Embedded systems, the road to Linux

Exploring boot methods

Angelo Dureghello
angelo.dureghello@timesys.com

Feb 1, 2020

Why talking again of boot methods

The main idea of this talk session is to provide useful information related to the early stages of the boot process on some common Linux-based embedded scenarios.

While Linux kernel boot-related aspects may be more standard and abstracted from the hardware, previous boot stages are generally board/CPU-specific and linked with the hardware design.

Commonly Used Acronyms and Abbreviations

GPMC	Generic Purpose Memory Controller
OCRAM	On Chip RAM, same as SRAM
RBL	ROM BootLoader
SoC	System on Chip
SPL	Secondary Program Loader
SRAM	Static RAM
TPL	Tertiary Program Loader
XIP	eXecution In Place
SLC	Single Level Cell
MLC	Multi Level Cell
TLC	Triple Level Cell

Roadmap

- generic boot concepts
- Linux-oriented boot
- ROM bootloader
- bootstrap pins
- boot from different memory types
- boot types
- useful U-Boot commands
- first boot time optimization
- boot troubleshooting
- boot modes - SoC comparison

Generic boot concepts

- focusing on SoC's
- they generally have multiple boot options
- ROM bootloader (RBL) comes into play
- where to load the code ? internal SRAM comes into play
- execution chances are
  - execution in place (XIP) or ...
  - execution of code shadowed to internal SRAM

Generic boot concepts

- binary read from RBL, if serially read, should have an header with size
- RBL may miss (NOR flash boot, CS @ address 0, XIP)
- XIP requires random access to the program memory
- RBL is also known as First Program Loader
- first executed code is the Secondary Program Loader (SPL)
- SPL is also known as 1st stage bootloader
- SPL may be followed by a TPL (Tertiary Program Loader)

Linux-oriented boot

- we are landing on >= 32bit CPU / SoC's
- kernel loading requires SDRAM/DDR initialized
- appropriate amount of RAM to run the kernel is needed
- a non volatile memory to store kernel/rootfs is needed
- as an additional development boot mode as SD/USB boot is often mandatory

Linux-oriented boot

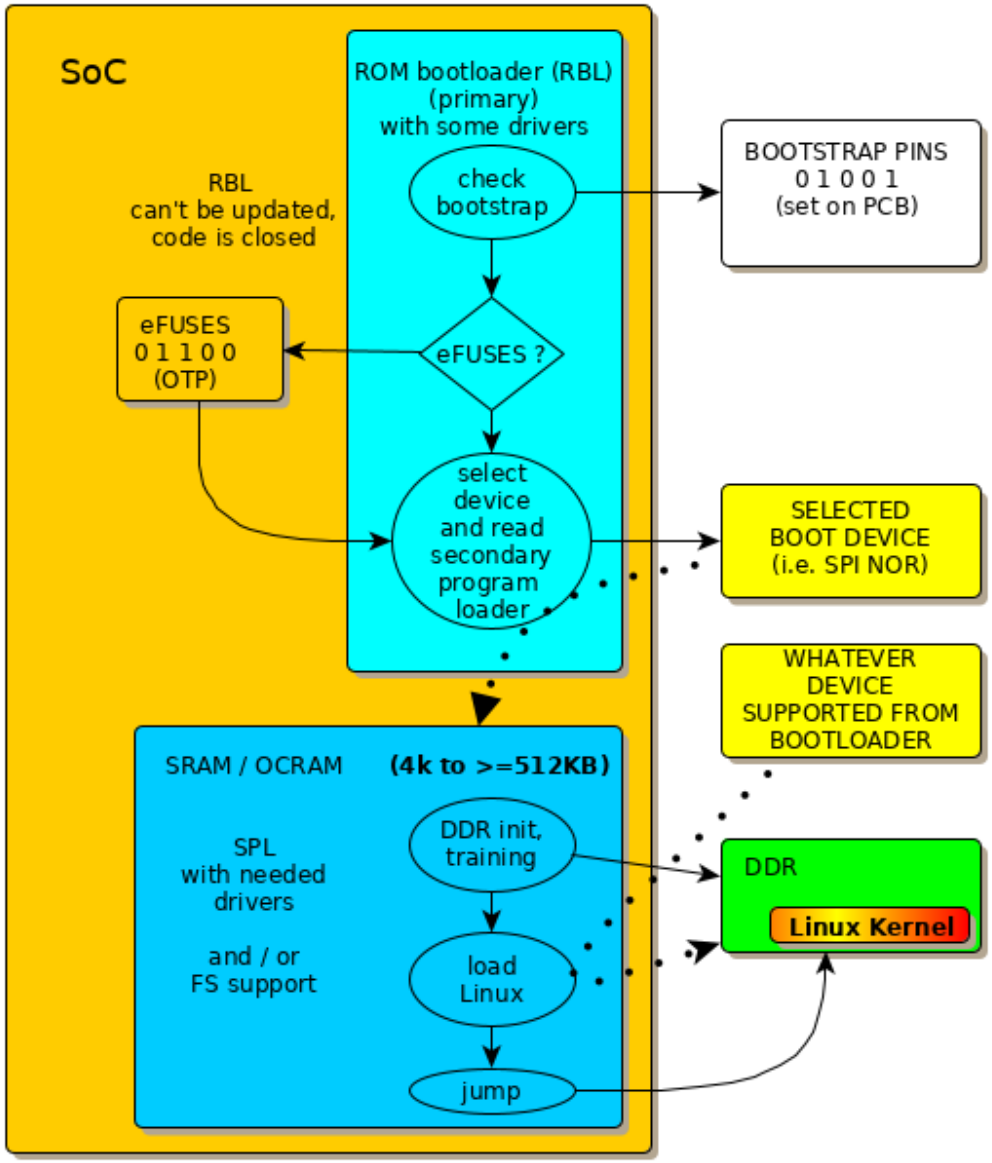
- U-boot as an intermediate step is a common choice
  - is similar to Linux kernel as source code organization and development process, a lot of drivers
  - allows DDR setup/training
  - kernel parameters selection, boot mode selection
  - offers a hush shell, scripting, and lot of commands, filesystems access and boot from fs
  - allows to load separate devicetree, initramfs into ram
  - allows system updates
  - allows debugging prior to kernel boot
  - allows pre-load of any firmware on separate cores
  - popular, opensource, maintained, regular relases

