Operating System Design and Implementation

Booting process

Charles Tsao

Outline

- Why boot-loader/OS booting are necessary?
- How boot-loader work?
- How does OS boot?
- A case study of PC and Linux 0.11
- Advanced topics

Why boot-loader/OS booting are necessary?

- Bootloader vs. OS booting
  - Bootloader: The program (boot-loader) loads OS into memory
  - OS booting: initialize and prepare for providing services

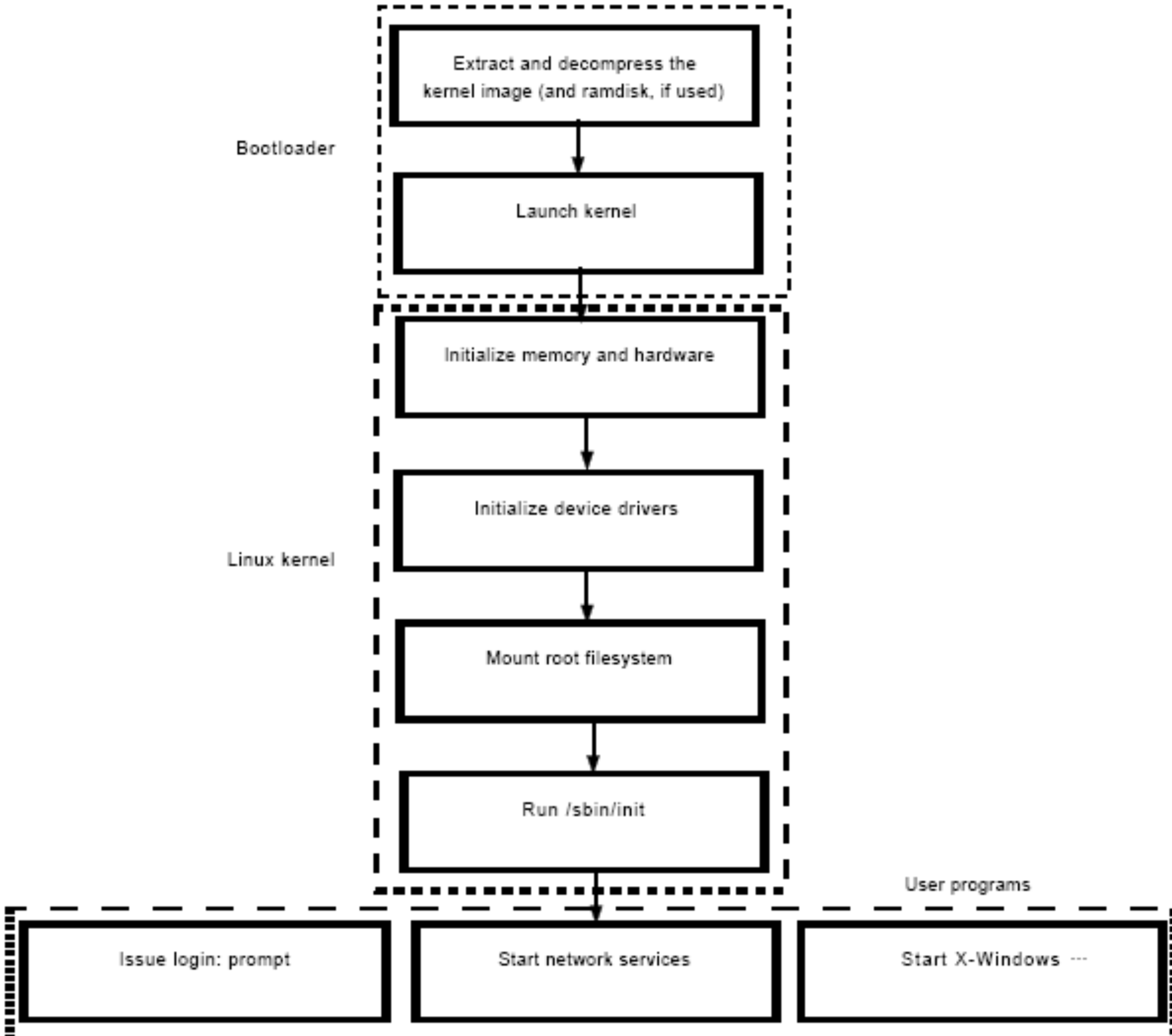
Why a boot-loader is necessary?

