M28 Quick Start Guide

Release 1

ARIES Embedded GmbH

August 30, 2016

CHAPTER

ONE

ABOUT THIS MANUAL

1.1 Imprint

Adress:

ARIES Embedded GmbH

Schöngeisinger Str. 84

D-82256 Fürstenfedbruck

Germany

Phone:

+49 (0) 8141/36 367-0

Fax:

+49 (0) 8141/36 367-67

1.2 Disclaimer

ARIES Embedded does not guarantee that the information in this document is up-to-date, correct, complete or of good quality. Liability claims against ARIES Embedded, referring to material or non-material related damages caused, due to usage or non-usage of the information given in this document, or due to usage of erroneous or incomplete information, are exempted, as long as there is no proven intentional or negligent fault of ARIES Embedded. ARIES Embedded explicitly reserves the rights to change or add to the contents of this Preliminary User's Manual or parts of it without notification.

1.3 Copyright

1.4 Registered Trademarks

1.5 Care and Maintenance

- Keep the device dry. Precipitation, humidity, and all types of liquids or moisture can contain minerals that will corrode electronic circuits. If your device does get wet, allow it to dry completely.
- Do not use or store the device in dusty, dirty areas. Its moving parts and electronic components can be damaged.
- Do not store the device in hot areas. High temperatures can shorten the life of electronic devices, damage batteries, and warp or melt certain plastics.
- Do not store the device in cold areas. When the device returns to its normal temperature, moisture can form inside the device and damage electronic circuit boards.
- Do not attempt to open the device.
- Do not drop, knock, or shake the device. Rough handling can break internal circuit boards and fine mechanics.
- Do not use harsh chemicals, cleaning solvents, or strong detergents to clean the device.
- Do not paint the device. Paint can clog the moving parts and prevent proper operation.
- Unauthorized modifications or attachments could damage the device and may violate regulations governing radio devices.

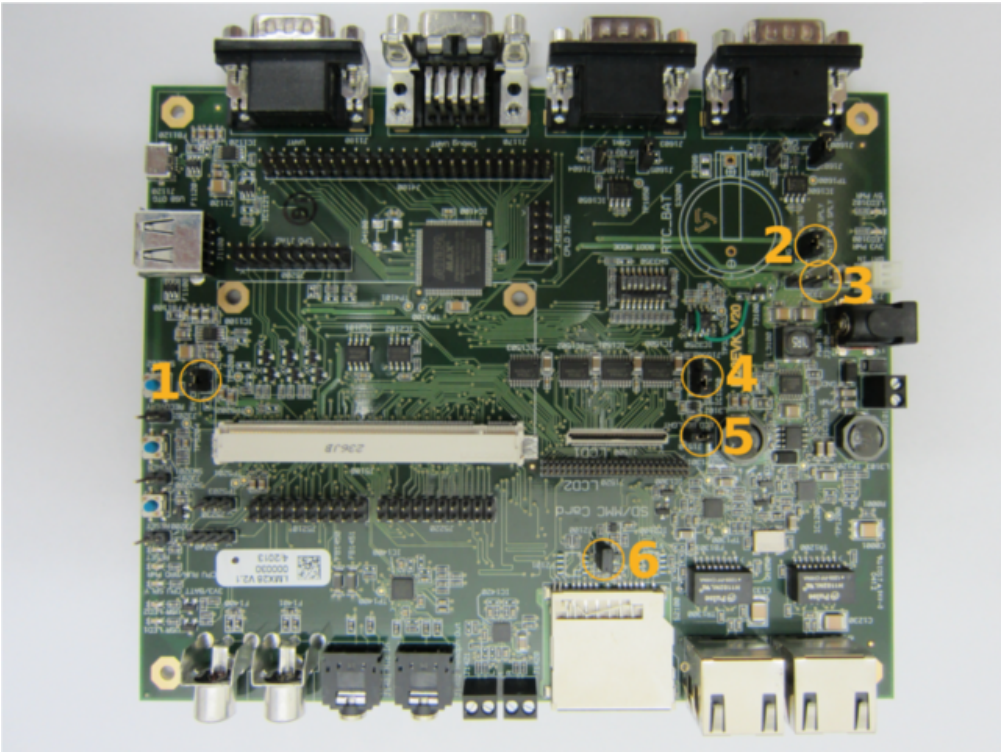
CHAPTER

TWO

INITIAL OPERATION

The following steps guide you to the correct settings of your M28EVK. The following steps assume that you will run M28 on M28EVK in 5V supply mode.

2.1 Apply Correct Jumper Settings

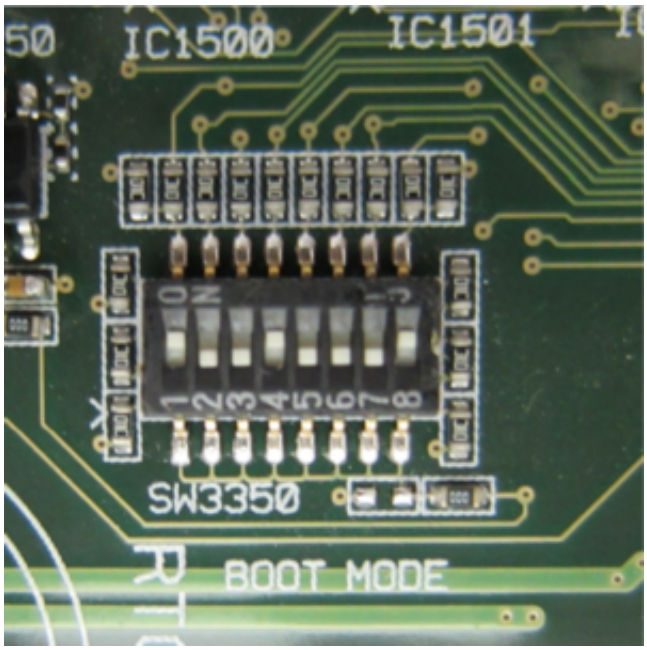


1. Place Jumper J3300 to set 5V supply mode for M28
2. Place Jumper J3301 “CRG Batt”
3. Open Jumper J3302 “3V3 CPU supply/Batt CPU supply”
4. Place Jumper J1550 to position 1-2 and leave there for position 2-3 open to enable backlight dimming via PWM4.
5. Place Jumper J1510 to enable the backlight
6. Place Jumper J2100 to power the SD-card interface

2.2 Apply Boot Mode Settings

Position of the switch is represented by 0 = OFF, 1 = ON . The bootmode configuration switches can be found above andto the right of the RTC battery slot.

