

U-boot startup sequence

CCMA, ITRI

houcheng

Agenda

- U-boot introduction
- Multistage boot
- U-boot on Arndale octa board
- Secure boot

U-boot

- Features
 - initialize: Bootstrap CPU, serial console, boot string`
 - hardware: detection, test and flash EEPROM
 - loading kernel image from: network, SSD, EEPROM
 - support secure boot and loading trust software
 - loading hypervisor (TYPE1/ TYPE2)
- Supports
 - hardware: different architecture, CPU, boards and devices
 - software: different file format, flash based file system and fat file system

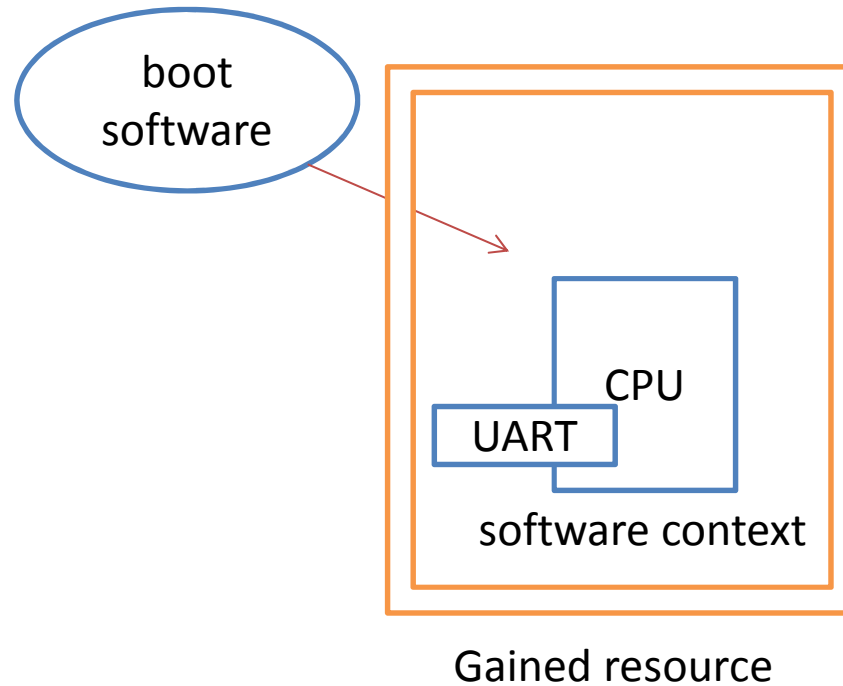
U-boot history

- 1999: PPCBoot start from MPC860-fads board, PPC CPU
- 2000: network support
- 2002: merge with armboot (support 106 boards PPC/ ARM)
- 2002: x86 support
- 2003: MIP32 MIPS64 support
- 2003: NIOS support
- today: u-boot 1.1.2 , support > 216 boards