Linux-oriented boot

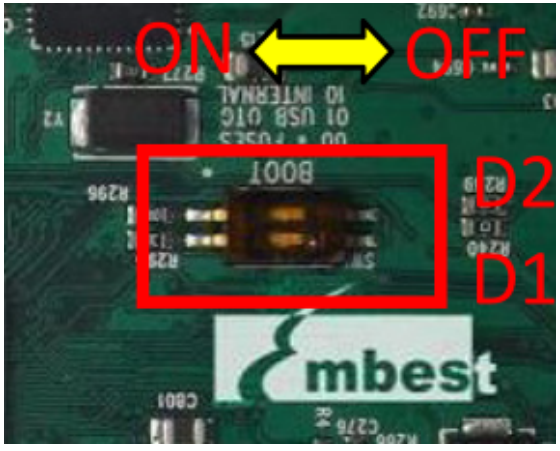
- so we go from simple systems, booting from NOR XIP, 1st stage bootloader may be a simple standalone binary
- to recent SoC's, and ARM TrustZone, there are more bootlaoder blobs involved and packed together, with headers and secure boot signatures, etc
- a bootloader may be avoided (merging absolutely necessary parts to the beginning of Linux kernel) i.e. for boot time optimizations.

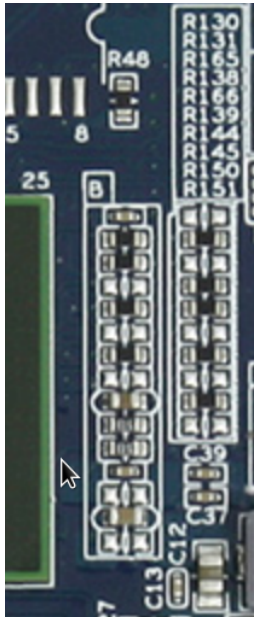
ROM bootloader

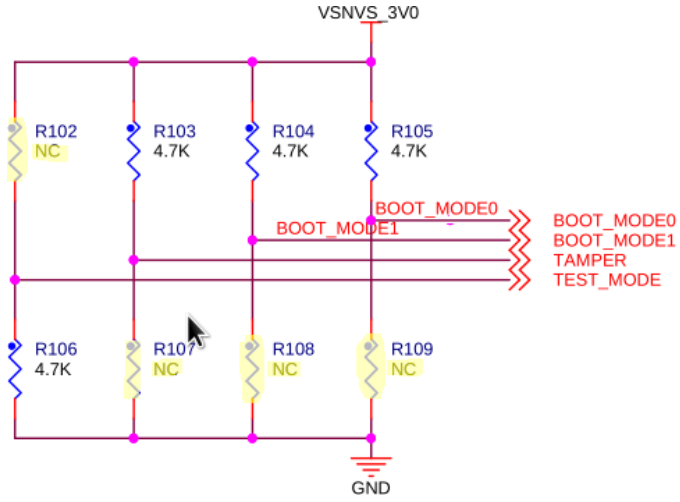
A quite common scenario on 32bit systems



Bootstrap pins







- to select boot mode, some switches can be used
- or some fixed pull-up/down resistors
- resistors are generally 4k7 or 10k, kohm range
- SoC reads pin values at reset, then pins can be used of other purposes