Set the boot mode settings on SW3350 for SD-card boot:



2.2.1 SD-card Boot

M28EVK settings for SW3350 to boot from SD-card:

1	2	3	4	5	6	7	8
on	off	off	on	off	off	off	on

2.2.2 NAND Flash Boot

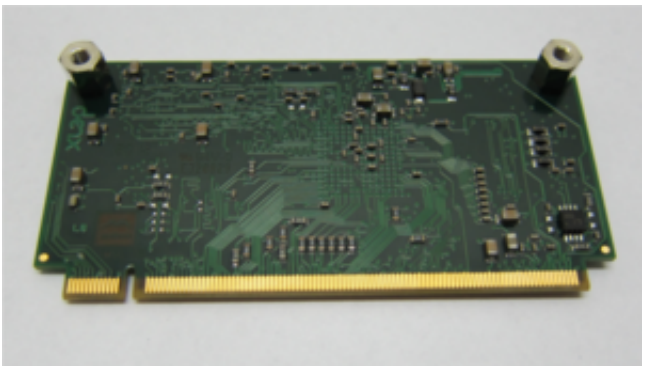
In case you want to boot from NAND flash the following settings have to be applied for SW3350:

M28EVK settings for SW3350 to boot from SD-card:

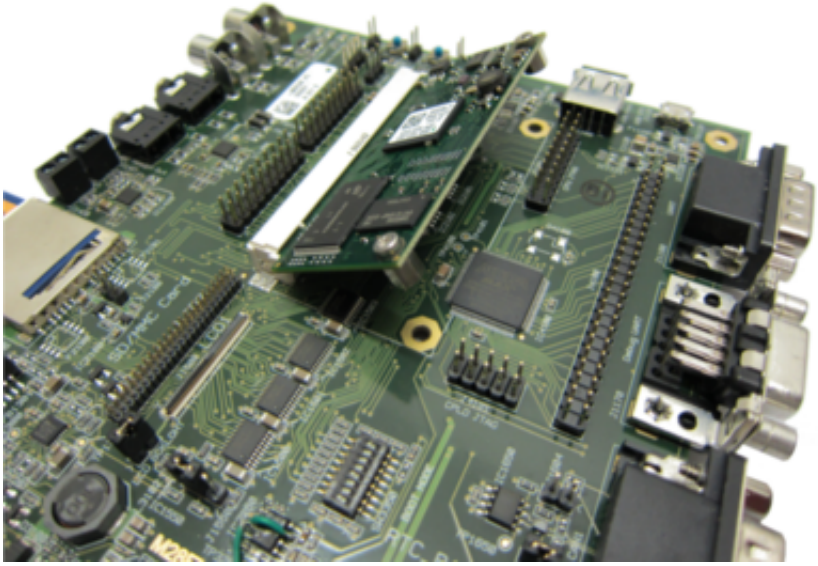
1	2	3	4	5	6	7	8
off	off	on	off	off	off	off	on

2.3 Mount spacer on M28 SoM

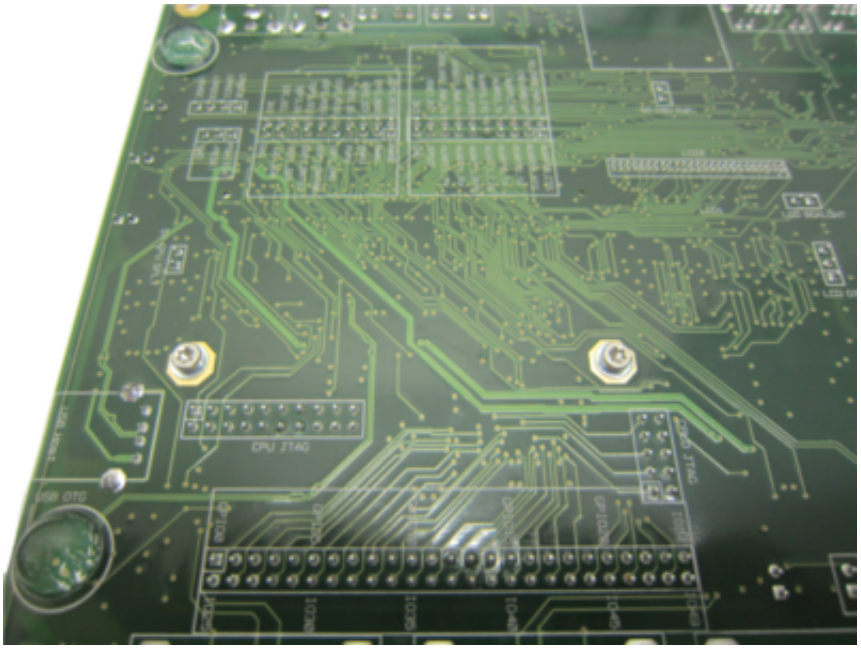
Mount spacers on M28 by fixing the screws properly but not too tight.