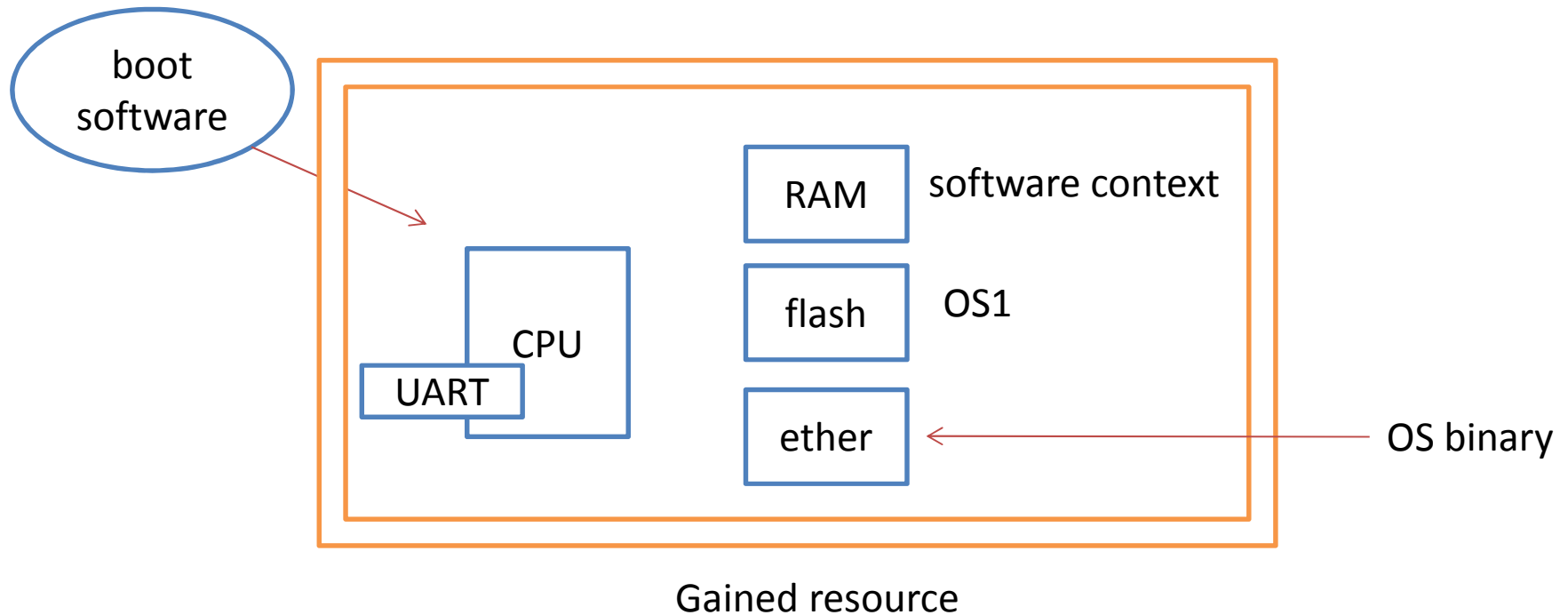
Boot code initialize these

- initialize components on CPU
 - disable: icache, dcache, TLB, MMU
 - setting UART
- initialize software context
 - Interrupt vector of all mode
 - Stack pointer point to iRAM or RAM
- initialize components on board
 - System clock controller
 - DDR memory controller
- Relocate codes
 - move code to proper location (iRAM or RAM) and jump to it

Initial state, few resource with limitation



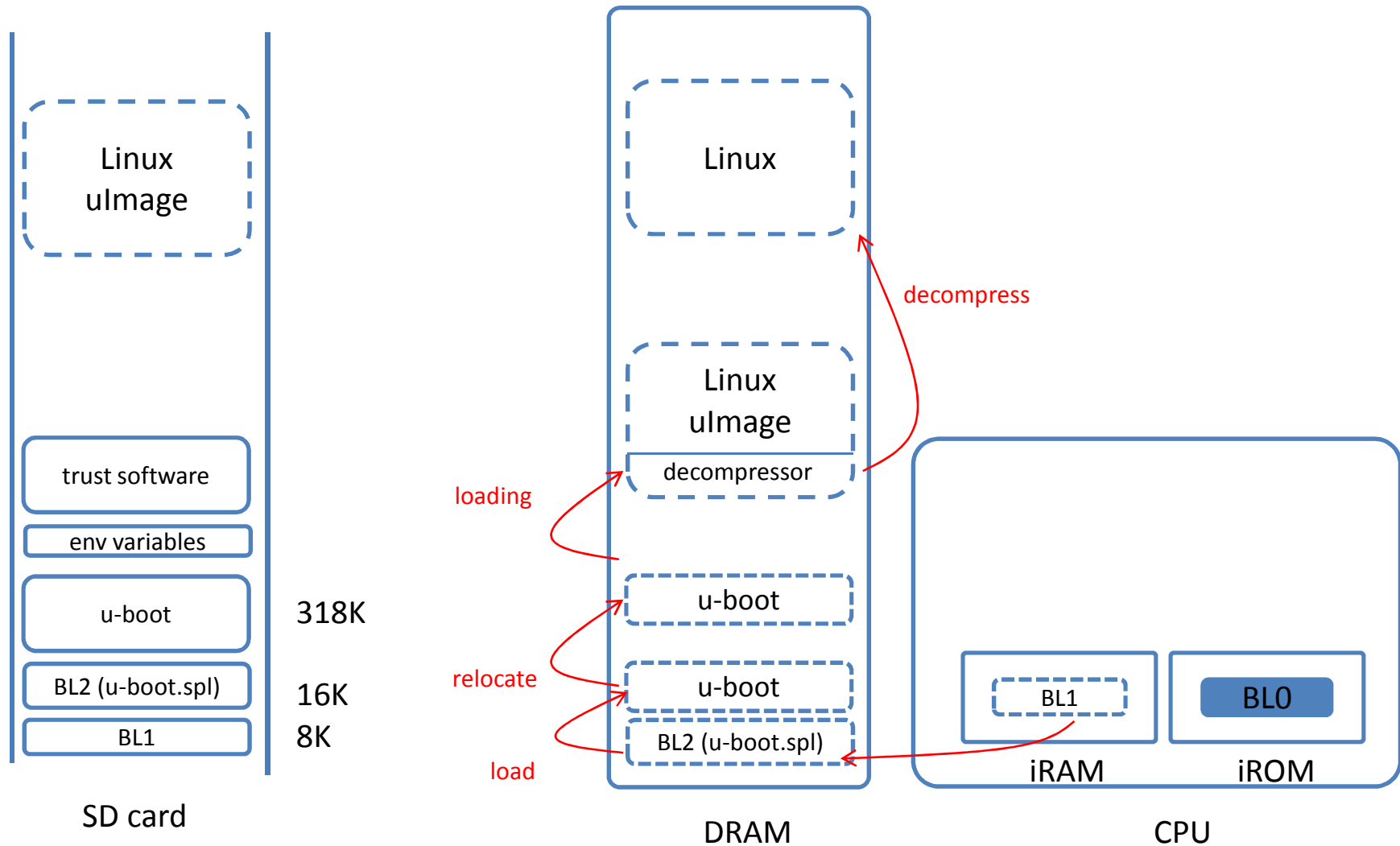
After initialize devices/ controller on board