- Where is operating system?
- Where is the boot-loader?
  - Permanent storage
  - Addressable
  - ROM, PROM, EPROM, EEPROM, NOR Flash, NV-RAM, …
- Can we store OS on permanent and addressable memory?
  - Why and why not?

Why OS booting is necessary?

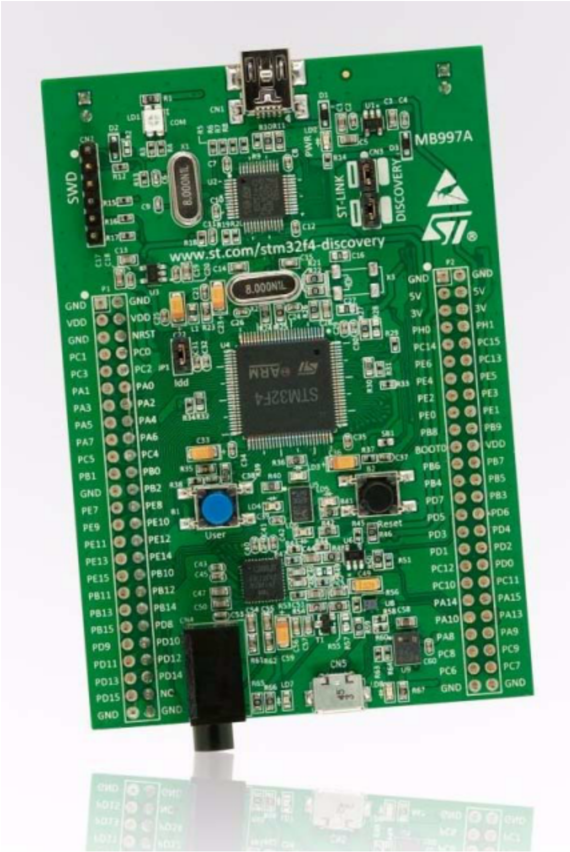
- Load OS/modules itself to correct memory address
- Initialize hardware
- Initialize kernel structure
- Load necessary default process
- …