2.4 Insert M28 SoM into MXM connector on M28EVK

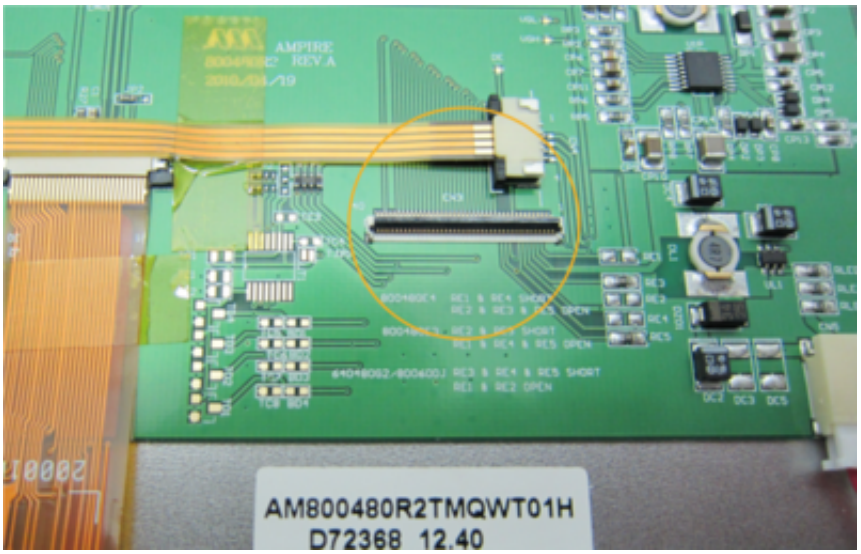


2.5 Mount M28 SoM on M28EVK by fixing the spacers carefully

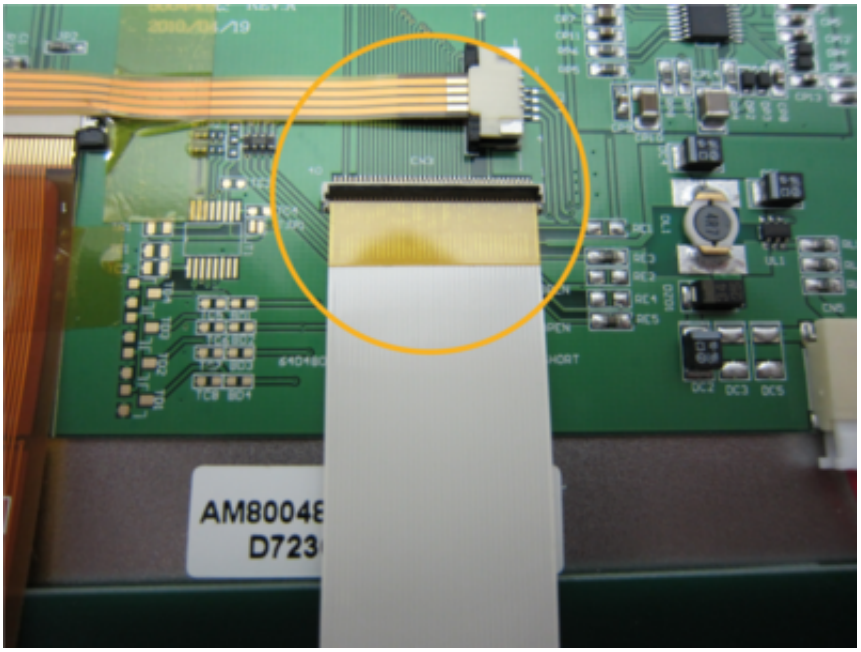


2.6 Mount the MxxDK Displaykit on M28EVK

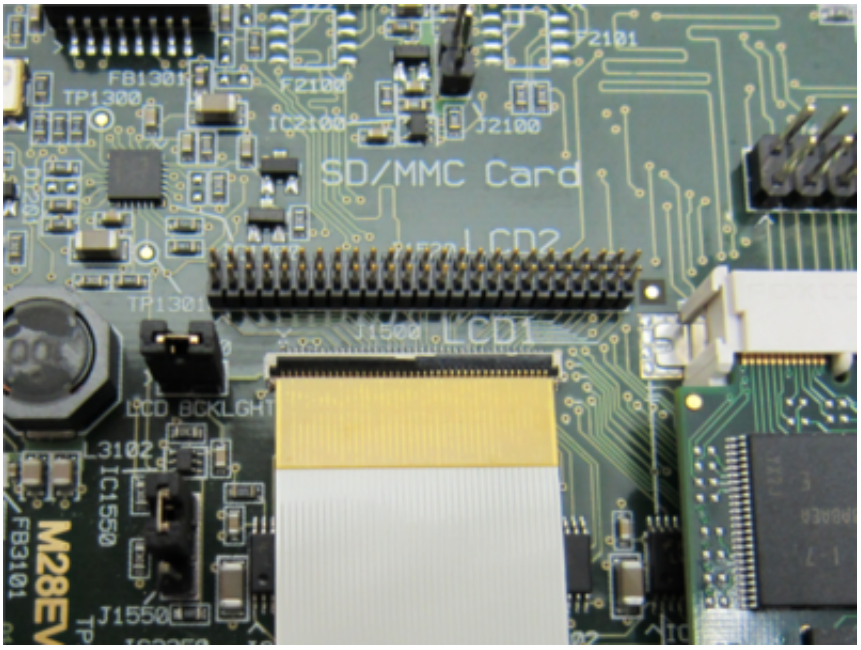
Open the FPC connector on the backside of your MxxDK carefully



Place the FPC cable in the FPC connector on MxxDK, the contact side has to be placed towards the TFT PCB so that the isolation of the cable is on top. Close the FPC connector on the MxxDK carefully.



Place the FPC cable in the FPC connector on M28EVK, the contact side has to be placed towards the TFT PCB so that the isolation of the cable is on top. Close the FPC connector on the M28EVK carefully.



Place the MxxDK Displaykit on M28EVK without twisting or disconnecting the FPC cable. Mount the MxxDK Displaykit on M28EVK by using four screws.

2.7 Plug in your bootable SD-card into M28EVK

2.8 Start your Linux Computer and install ckermit

Boot your Linux Computer and log in. Open a console port and install ckermit (if not already installed):

```
sudo apt-get install ckermit
```

After typing in your sudo password ckermit will be installed.

2.9 Configure kermit

Configure ckermit by creating the file .kermit in your home directory:

cd nano .kermit

Copy into .kermit:

set line /dev/ttyS0 set speed 115200 set carrier-watch off set handshake none set flow-control none robust set file type bin set file name lit set rec pack 1000 set send pack 1000 set window 5

Save the file by typing:

<ctrl> + o

Exit nano by typing:

<ctrl> + x

2.10 Start kermit and connect

Start kermit by typing:

kermit

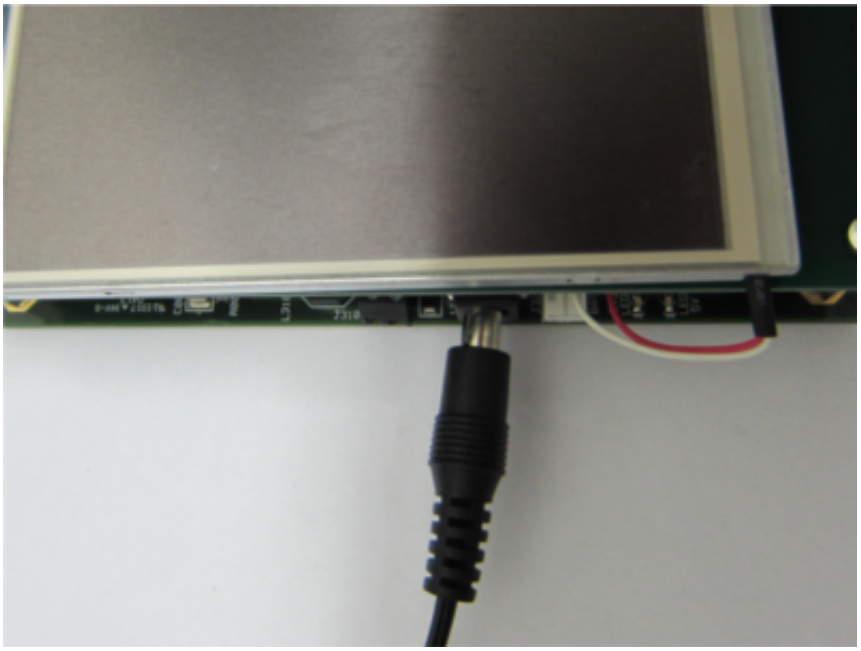
followed by:

c

2.11 Connect the console cable

Connect the console cable to the DUART port J1170 of your M28EVK and the serial port of your Computer as specified above

2.12 Apply power to your kit



M28 will start booting:

```
HTLLCLLC
U-Boot 2014.01 (Feb 15 2014 - 15:59:03)
CPU: Freescale i.MX28 rev1.2 at 454 MHz
BOOT: SSP SD/MMC #0, 3V3
I2C: ready
DRAM: 128 MiB
NAND: 256 MiB
MMC: MXS MMC: 0
Video: 800x480x18
In: serial
Out: serial
Err: serial
Net: FEC0 [PRIME], FEC1
2934624 bytes read in 1059 ms (2.6 MiB/s)
Running bootscript...
## Executing script at 42000000
No FIT subimage unit name
Hit any key to stop autoboot: 0
=>
```

After booting the kernel the login will appear:

```
Starting Bootlog daemon: bootlogd.
Populating dev cache
Configuring network interfaces... done.
Starting rpcbind daemon...done.
hwclock: RTC_RD_TIME: Invalid argument
Sat Feb 15 20:04:00 UTC 2014
INIT: Entering runlevel: 5
Starting system message bus: dbus.
Starting OpenBSD Secure Shell server: sshd
    generating ssh RSA key...
    generating ssh ECDSA key...
    generating ssh DSA key...
done.
Starting Distributed Compiler Daemon: distcc.
Starting syslogd/klogd: done
[ ok ]rting Avahi mDNS/DNS-SD Daemon: avahi-daemon
Starting OProfileUI server
Stopping Bootlog daemon: bootlogd.
ELDK 5.5 m28evk /dev/ttyAMA0
m28evk login:
```

The login on M28 is by default “root” with no password (just press enter on your keyboard):

```
m28evk login: root
root@m28evk:~#
```

After booting the Qt framwork demo will be launched


```
root@m28evk:~# qtdemoE -qws &
```

```
Start hacking, developing your application, setting up benchmarks.......
```

2.13 Shut down your kit

To avoid damage of the SD-card alway shut down your kit properly by typing:

root@m28evk:~# poweroff

Broadcast message from root@m28evk (ttyAMA0) (Sat Feb 15 20:33:41 2014):

The system is going down for system halt NOW!