Multi stages then U-boot

- Multistage boot loader
 - BL0, BL1, BL2, u-boot
- Why multi stages?
 - CPU comes up with bare resource: CPU, iRAM and iROM only
 - incremental initialize more resources on every stage
 - CPU components (caches, TLB, MMU)
 - execution environment: stack, exception vector, interrupt controller
 - devices: sys clock, memory, serial IO, network, RTC,
 - every stage runs on different text base
 - BL0: runs on iROM (vendor fused)
 - BL1: runs on iRAM (vendor provided)
 - BL2: runs on memory (u-boot SPL version)
 - U-boot boot loader: runs on high memory (u-boot)
 - ulmage: runs on memory (a compressed kernel with de-compressor)

Boot loaders sequence



U-boot for Arndale Octa board

- Board configuration defines
 - include/configs/arndale_octa.h
 - arch/arm/include/asm/arch-exynos/movi_partition.h
- CPU dependent code
 - arch/arm/cpu/armv7/*.c;*.S
 - arch/arm/cpu/armv7/exynos/*.c
 - arch/arm/lib/*.c
- Board dependent code
 - board/samsung/smdk5420/*.c; *.S
- Board independent code
 - common(cmd, flash, env, stdio, usb, ...), disk (partition),
 - drivers, fs, net, lib (CRC, SHA1,...),
 - SPL: ld script to generate BL2 u-boot

armv7/start.S


```
.globl _start
_start: b    reset
        ldr pc, _undefined_instruction
        ldr pc, _software_interrupt
        ldr pc, _prefetch_abort
        ldr pc, _data_abort
        ldr pc, _not_used
        ldr pc, _irq
        ldr pc, _fiq
```

→ reset:

- 1) bl save_boot_params
- 2) set CPU to SVC32 mode
- 3) set exception table VBAR to _start
- 4) bl cpu_init_cp15
- 5) bl cpu_init_cr
- 6) setup stack to memory and call C function: _board_init_f

Exception vector table on _start

exception vector



```
.globl _start
_start: b    reset
        ldr pc, _undefined_instruction
        ldr pc, _software_interrupt
        ldr pc, _prefetch_abort
        ldr pc, _data_abort
        ldr pc, _not_used
        ldr pc, _irq
        ldr pc, _fiq
```

handler table

```
_undefined_instruction: .word undefined_instruction
_software_interrupt:    .word software_interrupt
_prefetch_abort:       .word prefetch_abort
_data_abort:           .word data_abort
_not_used:             .word not_used
_irq:                  .word irq
_fiq:                  .word fiq
_pad:                  .word 0x12345678 /* now 16*4=64 */
```

handler function:

```
undefined_instruction:
    get_bad_stack
    bad_save_user_regs
    bl  do_undefined_instruction
```

cpu_init_cp15 and cpu_init_cr

- `cpu_init_cp15`
 - invalidate TLB, MMU, icache and dcache
- `cpu_init_cr`
 - call smdk5420 board's `lowlevel_init.S`
 - use iRAM as stack
 - read board boot switch and store flag in iRAM
 - relocate code if needed
 - check every reset switches on board is not set
 - read boot string stored on board's EEPROM
 - initialize system clock (`system_clock_init`)
 - initialize memory controller (`mem_ctrl_init`)

board_init_f (f:flash)

- Fill board information into global_data object
 - board rate, clock rate, FDT data (hard coded)
 - monitor size, ram size, ram base, memory bank size
- Reserve high memory to store
 - hardware buffer: TLB, frame buffer, monitor
 - software buffer: IRQ stack, heap, program stack and another copy of global data object
- Call init_sequence function array
 - serial_init, console_init, timer_init, disp_banner,
 - env_init, dram_init (detect memory size)
- relocate code then call call board_init_r with param:
 - pointer of stack
 - pointer of copied global_data object
 - pointer of heap