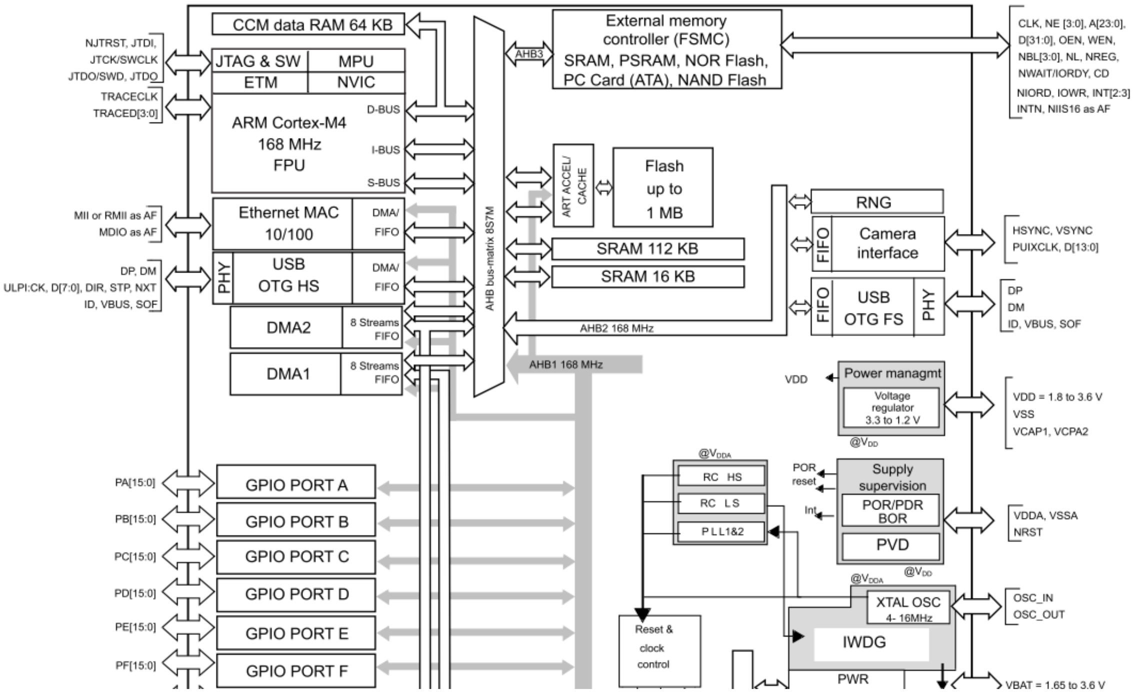
Bootloader and OS



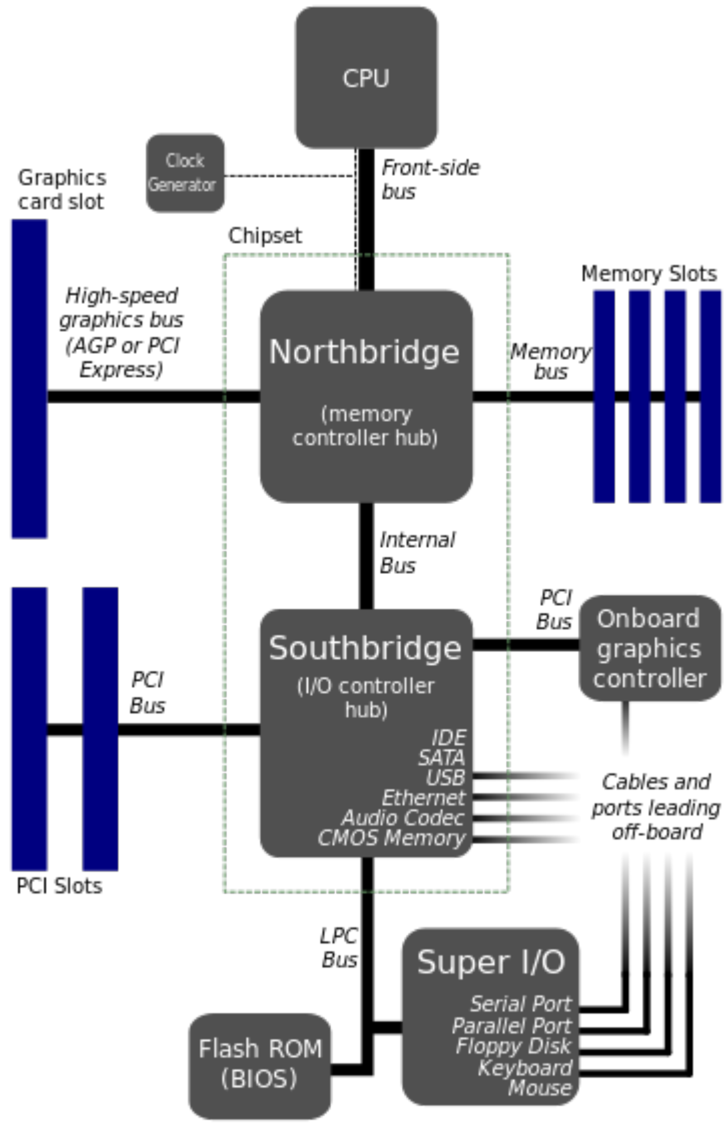
How boot-loader work?