INIT: Switching to runlevel: 0

INIT: Sending processes the TERM signal

INIT: Sending processes the KILL signal

Stopping OpenBSD Secure Shell server: sshdstopped /usr/sbin/sshd (pid 480)

.

- Stopping Avahi mDNS/DNS-SD Daemon: avahi-daemon

[ok]rofile-server:510): WARNING **: Avahi error: Daemon connection failed

Stopping system message bus: dbus.

Stopping Distributed Compiler Daemon: distcc/etc/rc0.d/K20distcc: stop failed with error code 1

Stopping OProfileUI server

Stopping syslogd/klogd: stopped syslogd (pid 491)

stopped klogd (pid 494)

done

Deconfiguring network interfaces... done.

Sending all processes the TERM signal...

rpcbind: rpcbind terminating on signal. Restart with “rpcbind -w”

Sending all processes the KILL signal...

Unmounting remote filesystems...

Stopping rpcbind daemon...

Deactivating swap...

Unmounting local filesystems...

[1820.943477] EXT4-fs (mmcblk0p3): re-mounted. Opts: (null)

[1823.103543] System halted.

CHAPTER

THREE

BOOTING FROM SD-CARD

3.1 General Assumptions

The SD card is mapped to:

    /dev/mmcblk0

modify the path according to your local setup if necessary

```
$ COMMAND
```

such COMMAND shall be executed on the host system

```
=> COMMAND
```

such COMMAND shell be executed from U-Boot

```
# COMMAND
```

such COMMAND shell be executed from Linux on the target system

the generic MAC adresses:

    C0:E5:4E:XX:XX:XX

will be replaced by the valid MAC-adresses assigned to your M28 SoM.

3.2 Write the supplied image onto the SD card

For Qt/Embedded , unzip the image-file and write the 'sd-qte.img' image:

    $ bzip2 -d core-image-qte-sdk-m28evk.sdcard.bz2

    $ dd if=core-image-qte-sdk-m28evk.sdcard of=/dev/mmcblk0 bs=512

3.3 Insert the SD card into the kit

3.4 Turn the kit on by connecting it to the power supply

3.5 Abort the boot procedure by pressing any key

You will be presented with a U-Boot prompt:

```
'=>'
```

3.6 Reset the U-Boot environment

NOTE: This step is only necessary if the kit is being upgraded from older version of the system software.

Execute the following commands in sequence:

```
=> env default -a
=> saveenv
```

The expected output will be:

```
=> env default -a
## Resetting to default environment
=> saveenv
Saving Environment to NAND...
Erasing redundant NAND...
Erasing at 0x3e0000 – 100% complete.
Writing to redundant NAND... done
```

3.7 Set the MAC adresses

NOTE: This step is only necessary if the MAC address settings were somehow corrupted in the development process. The kit is delivered with pre-configured environment containing the correct MAC address.

First check if the MAC address settings are correct:

```
=> printenv ethaddr
=> printenv eth1addr
```

Validate the output of the above commands against the sticker on the M28 SoM's CPU. The sticker must match the $ethaddr value and the $eth1addr value must be exactly the same as $ethaddr value, but incremented by one in the last octet .

If and only if this is not the case, please restore the correct MAC address using these commands to match the sticker on the CPU:

```
setenv -f ethaddr C0:E5:4E:XX:XX:XX
setenv -f eth1addr C0:E5:4E:XX:XX:XY
saveenv
```

Restart the kit:

```
=> reset
```

CHAPTER

FOUR

BOOTING FROM NAND FLASH

4.1 General Assumptions

The SD card is mapped to:

/dev/mmcblk0

modify the path according to your local setup if necessary

```
$ COMMAND
```

such COMMAND shall be executed on the host system

```
=> COMMAND
```

such COMMAND shell be executed from U-Boot

```
# COMMAND
```

such COMMAND shell be executed from Linux on the target system

4.2 Prerequisites

4.2.1 Make M28EVK bootable from SD card

4.2.2 Download an basic image file for the M28 SoM

4.2.3 Convert basic image into UBI image

Extract the ELDK tarball on the host system

(Note that this operation needs root privileges to allow extraction of the contents of the /dev/ directory.)

```
$ cd path/to/core-image-basic-generic-m28evk.tar.bz2
```

```
$ mkdir eldk
```

```
$ sudo tar -C eldk -xjf core-image-basic-generic-m28evk.tar.bz2
```

Produce the UBIFS image

Use the following command to pack the root filesystem directory into the UBIFS image called 'ubifs.img':

```
$ sudo mkfs.ubifs -r eldk/ -e 126976 -m 2048 -c 1850 -o ubifs.img
```

The -r option specifies the path to root filesystem directory. The -e option specifies the "Default UBI LEB size". The -m option specifies the "Minimum input/output unit size", which is 2048b for the NAND chip used on M28EVK. The -c option specifies maximum LEB count for the particular filesystem image. This value is derived from "Amount of eraseblocks" of the NAND flash and it is slightly lower then that to allow UBI do badblock handling correctly.

The magic numbers above can be retrieved if the board is running from SD card as set up in the section III. paragraph 1. below by running:

```
# mtdinfo /dev/mtd5 -u
```

Each of the option mentioned above has an entry in the list.

Note, it is possible to enable filesystem compression for UBIFS to let larger filesystems to fix into the NAND. The operation with the NAND will in turn be slower due to the compression. This is realized by using the -x and -X options of the mkfs.ubifs tools, please see manpage for more details:

```
$ man mkfs.ubifs
```

Produce NAND UBI image with ELDK

First we need to write an UBI configuration file to drive the creation of the UBI and it's volume containing ELDK. We will produce a volume which supports auto-resizing , which will allow ELDK to spread nicely across most of the free space in the NAND. Write the following lines into 'ubinize.cfg' file:

```
[ubifs]

mode=ubi

image=ubifs.img

vol_id=0

vol_size=192MiB

vol_type=dynamic

vol_name=rootfs
```

Now use the "ubinize" tool to produce the resulting UBI image:

```
$ sudo ubinize -o ubi.img -m 2048 -p 128KiB -s 2048 ubinize.cfg
```

The -m option specifies the "Minimum input/output unit size", the -p option is the "Eraseblock size" and the -s option is the "Sub-page size" . They can again be retrieved by running the following command running on M28EVK board booted as explained in section III 1) below:

```
# mtdinfo /dev/mtd5 -u
```

Make the "ubi.img" file accessible from M28EVK

Please make the "ubi.img" file accessible to M28EVK when booted into ELDK from SD card. Either copy it onto /dev/mmcblk0p3/ device, or make it available via network or USB stick. In the subsequent text, we will assume the

file is available in /ubi.img from the ELDK running from SD card .