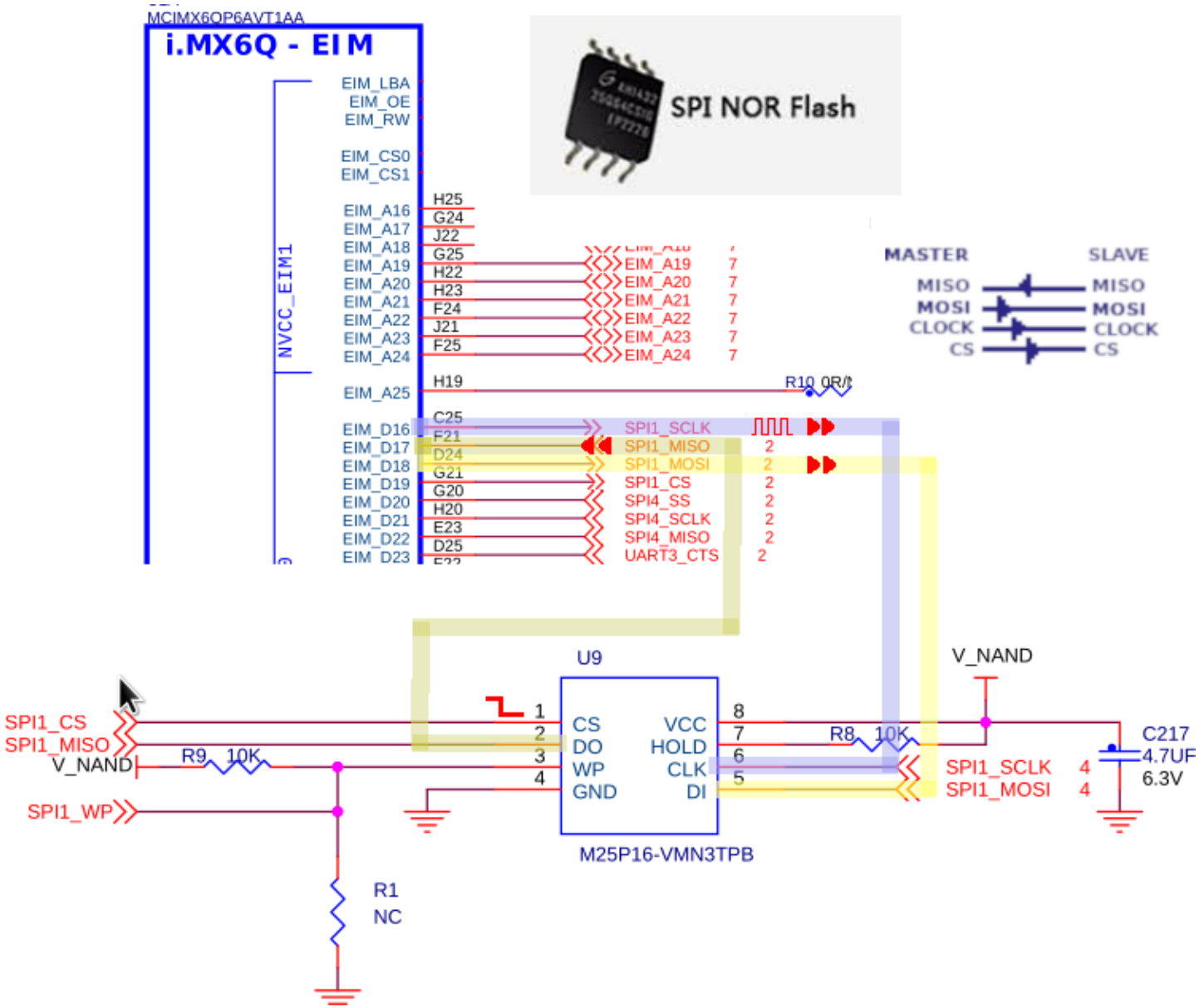
Boot from different memory types

- system architect should define the proper hardware
- is a fast boot time needed ?
- or, is more important keep the cost low ?

Let's take a tour over the most common boot memories

Boot from different memory types

SPI NOR



Boot from different memory types

SPI NOR

- quite common option for booting U-boot
- synchronous, full-duplex
- simple SPI is easy to wire and control (4 wires)
- generally used to provide isolation of U-boot
- it can be simple (MISO/MOSI), dual, quad or octal
- SOIC8 package is simple to de-solder and re-program off-board

