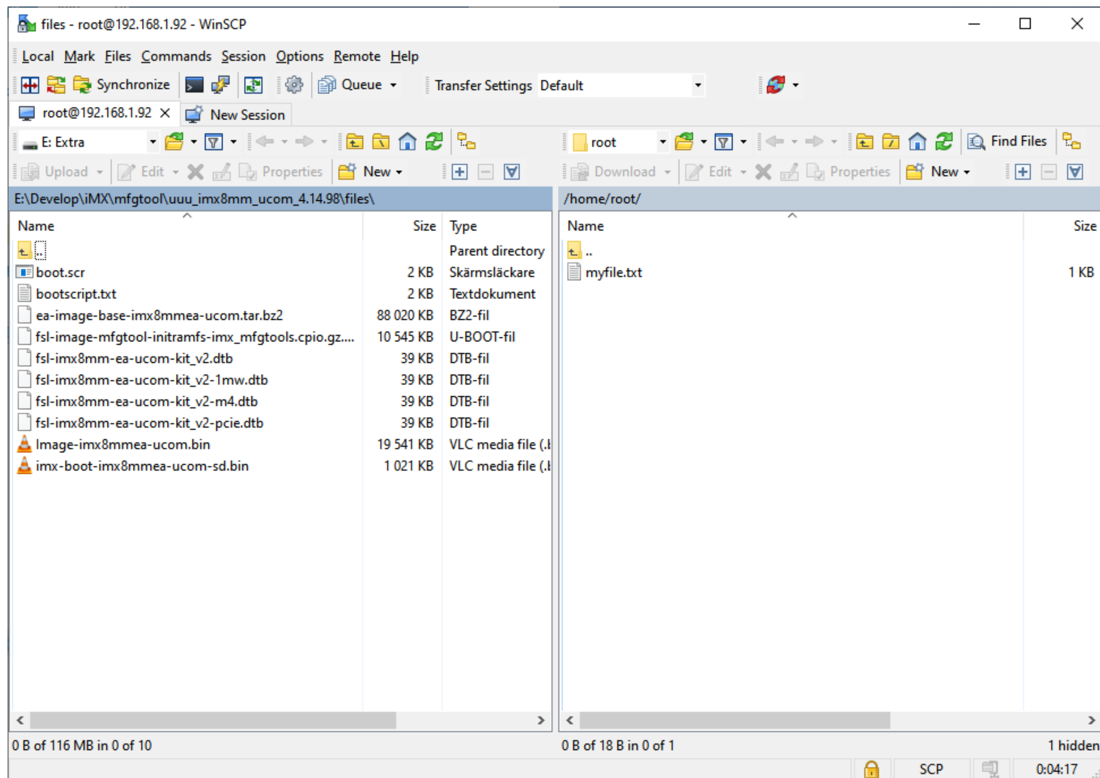

Figure 6 - WinSCP application

Copyright 2020 © Embedded Artists AB