4.3 Install system into NAND flash

1. Boot M28EVK from SD card, log in as "root", no password is set.
2. Erase the NAND UBI partition.Erase the UBI partition in NAND that will be rewritten by the newly installed software. The partitions to be erased is the "UBI" partition. The "bootloader" partition will be rewritten separatelly from U-Boot below, since this partition is Read-Only in Linux to avoid any possibility of damaging the board. Run the following command:

   ```
   # flash_erase /dev/mtd5 0 0
   ```

3. Install Linux FIT and UBI images into NAND. To install the ELDK UBI image, run:

   ```
   # ubiformat /dev/mtd5 -f /ubi.img -y
   ```

To install Linux kernel image, run:

```
# mount /dev/mmcblk0p2 /boot/

# ubiattach -p /dev/mtd5

# mount /dev/ubi0_0 /mnt/

# cp /boot/fitImage /mnt/boot/

# umount /mnt/

# ubidetach -p /dev/mtd5

# umount /boot/
```

4. Install bootloader into NAND

Please reboot the board into U-Boot prompt. Load the "u-boot.nand" from the ELDK 5.5 SD card into RAM:

```
=> mmc rescan

=> ext4load mmc 0:2 ${loadaddr} u-boot.nand
```

Load default NAND partitioning layout and wipe the "bootloader" partition. Note that the bootloader partition must be 'scrub'ed to circumvent the regular ECC engine operation on this partition. This is a feature of the i.MX28 BootROM which needs this partitions' ECC data in different format. The update_nand_get_fcb script will retrieve the NAND geometry and fill the necessary variables with correct values:

```
=> mtdparts default

=> run update_nand_get_fcb_size

=> nand scrub -y 0x0 ${filesize}
```

Finally install the new bootloader:

```
=> nand write.raw ${loadaddr} 0x0 ${fcb_sz}

=> setexpr update_off ${loadaddr} + ${update_nand_fcb}
```

```
=> setexpr update_sz ${filesize} - ${update_nand_fcb}

=> nand write ${update_off} ${update_nand_fcb} ${update_sz}
```

5. Adjust the commands to boot from UBI. Adjust the commands by executing the following in U-Boot prompt. This will allow kernel to be loaded from UBI:

   ```
   => setenv nandload 'ubi part UBI ; ubifsmount ubi0:rootfs ; ubifsload ${kernel_addr_r} /boot/fit
   ```

   => saveenv

6. **Reconfigure the board to boot from NAND.** Power down the board. Adjust the bootsettings on the M28EVK accordingly for booting from NAND flash.
7. The board will now boot U-Boot from NAND.

To proceed to booting installed ELDK from NAND, run the following command:

```
=> run nand_nand
```

To permanently configure the board to boot ELDK from NAND, run the following commands:

```
=> setenv bootcmd run nand_nand

=> saveenv

=> saveenv
```

Note the saveenv has to be run twice to update both the environment and the redundant environment.

CHAPTER

FIVE

BUILDING U-BOOT

5.1 General Assumptions

```
$ COMMAND
```

such COMMAND shall be executed on the host system

```
=> COMMAND
```

such COMMAND shell be executed from U-Boot

```
# COMMAND
```

such COMMAND shell be executed from Linux on the target system

This document describes the M28 U-Boot port. This document mostly covers topics related to making the M28 SoM bootable.

Contents

- Compiling U-Boot for a MX28 based board
- Installation of U-Boot for a MX28 based board to SD card
- Installation of U-Boot into NAND flash

1. Compiling U-Boot for M28

NOTE: Starting U-Boot 2014.01, elftosb nor any other external tool is NOT needed anymore!

Compiling the U-Boot for a MX28 board is straightforward and done as compiling U-Boot for any other ARM device. For cross-compiler setup, please refer to ELDK 5.5 documentation.

The source code for U-Boot can be obtained from either the DENX FTP server or GIT server. We will use U-Boot v2014.01 in this text:

```
$ wget ftp://ftp.denx.de/pub/u-boot/u-boot-2014.01.tar.bz2
```

```
$ git clone git://git.denx.de/u-boot.git ; cd u-boot ; git checkout -b m28evk v2014.01
```

First, prepare the environment:

```
$ unset LDFLAGS
```

```
$ export ARCH=arm
```

```
$ export CROSS_COMPILE=arm-linux-gnueabi-
```

Then clean up the source code:

```
$ make mrproper
```

Next, configure U-Boot for a M28 based board:

```
$ make m28evk_config
```

Lastly, compile U-Boot and prepare a "BootStream". The "BootStream" is a special type of file, which the i.MX28 CPU can boot. This is handled by the following command:

```
$ make u-boot.sb
```

HINT: To speed-up the build process, you can add -j<N>, where N is number of compiler instances that'll run in parallel.

The code produces "u-boot.sb" file. This file needs to be augmented with a proper header to allow successful boot from SD or NAND. Adding the header is discussed in the following chapters.

2. Installation of U-Boot for M28 to SD card

To boot a M28EVK board from SD card, set the boot mode DIP switches according to our documentation to SD-card-boot. More details can be found in i.MX28 manual chapter 12.2.1 (Table 12-2), PORT=SSP0, SD/MMC master on SSP0, 3.3V.

An SD card the i.MX28 CPU can use to boot U-Boot must contain a DOS partition table, which in turn carries a partition of special type and which contains a special header. The rest of partitions in the DOS partition table can be used by the user.

To prepare such partition, use your favourite partitioning tool. The partition must have the following parameters:

```
Start sector ..... sector 2048
Partition size ..... at least 1024 kb
Partition type ..... 0x53 (sometimes "OnTrack DM6 Aux3")
```

For example in Linux fdisk, the sequence for a clear card follows. Be sure to run fdisk with the option "-u=sectors" to set units to sectors:

```
o ..................... create a clear partition table
n ..................... create new partition
    p ................. primary partition
    1 ................. first partition
    2048 .............. first sector is 2048
    +1M ............... make the partition 1Mb big
t 1 ................... change first partition ID
    53 ................ change the ID to 0x53 (OnTrack DM6 Aux3)
    <create other partitions>
w ..................... write partition table to disk
```

The partition layout is ready, next the special partition must be filled with proper contents. The contents is generated by running the following command (see chapter 2)):

```
$ ./tools/mxsboot sd u-boot.sb u-boot.sd
```

The resulting file, "u-boot.sd", shall then be written to the partition. In this case, we assume the first partition of the SD card is /dev/mmcblk0p1:

```
$ dd if=u-boot.sd of=/dev/mmcblk0p1
```

Last step is to insert the card into MX28 based board and boot.

NOTE: If the user needs to adjust the start sector, the "mxsboot" tool contains a "-p" switch for that purpose. The "-p" switch takes the sector number as an argument.

3. Installation of U-Boot into NAND flash

To boot a M28 from NAND, set the boot mode DIP switches according to i.MX28 manual chapter 12.2.1 (Table 12-2), PORT=GPMI, NAND 1.8 V.

There are two possibilities when preparing an image writable to NAND flash.