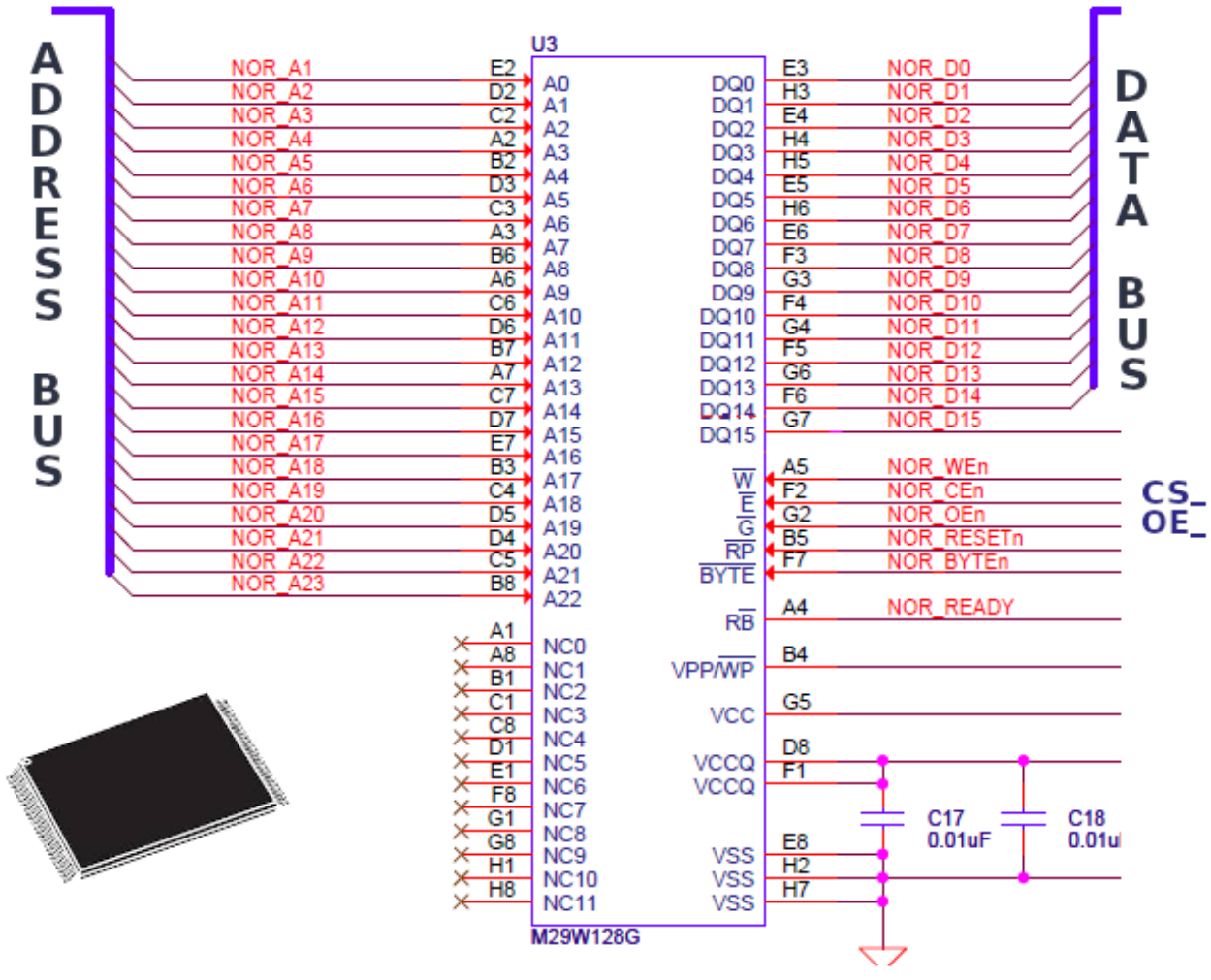
Boot from different memory types

SPI NOR

- transfer rate is type dependent (simple, dual, quad, octal)
- transfer rate is clock dependent, but not only
- transfer rate depends also from time interval between each data packet
- standard SPI, SOIC8 density up to 256Mbit (32MB), clock up to 166Mhz
- standard SPI is not random access, but pages can be read and memory-mapped
- quad and octal can be used for XIP, through specific commands, where chunks of 4/8/16 bytes can be read and cached

Boot from different memory types

Parallel NOR flash, an old new friend



Boot from different memory types

Parallel NOR flash, an old new friend

- SoC must have proper parallel bus and CS for boot
- from reset, execution in place (XIP)
- similarly to an old CPU, code read/fetched and executed, z80, 8051 etc
- a CS is associated to a certain address range, CPU will boot from there
- concept of wait states, to be configured on CPU side
- address and data buses, behave as a normal static RAM chip
- random access
- not only XIP, no one forbid to copy code in internal SRAM
- up to 2Gb (512MB)

Boot from different memory types

Parallel NOR flash - random read

- address is set on address lines
- CS goes low (active low)
- OE goes low (active low)
- after configured wait state (delay) data is available on data lines
- CS and OE comes back high (inactive)
- whole cycle may take 70 to 120ns
- throughput is 1/100 ns * (16/8) = 20 MB/s, not very impressive
- smaller density are faster

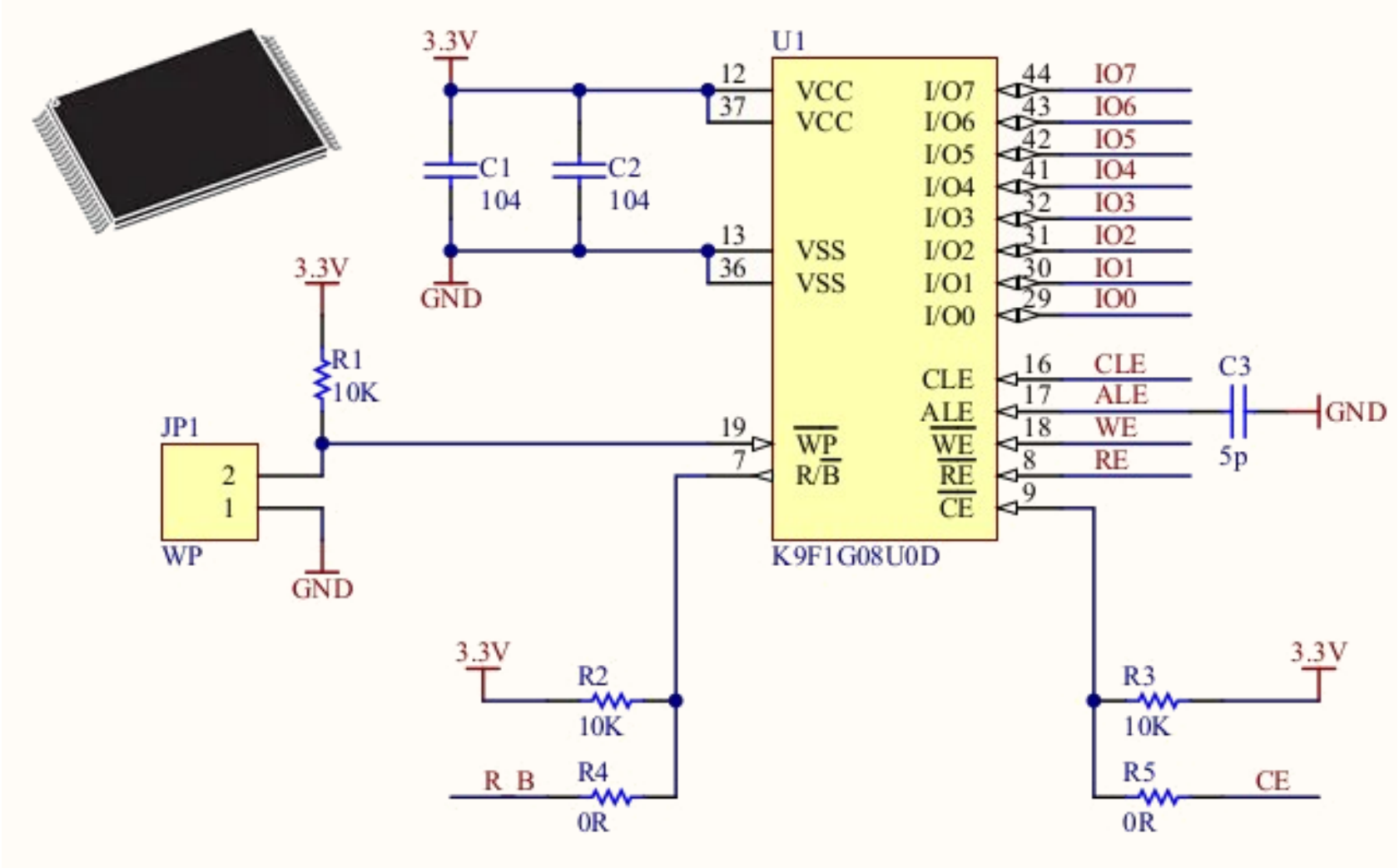
Boot from different memory types

Parallel NOR flash - improving speed

- page mode (can be configured from U-boot)
- must map the NOR memory region as cachable
- address is set on address lines
- CS and OE are set low
- after first read delay (about 100 ns), first word is available on data lines
- CS and OE remain low, CPU increments only the address
- * next word is available in 10 / 20 ns
- after reading 4, 8 or 16 words in this manner, OE and CS return high
- speed is 1/(100 ns + 7 * 15 ns) * 8 * (16/8) = 78 MB/s