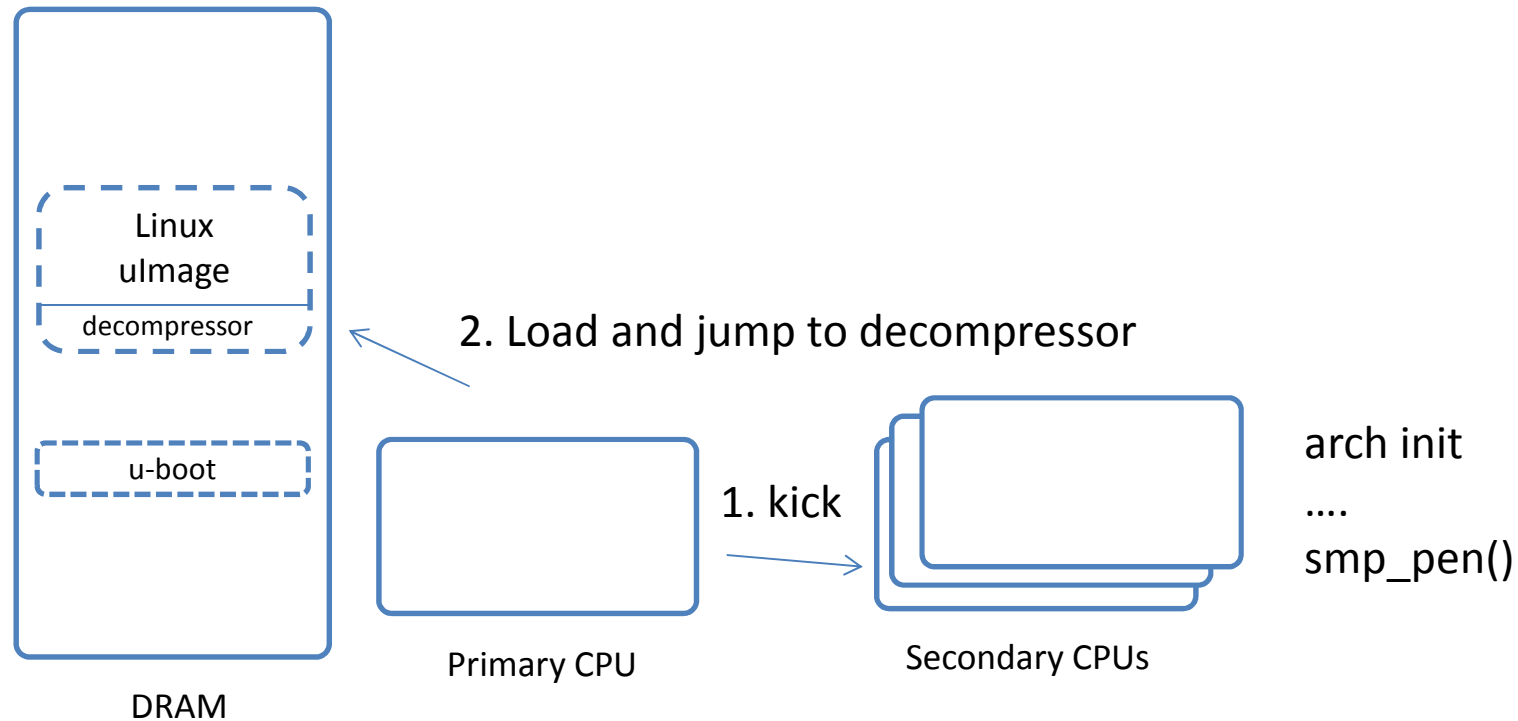
board_init_r (r: ram)

- Initialize devices and related libraries
 - serial console, flash, MMC, ether net
- setup interrupt handlers and enable interrupt
- POST
- Command line main loop

ARM trusted boot

- Trust boot
 - SOC vendor fuse PUK in BL0 and stored in ROM
 - SOC vendor hold the private key that can sign BL1/ BL2
 - Trust boot sequence
 - CPU runs in secure mode, from BL0
 - BL0 initialize bare hardware and verify BL1's signature with PUK
 - BL1 or BL2 setup necessary protection in secure world, loading trust software in secure world and perform world switch
- Some trust boot options
 - Use OTP (One Time Programming) in SOC to store PUK
 - If PUK > OTP memory, store hash of Puk in OTP and store Puk from flash
 - PUK may changed on next stage

Secondary CPU enter hyper mode



Primary CPU in kernel

- a) decompress done and jump
- b) board init complete
- c) SMP code kick Secondary CPU

Secondary CPU in kernel

- wake up by CPU0
- Install hyper stubs by setting HTBAR
- enter SVC32
- run ordinary kernel SMP init