I) The NAND wasn't written at all yet or the BCB is broken

In this case, both BCB (FCB and DBBT) and firmware needs to be written to NAND. To generate NAND image containing all these, there is a tool called "mxsboot" in the "tools/" directory. The tool is invoked on "u-boot.sb" file from chapter 2):

```
$ ./tools/mxsboot nand u-boot.sb u-boot.nand
```

NOTE: The above invokation works for NAND flash with geometry of 2048b per page, 64b OOB data, 128kb erase size. If your chip has a different geometry, please use:

```
-w <size>      change page size (default 2048 b)
-o <size>      change oob size (default 64 b)
-e <size>      change erase size (default 131072 b)
```

The resulting file, "u-boot.nand" can be written directly to NAND from the U-Boot prompt. To simplify the process, the U-Boot default environment contains script "update_nand_full" to update the system.

This script expects a working TFTP server containing the file "u-boot.nand" in it's root directory. This can be changed by adjusting the "update_nand_full_filename" varible.

To update the system, run the following in U-Boot prompt:

```
=> run update_nand_full
```

In case you would only need to update the bootloader in future, see II) below.

II) The NAND flash was already written with a good BCB

This part applies after the part I) above was done at least once.

If part I) above was done correctly already, there is no need to write the FCB and DBBT parts of NAND again. It's possible to upgrade only the bootloader image.

To simplify the process of firmware update, the U-Boot default environment contains script "update_nand_firmware" to update only the firmware, without rewriting FCB and DBBT.

This script expects a working TFTP server containing the file "u-boot.sb" in it's root directory. This can be changed by adjusting the "update_nand_firmware_filename" variable.

To update the firmware, run the following in U-Boot prompt:

```
=> run update_nand_firmware
```

III) Special settings for the update scripts

There is a slight possibility of the user wanting to adjust the STRIDE and COUNT options of the NAND boot. For description of these, see i.MX28 manual section 12.12.1.2 and 12.12.1.3.

The update scripts take this possibility into account. In case the user changes STRIDE by blowing fuses, the user also has to change "update_nand_stride" variable. In case the user changes COUNT by blowing fuses, the user also has to change "update_nand_count" variable for the update scripts to work correctly.

In case the user needs to boot a firmware image bigger than 1Mb, the user has to adjust the "update_nand_firmware_maxsz" variable for the update scripts to work properly.

CHAPTER

SIX

USB FIRMWARE UPLOAD

The USB Firmware Upload tool can be used to upload U-Boot to M28EVK via on-board USB OTG port. The M28EVK has to be connected to a PC using a MicroUSB cable connected to J1120 USB OTG port.

In case of a blank board, the board will fall back to USB Download mode automatically. In case of incorrectly programmed board, it is possible to force USB Download mode by configuring SW3350 to all OFF positions. For further details refer to MX28 datasheet section 12.2.1.

Typical use cases of the USB downloader are

- programming M28 during production
- restoring U-Boot after a faulty non-booting version has been installed during development

Availability

The source code can be found under http://git.denx.de/?p=mxsldr.git;a=summary

Prerequisites

This tool depends on libusb 1.0, make sure to install development version of this library.

In case your kernel is very outdated, meaning kernel older than 3.7.4, 3.4.26 (stable) etc. or if your kernel does not properly handle the MXS BootROM recovery mode, which can be seen in 'dmesg' output by kernel reporting that the device is ignored due to problem with HID descriptor, apply patches (*.diff) onto kernel on your host machine and rebuild the kernel. Then restart the machine with the new kernel. Usage

- Connect the M28 to USB via device cable
- Compile U-Boot and produce u-boot.sb
- Compile this tool:

```
$ make
```

- Load u-boot.sb onto the device:

```
$ sudo ./mxsldr <path to u-boot.sb>
```

CHAPTER

SEVEN

VERIFIED BOOT OF LINUX

7.1 General Assumptions

The SD card is mapped to:

    /dev/mmcblk0

modify the path according to your local setup if necessary

```
$ COMMAND
```

such COMMAND shall be executed on the host system

```
=> COMMAND
```

such COMMAND shell be executed from U-Boot

```
# COMMAND
```

such COMMAND shell be executed from Linux on the target system

1. Write the supplied image onto the SD card

We will use standard OpenSSL installation for generating the key. We will be generating two files, the private signing key and a certificate that contains a public key, which is used by U-Boot for verification of the signed image.

It is of importance to store the new files in a separate directory and even with a specific naming scheme, since this is used later on when signing the kernel image. Yet, the rule of a thumb for the naming is very easy. We will use the following variables throughout the text to point out where the naming scheme matters:

```
key_dir="/work/keys/"
key_name="my_key"
```

The ${key_dir} variable points to a directory containing all the cryptographic material, that is, the private signing key and the certificate. The ${key_name} is trickier in that it's the shared part of the name of both the key and the certificate. The public signing key must thus be called ${key_name}.key and the certificate must be called ${key_name}.crt and both of these files must be located in ${key_dir} directory. Please also note that the .key and .crt extensions of the files are also mandatory. This is the entire rule.

Let us now generate the private signing key. First of all, let's create the directory for storing the cryptographic material:

```
$ mkdir -p "${key_dir}"
```

To generate the private signing key, use the following command:

```
$ openssl genrsa -F4 -out "${key_dir}"/"${key_name}".key 2048
```

You can refer to genrsa(1) for a fine explanation for the options used above, yet we will explain it in terse form. The command generates an RSA private key with public exponent of 65537 , stores it in "${key_dir}"/"${key_name}".key and the size of the key is 2048 bits. Please note that the 2048 bits length of key is the only one supported by U-Boot 2014.01, but a larger key will be supported as of 2014.04 onward.

To generate the certificate containing public key used for verification, apply the following command:

```
$ openssl req -batch -new -x509 -key "${key_dir}"/"${key_name}".key \
        -out "${key_dir}"/"${key_name}".crt
```

Again, you can refer to req(1) for a fine documentation for the options used in the command. The command requests a generation of a new x509 certificate, all in non-interactive mode, with private key for this certificate located at "${key_dir}"/"${key_name}".key . The resulting certificate will be stored at "${key_dir}"/"${key_name}".crt .

We now have both of the necessary files generated, therefore we can continue to adjusting U-Boot.

2. Adjusting U-Boot to support image verification

U-Boot needs certain configuration options to support verification of signed images. Certain things need to be enabled – support for control device tree, support for FIT image and support for FIT image signature verification.

1. Enabling support for control device tree

The U-Boot supports being configured from device tree blob, almost like the Linux kernel does. We will not use the probing functionality here, yet we also use certain nodes in the device tree blob to store the RSA public key used for verification of the Linux kernel images. To enable support for control device tree, add the following line into your boards' configuration file:

```
#define CONFIG_OF_CONTROL
```

We will need to construct such a control device tree, but we will describe this later on.

There are multiple options on the placement of the control device tree, it can be either embedded into U-Boot binary (see CONFIG_OF_EMBED) or attached to the end of the U-Boot binary (see CONFIG_OF_SEPARATE). We will use neither of these and will instead use a third option.

NOTE: The following text is specific to i.MX23 and i.MX28 and U-Boot 2014.01 and newer!

We will adjust the mxsimage.mx28.cfg file (or mxsimage.mx23.cfg respectively) to load the control device tree into OCRAM for us and then we will use the "fdtcontroladdr" to point to this control device tree.