Boot from different memory types

NAND



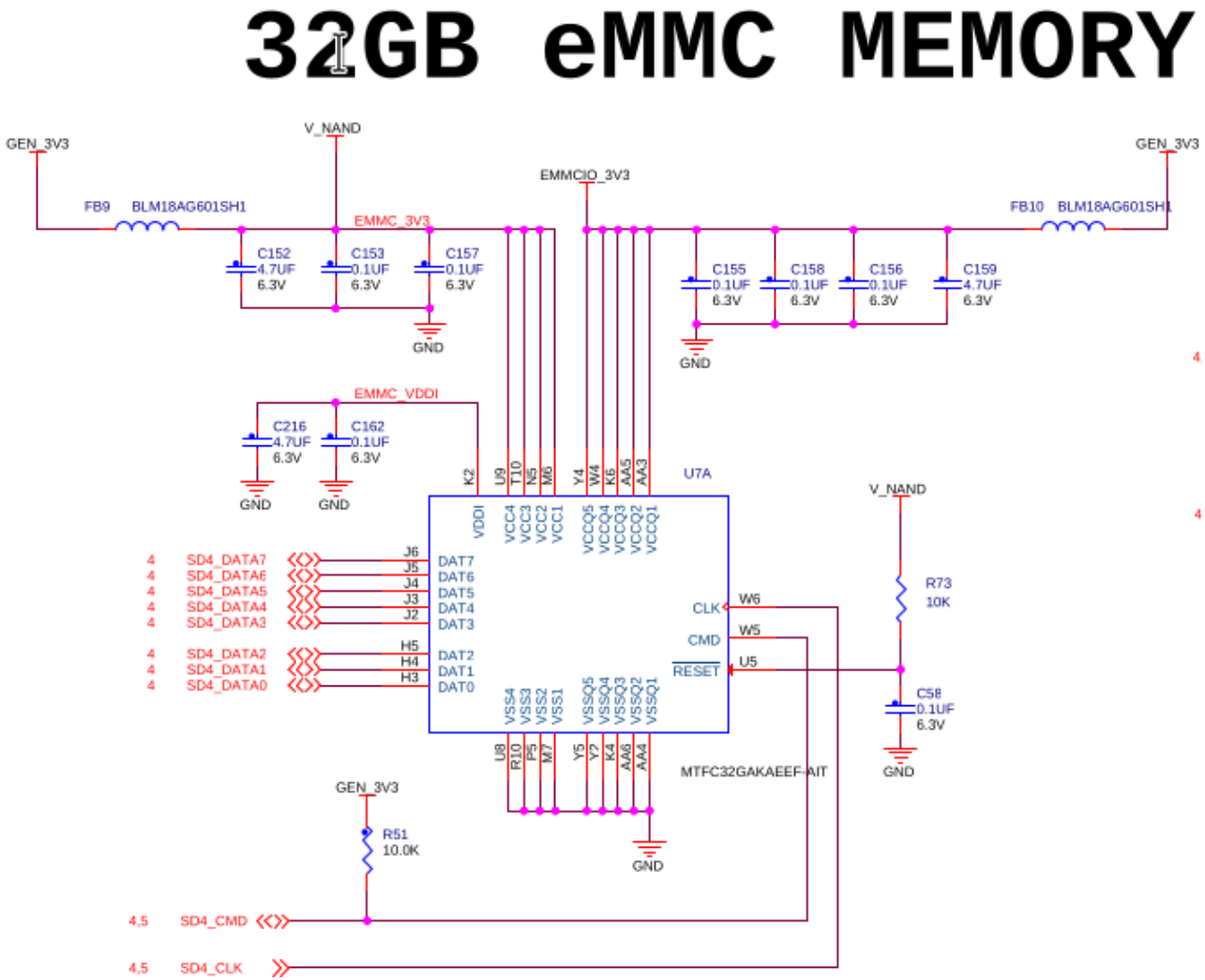
Boot from different memory types

NAND

- NAND flash devices use a multiplexed address and data bus
- asynchronous, CLE and ALE for command / address / data cycle
- NAND chips are block based (versus truly random access)
- RBL caches blocks to static RAM
- faster on write/erase compared to NOR
- SLC, MLC, eMLC, TLC, QLC, 3D NAND etc
- boot sectors generally guaranteed to be better than the rest of the chip
- but known to be error prone (needs sw ECC mechanism)
- density ranges between 1Gb to 16Gb (2GB)
- read performance more or less around 50 MB/s, write speed faster than NOR

Boot from different memory types

NAND



Boot from different memory types

eMMC

- NAND based with a complex hw front-end inside
- protocol similar to SDIO, reliable, low power
- vast majority is 8 bit data bus
- no need to care about error correction
- usage similar to SD, so engineers like eMMC
- fewer lines than parallel NOR or NAND, attractive, so leave free pins for other purposes
- MMC 5.1 up to 400MB/sec with DDR and 8 data lines
- but, $>=$ 50ms of initialization time
- SoC can boot from eMMC shadowing to RAM, block based, no direct execution possbile

Boot from different memory types

SD

- generally used for development
- as per eMMC, has internal circuit, a startup time is needed, up to 250ms may be needed
- in many SoC's, RBL can boot initial fw blob directly from SD

Boot from different memory types

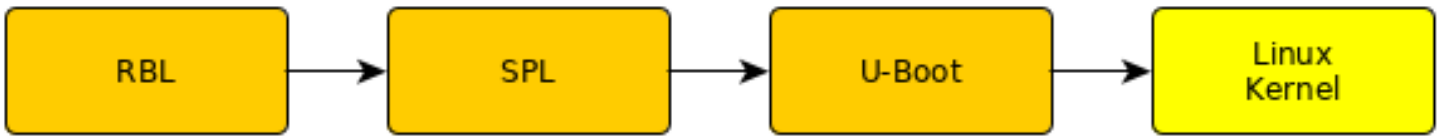
Other technologies

- OneNAND
- UFS chips / cards, up to 3 GB/s, 50ms init time
- Semper Flash (Octal Interface)
- ...