The change to the mxsimage.mx28.cfg is a simple one-line addition. We will be loading the control device tree to address 0x4000 in OCRAM, which is not used by anything and is available on both i.MX23 and i.MX28:

```
diff --git a/arch/arm/cpu/arm926ejs/mxs/mxsimage.mx28.cfg b/arch/arm/cpu/arm926ejs/mxs/mxsimage.mx28
index 676f5c8..16a39ad 100644
--- a/arch/arm/cpu/arm926ejs/mxs/mxsimage.mx28.cfg
+++ b/arch/arm/cpu/arm926ejs/mxs/mxsimage.mx28.cfg
@@ -4,5 +4,6 @@ SECTION 0x0 BOOTABLE
  LOAD IVT 0x8000     0x14
  CALL HAB 0x8000     0x0
  LOAD     0x40000100 OBJTREE/u-boot.bin
+ LOAD     0x4000     /work/u-boot.dtb
  LOAD IVT 0x8000     0x40000100
  CALL HAB 0x8000     0x0
```

Note that we did not yet create the "/work/u-boot.dtb" file. This file will be created further down the road, yet this change now will render your U-Boot source temporarily unbuildable.

Let us do the final adjustment to environment first. We will add a new environment variable into the default environment specifying the location of the control device tree. This is achieved by adding this line to your boards' configuration file:

```
#define CONFIG_EXTRA_ENV_SETTINGS        "fdtcontroladdr=0x4000\0"
```

We are now finished with adjusting the U-Boot to support control device tree.

2. Support for FIT images and their verification

To allow U-Boot to verify a kernel image, the kernel image must be in the FIT image format. The FIT image format is a much more advanced and flexible format compared to regular uImage, let alone zImage. The FIT image format allows for storing multiple kernel images, multiple device tree blobs and even multiple configurations for their combinations in a single image. Each entry in this image can even have different protection options attached to it, be it a hash or a signature. Even the entire FIT image can be signed.

To enable support for this advanced format and support for the verification of signatures in it, add the following configuration options to your boards' configuration file:

```
#define CONFIG_FIT

#define CONFIG_FIT_SIGNATURE
#define CONFIG_RSA
```

To make the output more verbose when using FIT images, you can also add the following, which is rather helpful during the debugging phase:

```
#define CONFIG_FIT_VERBOSE
```

These adjustments now give our U-Boot support for FIT images and their verification.

Creating the control device tree for U-Boot

The control device tree for U-Boot will be really simple in our case. The file will only contain a node for storing the RSA public key and thus will look as below. The file will be placed in /work/u-boot.dts to correlate with the location we added into mxsimage in section 2.1) above:

```
/dts-v1/;

/ {
        model = "Keys";
        compatible = "denx,m28evk";

        signature {
                key-dev {
                        required = "conf";
                        algo = "sha1,rsa2048";
                        key-name-hint = "my_key";
                };
        };
};
```

Notice the /signature/key-dev/key-name-hint node, which must contain the name of the key used for signing. This value must match the ${key_name} variable we defined in section 1). The value is also the one used by the mkimage tool down below to find the correct section to augment with the public key for image verification.

Rebuilding the U-Boot tools with FIT signing support

Finally, let us rebuild the U-Boot tools so we will have an mkimage binary which also supports FIT image signing. This is a precursory measure in case your distribution did not contain it. To rebuild just the U-Boot tools, use the following command in the U-Boot source tree:

```
$ make tools
```

We now have U-Boot source augmented with support for both usage of control device tree and verification of FIT images. But, we will not rebuild the source code just yet. We first need to extend the control device tree with the RSA public key for verification of the FIT image.

3. Building the FIT image with Linux kernel

Producing a FIT image with Linux kernel and a device tree blob that goes with the Linux kernel is rather straightforward. We will first need the Linux kernel zImage and the device tree blob that goes with it. We will assume we are now located in the Linux kernels' source directory and that the zImage is located in arch/arm/boot/zImage and that the device tree blob for this board is in arch/arm/boot/dts/imx28-m28evk.dtb .

We will need to write the FIT image device tree descriptor. The file for this particular configuration will look as such and is one of the simplest possible example of FIT image configs. We will save the content below into the /work/kernel_fit.its file:

```
/dts-v1/;

/ {
        description = "Simple image with single Linux kernel and FDT blob";
        #address-cells = <1>;

        images {
                kernel@1 {
                        description = "Linux kernel";
                        data = /incbin/("./arch/arm/boot/zImage");
                        type = "kernel";
                        arch = "arm";
                        os = "linux";
                        compression = "none";
                        load = <0x40008000>;
                        entry = <0x40008000>;
                        hash@1 {
                                algo = "sha1";
                        };
                };

                fdt@1 {
                        description = "Flattened Device Tree blob";
                        data = /incbin/("./arch/arm/boot/imx28-m28evk.dtb");
                        type = "flat_dt";
                        arch = "arm";
                        compression = "none";
                        hash@1 {
                                algo = "sha1";
                        };
                };
        };

        configurations {
                default = "conf@1";
                conf@1 {
                        description = "Boot Linux kernel with FDT blob";
                        kernel = "kernel@1";
                        fdt = "fdt@1";
                        signature@1 {
                                algo = "sha1,rsa2048";
                                key-name-hint = "my_key";
                        };
                };
```

```
        };
};
```

You can notice the /images/kernel@1/data and /images/fdt@1/data nodes, which point to the location of the payloads in the filesystem, in this case the kernel and device tree blob. You can also notice the /images/kernel@1/load and /images/kernel@1/entry nodes, which point to the kernel's load address and the entry point. These are specific to the i.MX23 and i.MX28 CPUs. You should also notice the /images/kernel@1/hash@1 and /images/fdt@1/hash@1 nodes, which describe the type of hash used to protect the files in these sections – in this case it is SHA-1 hashing algorithm used to protect both the kernel and the device tree blob.

Finally, there is an property which is important for signing of the FIT image. It is the /configurations/conf@1/signature@1/key-name-hint property. This one must contain the name of the key that will be used to sign this image. This must match the ${key_name} variable we defined in section 1) above.

4. Signing all things

In this section, we will finally produce a signed FIT image and also build the control device tree for U-Boot with the RSA public key for verification.

1.Building the control device tree for U-Boot

We will start with compiling the control device tree for U-Boot, since the mkimage tool which places the RSA public key into the control device tree operates on compiled device tree blob. To compile the U-Boot's control device tree into a device tree blob, use the following command:

```
$ dtc -p 0x1000 /work/u-boot.dts -O dtb -o /work/u-boot.dtb
```

Please notice the -p option, which reserves 4096 bytes of data in the device tree blob for the RSA public key, which will be added shortly. The resulting U-Boot's device tree blob is stored in /work/u-boot.dtb .

2. Producing the signed FIT image

To produce the signed FIT image, all we have to do is to be in the kernel source tree and execute the following commands:

```
$ ln -s /work/kernel_fit.its .
$ mkimage -D "-I dts -O dtb -p 2000" -f kernel_fit.its /work/fitImage
```

The symbolic link is necessary, otherwise mkimage won't pick the correct base paths as specified in the kernel_fit.its and will thus fail producing the FIT image.

Explanation of the second command is also appropriate, especially for the '-D "-I dts -O dtb -p 2000"' portion. This portion reserves 2000 bytes in the FIT image for storing the RSA signature later on. The '-f' option points to the FIT image source file we produced in section 3) and the /work/fitImage is the resulting FIT image. Note that this FIT image is not signed yet.

3. Signing the kernel image and augmenting the U-Boot's control device tree

Both the signing of a kernel image and augmentation of the U-Boot's control device tree happen as a one command for the first time. Note that the U-Boot's control device tree needs to be augmented with the RSA public key only once. There will be a section on signing the FIT images without touching the U-Boot's control device tree further down the road, the section 6).

Without further ado, we will now sign the FIT image and augment the U-Boot's control device tree with this command:

```
$ mkimage -D "-I dts -O dtb -p 2000" -F -k "${key_dir}" \
    -K /work/u-boot.dtb -r /work/fitImage

You can verify that the U-Boot's control device tree was augmented with four new nodes by using this

$ dtc -I dtb -O dts /work/u-boot.dtb
```

You will notice four new device tree nodes in /signature/key-my_key/ , named 'rsa,r-squared', 'rsa,modulus', 'rsa,n0-inverse' and 'rsa-num-bits' . Now that we have a signed FIT image stored in the kernel source directory and a U-Boot's control device tree with the public key implanted in it, we can proceed to building a new U-Boot binary and testing the result.

5. Assembling and testing the result

First of all, we need to rebuild U-Boot. This is now following the regular procedure of building U-Boot for DENX M28EVK. Once the U-Boot is built, install it on the board again via documented procedure.

With U-Boot in place, we can test if the board can verify the fitImage file. Boot the board and load the file. We will assume the FIT image is loaded at ${loadaddr} address ; loadaddr=0x42000000 . We can now use the 'bootm' command in U-Boot to test the verification as such:

```
=> bootm start ${loadaddr}
```

This command will not boot the kernel image. Instead, it will verify if the fitImage hashes for all it's parts are correct and it will also check if the signature verification passes. We can check if the verification passed by observing the following line in the output:

```
Verifying Hash Integrity ... sha1,rsa2048:my_key+ sha1,rsa2048:my_key+ OK
```

Note that the 'my_key+' is significant in two ways. First of all, the 'my_key' again matches the ${key_name} variable we defined in section 1). Moreover, the plus sign right past it indicates that the verification was successful. On the contrary, a minus sign would signify the verification was unsuccessful.

An unsuccessful verification can easily be simulated by signing the fitImage with a different key and attempting the verification again. An unsuccessful verification output looks as such, but please note that this output is with the CONFIG_FIT_VERBOSE enabled, so it's extra verbose:

```
Verifying Hash Integrity ... sha1,rsa2048:my_keyrsa_verify_with_keynode: RSA failed to verify: -22
rsa_verify_with_keynode: RSA failed to verify: -22
- Failed to verify required signature 'key-my_key'
Bad Data Hash
ERROR: can't get kernel image!
```

6. Signing subsequent kernel images

This final chapter is but a quick addendum to cover signing of subsequent kernel image without the need to replace the U-Boot's control device tree every time a new kernel is signed. The command to produce only a new signed fitImage is as follows and shall be executed from the kernel's source tree:

```
$ mkimage -D "-I dts -O dtb -p 2000" -k "${key_dir}" -f kernel_fit.its -r fitImage
```

CHAPTER

EIGHT

POWER CONSUMPTION

8.1 General Comments

The following information is meant for information only, to give developers an idea in what range of power consumption M28 operates It should not be used as a reliable basis for the determination of supply currents for the M28 SoM or the M28 EVK. These values may change due to the load of the system that is caused by

- the interfaces used on M28
- the basedboard that M28 is populated on
- the software that runs on M28
- other influencing factors that are not highlighted here

8.2 Test Setup

The basic test setup included a M28 System on Module applied on the M28EVK. The kit operated optionally the MxxDK Displaykit.

The current measurements were performed at four points in the system:

- input power feed (9.2V) giving the overall (M28SoM + M28EVK) power
- M28SoM +5V
- M28SoM +3.3V (from on-board DC/DC)
- LCD backlight power

8.3 Operational Modes

Three operational modes were tested:

- M28 running in idle mode at U-Boot prompt
- M28 running in idle mode at Linux prompt
- M28 running Qt demo, with some activity on the screen
- M28 producing network traffic by the command “ping -f”. As we were not able to detect any measurable difference in power consumption compared to idle state this step was ommited, the results will not be shown in this document.

8.4 Total system power consumption

U-Boot idle, no LCD backlight:

```
Current: 200mA
Voltage: 9.2V
Power consumption: 1.84 W
```

U-Boot idle, with LCD backlight:

```
Current: 550mA
Voltage: 9.2V
Power consumption: 5.06 W
```

Linux idle, no LCD backlight:

```
Current: 270mA
Voltage: 9.2V
Power consumption: 2.48 W
```

Linux idle, with LCD backlight:

```
Current: 610mA
Voltage: 9.2V
Power consumption: 5.61 W
```

Linux, Qt demo, with backlight:

```
Current: 605mA
Voltage: 9.2V
Power consumption: 5.57 W
```

During the demo initialization/load, a peak of the power consumption to 625mA, 5.75 W was detected. Also a slightly lower power consumption with demo loaded compared to idle Linux system) may be attributed to the lighter image on the LCD. The LCD consumes most power when it shows black color, exactly when the Linux idle and login prompt is the only text on it. The Qt demo has overall gray background.

With the network active (ping -f on one interface), no LCD backlight, no difference with Linux idle state was observed.

Note:

During the system power consumption measurements the module was powered from 5V power. If the measurements were taken with the module powered from +3.3V the power consumption might have been lower a little bit due to efficiency of on-module DC-DC converter (5V to 3.3V).

8.5 Module power consumption, +5V power

U-Boot idle:

```
Current: 137mA
Voltage: 5.0V
Power consumption: 0.685 W
```

U-Boot/bootdelay countdown active:

```
Current: 150mA
Voltage: 5.0V
Power consumption: 0.750 W
```

Linux idle:

```
Current: 150mA
Voltage: 5.0V
Power consumption: 0.750 W
```

197mA @ 5V = 0.985 W (200..230 mA during boot, 1 .. 1.15 W)

Linux Qt demo:

```
Current: 150mA
Voltage: 5.0V
Power consumption: 0.750 W
```

200 .. 210 mA @ 5V = 1.0 .. 1.05 W

8.6 Module power consumption, +3.3V power

U-Boot idle:

```
Current: 147mA
Voltage: 3.3V
Power consumption: 0.485 W
```

U-Boot/bootdelay countdown active:

```
Current: 150mA
Voltage: 3.3V
Power consumption: 0.545 W
```

Linux idle:

```
Current: 230mA
Voltage: 3.3V
Power consumption: 0.760 W
```

230mA @ 3.3V = 0.76 W

Linux Qt demo:

```
Current: 235mA
Voltage: 3.3V
Power consumption: 0.775 W
```

8.7 LCD Backlight

For the LCD backlight the following values were measured:

```
Current: 570mA
Voltage: 5V
Power consumption: 2.85 W
```