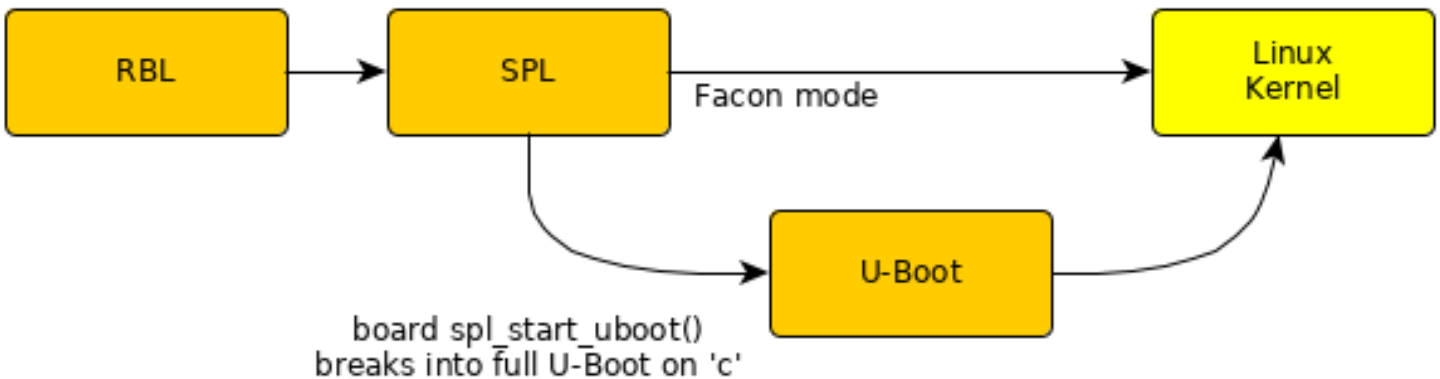
Boot types

U-Boot can be loaded into SoC internal SRAM and executed, or executed in place

U-Boot can't fit into internal SoC SRAM

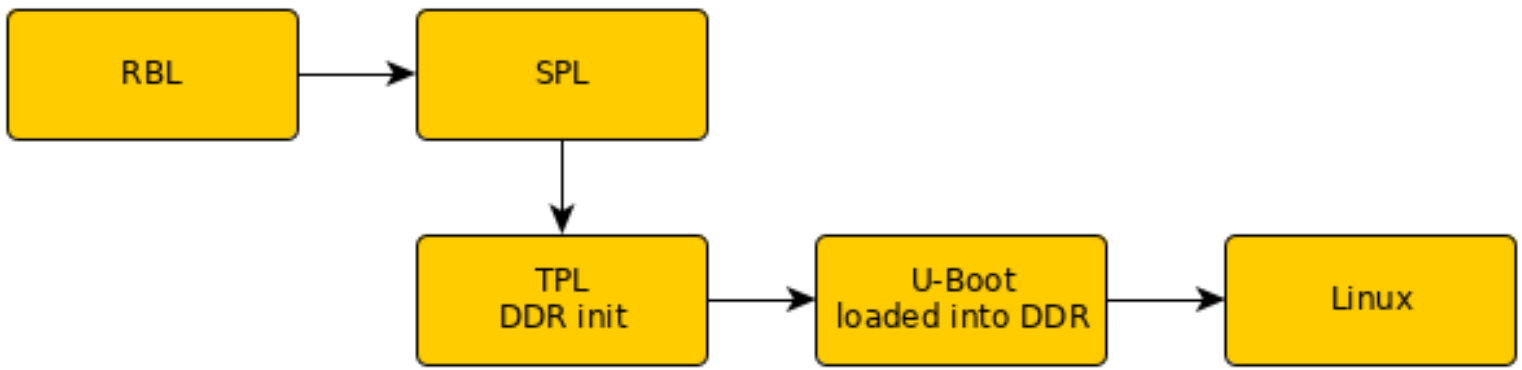


One reason for SPL can be to have a small bootlader that loads kernel directly (Falcon mode)



Boot types

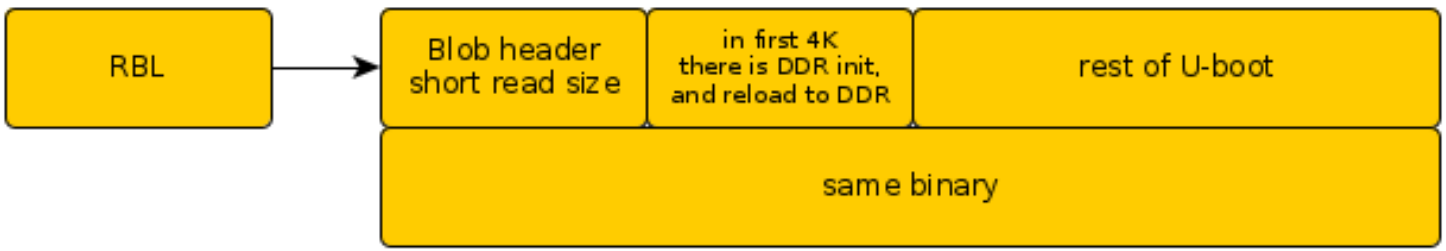
There may be cases where there are SPL read-size limitations,
as i.e. OneNAND that exposes 1KB only for the boot,
SPL cannot still initialize DDR or perform traninig, a TPL is needed



Note that sometimes SPL and TPL are used inverted (TPL -> SPL)
see #ifdef CONFIG_TPL_BUILD ... ad doc/README.TPL

Boot types

SPI NOR, U-boot does not fit into SRAM, but don't want SPL, reading single U-boot



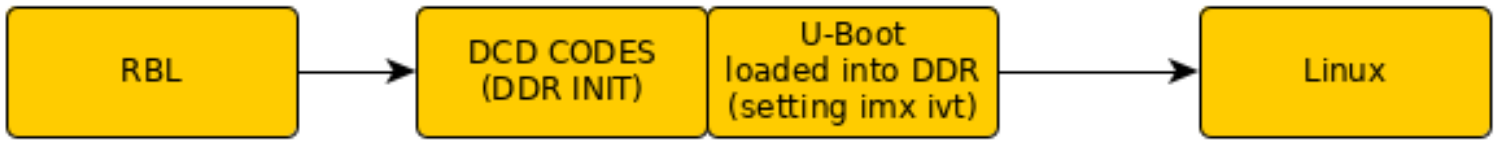
This can be done calling a ddr_init() from start.S,
(keeping ddr_init() in initial binary part, linker script),
and adjusting binary size in the header by proper tool.

As an example see arch/m68k/cpu/mcf5445x/start.S

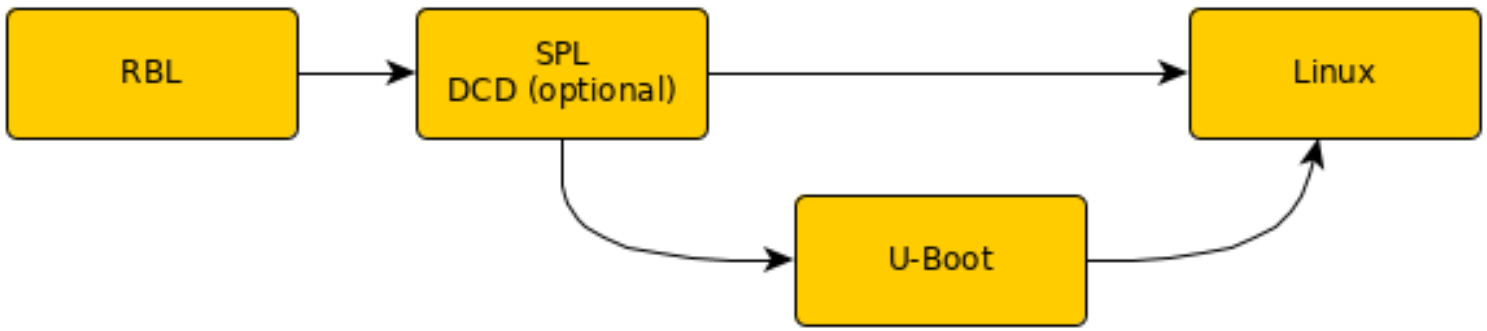
Boot types

Special case, IMX

For OCRAM smaller than U-boot size, U-Boot can be loaded directly into DDR



But may not be reliable, so U-Boot SPL for DDR3 calibration on edge thermal conditions can be chosed



Direct kernel boot, possible but not versatile

Boot types

ARM TrustZone - secure world

- some recent SoC starts in "secure world"
- ARM specification, but different implementations
- there may be a "system manager" additional core
- several boot stages (bl1, bl2, bl31) and bl33
- you may find propertary blobs (imx8qm, etc)

`https://github.com/ARM-software/arm-trusted-firmware`
`https://github.com/ARM-software/arm-trusted-firmware/tree/master/docs/plat`

Some useful U-Boot commands

General stuff

u-boot fdt info

```
U-Boot > dm tree
```

mmc load, fs listing, writing

```
U-Boot > mmc dev 0:1
U-Boot > ls mmc 0:1
U-Boot > load mmc 0:2 ${loadaddr} /boot/uImage
4856776 bytes read in 352 ms (13.2 MiB/s)
U-Boot > ls mmc 0:2 /boot
<DIR>       4096 .
<DIR>       4096 ..
<SYM>         63 uImage
           47312 imx6q-hello.dtb
U-Boot > size mmc 0:2 boot/uImage
U-Boot > echo ${filesize}
4a1bc8
```

commands for memory display (all memory access commands as md mw mm cmp are built-in)

```
U-Boot > md.b 0x20C4000 4
020c4000: ff 10 01 04
U-Boot > md.l 0x20C4000 1
020c4000: 040110ff
```

writing otp eFuses (program once)

```
U-Boot > help fuse
U-Boot > fuse read 0 5
Reading bank 0:
Word 0x00000005: 18000030
```

Some useful U-Boot commands

BMODE, reboot from device (imx)

CONFIG_CMD_BMODE=y in configs/yourboard_config

```
U-Boot > help bmode
```

acting on imx SRC_GPR9, SRC_GPR10 (0x20d8040, 0x20d8044)

```
=> bmode
bmode - mmc0|mmc1|normal|usb|sata|ecspi1:0|ecspi1:1|ecspi1:2|ecspi1:3|esdhc1|esdhc2|esdhc3  [noreset]
```

reboot from serial downloader (usb)

```
=> bmode usb
resetting ...
```

if bmode command is not available, boot mode switches not accessible, how to reboot from usb serial downloader ?
BOOTCFG values can be applied in reserved register SRC_GPR9

```
mw.l 0x20d8040 0xd860; md.l 0x20d8040 1; mw.l 0x20d8044 0x10000000; reset
```

Some useful U-Boot commands

GPIO, enable some stage

CONFIG_CMD_GPIO=y in configs/yourboard_config

```
U-Boot > help gpio
```

controlling gpios

```
U-Boot > gpio clear 86
gpio: pin 86 (gpio 86) value is 0
U-Boot > gpio set 86
gpio: pin 86 (gpio 86) value is 1
U-Boot > gpio toggle 86
gpio: pin 86 (gpio 86) value is 0
U-Boot > gpio input 86
gpio: pin 86 (gpio 86) value is 1
U-Boot > gpio status
Bank GPIO1_:
GPIO1_5: input: 1 [x] i2c_scl2
GPIO1_27: output: 0 [x] rgmii_reset_nitrogen6x

Bank GPIO2_:
GPIO2_1: input: 1 [x] menu
GPIO2_2: input: 1 [x] back
GPIO2_3: input: 1 [x] search
GPIO2_4: input: 1 [x] home
...
```

but how to determine gpio number ?
Each SoC constructor has a proper encoding. I.e., for imx6, 32 bit ports, (GPIO_PORT - 1) * 32 + gpio number

Some useful U-Boot commands

Reprogram SPI NOR Flash

CONFIG_CMD_SF=y in configs/yourboard_config

```
U-Boot > sf help
```

SPI NOR detection

```
U-Boot > sf probe
SF: Detected w25q32bv with page size 256 Bytes, erase size 64 KiB, total 4 MiB
```

Not detectetd ? Check CONFIG_SPI_*CONSTRUCTOR* to your include/configs/your_board.h

SPL U-Boot update (on imx6), load file through same console, block aligned erase, write (providing file by a ymodem terminal)

```
U-Boot > loady 0x12000000
## Ready for binary (ymodem) download to 0x12000000 at 115200 bps...
CCCCCC
File: /home/angelo/dev-timesys/u-boot-fslc/SPL
Size: 52224 bytes.
Starting file transfert ...
Transfer start, protocol y-modem, crc is on, block size is 1024
Sync received
53248/52224
Closing session ...
U-Boot > sf erase 0x0 0x10000
SF: 65536 bytes @ 0x0 Erased: OK
U-Boot > sf write 0x12000000 0x400 0xfc00
SF: 64512 bytes @ 0x400 Written: OK
```

Error ? Check block alignment and block size.

Some useful U-Boot commands

Boot commands

old, monolithic

```
U-Boot > go ${loadaddr}
```

bootm takes 3 arguments, kernel uImage + initramfs + devicetree

```
U-Boot > bootm ${loadaddr} ${loadaddr_ramfs}
## Booting kernel from Legacy Image at 40001000 ...
   Image Name:   mainline kernel
   Created:      2020-01-10  12:52:01 UTC
   Image Type:   M68K Linux Kernel Image (uncompressed)
```

kernel zImage + fdt, no initramfs

```
U-Boot > bootz ${loadaddr} - ${fdt_addr_r}
```

FIT images, group of blobs previously declared in a devicetree-like file (its)
mkimage -f fitimage.its fitImage

```
U-Boot > load mmc 0:1 ${loadaddr} fitimage
U-Boot > iminfo
U-Boot > bootm
```

First boot time optimization

Some optimization tips

- a good result starts from the hardware choice, so
- boot time experts should be consulted before new hardware or software is introduced to the project
- two memory types worth considering: NOR and NAND
- check U-boot and boot device drivers, be sure to have the maximum speed setup enabled
- disable console, U-boot countdown
- check clock signals by oscilloscope, to be sure the running speed is correct
- measure boot time from reset through gpios and oscilloscope
- then comes kernel boot time optimization, but this is another story

First boot

Troubleshooting, custom board/prototype, no boot, nothing on console

- check proper and stable power supply of involved devices (multimeter, oscilloscope)
- check proper reset signals and reset timings sequence for all involved devices (oscilloscope)
- check CPU clock (oscilloscope), often possible
- check if any activity on first boot device (clock and data lines)
- have a look at the schematic diagram, check proper connections (swapped / wrong connections)
- if there is activity, and data seems loaded from RBL, check header and boot device data content (reprogram)
- start to add some gpio toggling in the code to debug

Boot modes - SoC comparison

Table: Boot options

	bootstrap	SRAM	Boot options
ARM926EJ-S	SYSCFG	128K	NOR/XIP, NAND, SPI NOR, eMMC/SD, USB
Allwinner H3	UBOOT pin/FEL	32K	SPI NOR, NAND, eMMC/SD, USB
am335x	SYSBOOT[11:0]	63K	NOR/XIP, NAND(I2C)(1), MMC/SD, SPI, USB, EMAC
imx6Solo	BOOT_MODE[1:0] BOOT_CFG1[7:4]	128K	NOR/XIP, NAND, MMC/eMMC/SD, SATA
imx6DQ	BOOT_MODE[1:0] BOOT_CFG1[7:0]	256K	NOR/XIP, NAND, MMC/eMMC/SD, SATA, i2c, SPI
Zynq-7000	MIO[8:2]	256K	NOR/XIP, NAND, Quad-SPI/XIP, SD, JTAG
TI DRA7xx	SYSBOOT[5:0]	512K	NOR/XIP, NAND, SPI/QSPI, eMMC/SD, SATA, USB
imx8QM	BOOT_MODE[5:0]	256K(2)	eMMC/SD, NAND, FlexSPI, SATA, USB
BCM2837	GPIO BANK1/2	512K(3)	SD, NAND, SPI, USB

(1) ROM bootloader tries to read NAND geometry from i2c eeprom
(2) TCM
(3) L2 cache

QUESTIONS

THANK YOU