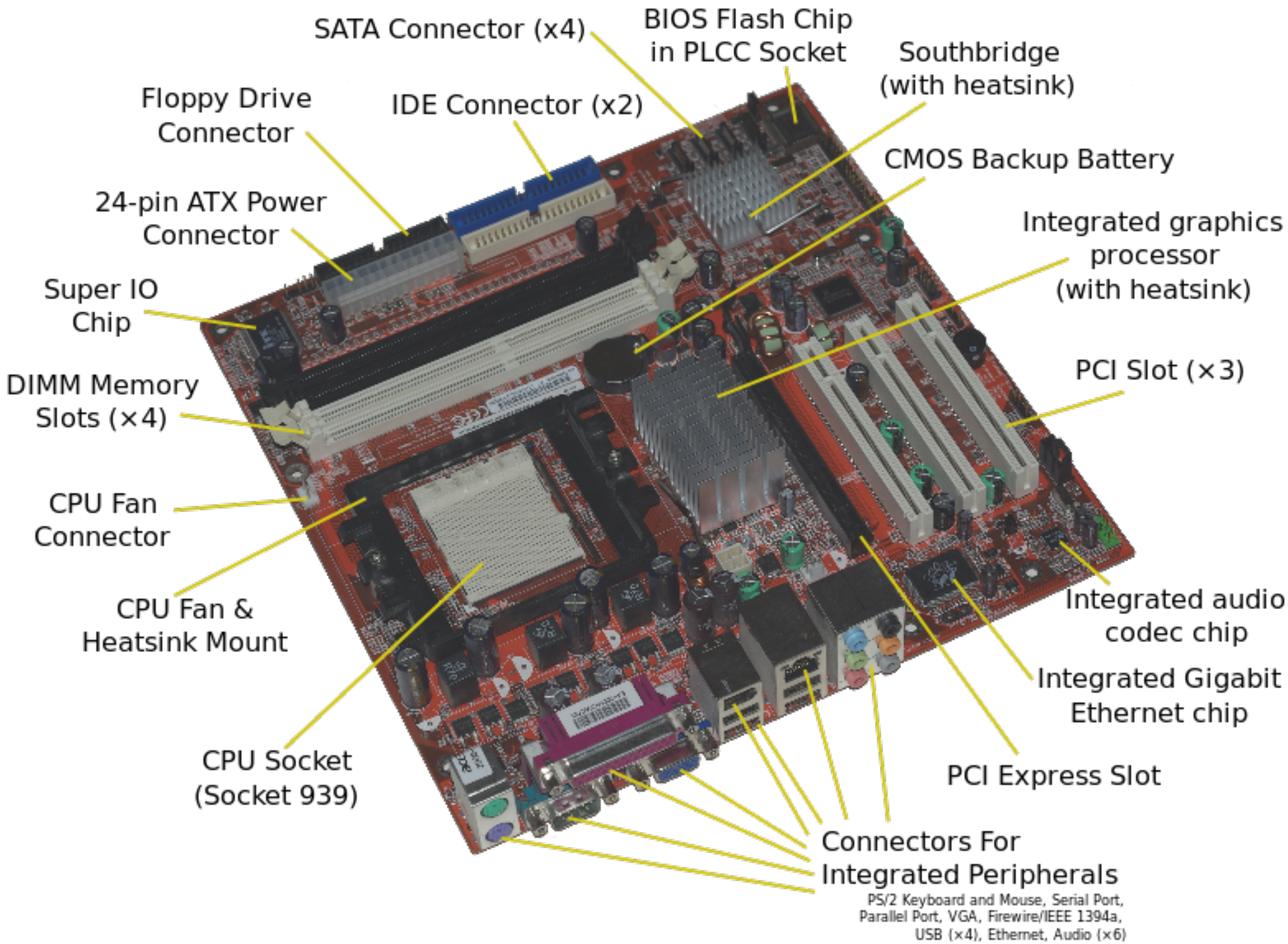
- SoC: e.g. STM32F4



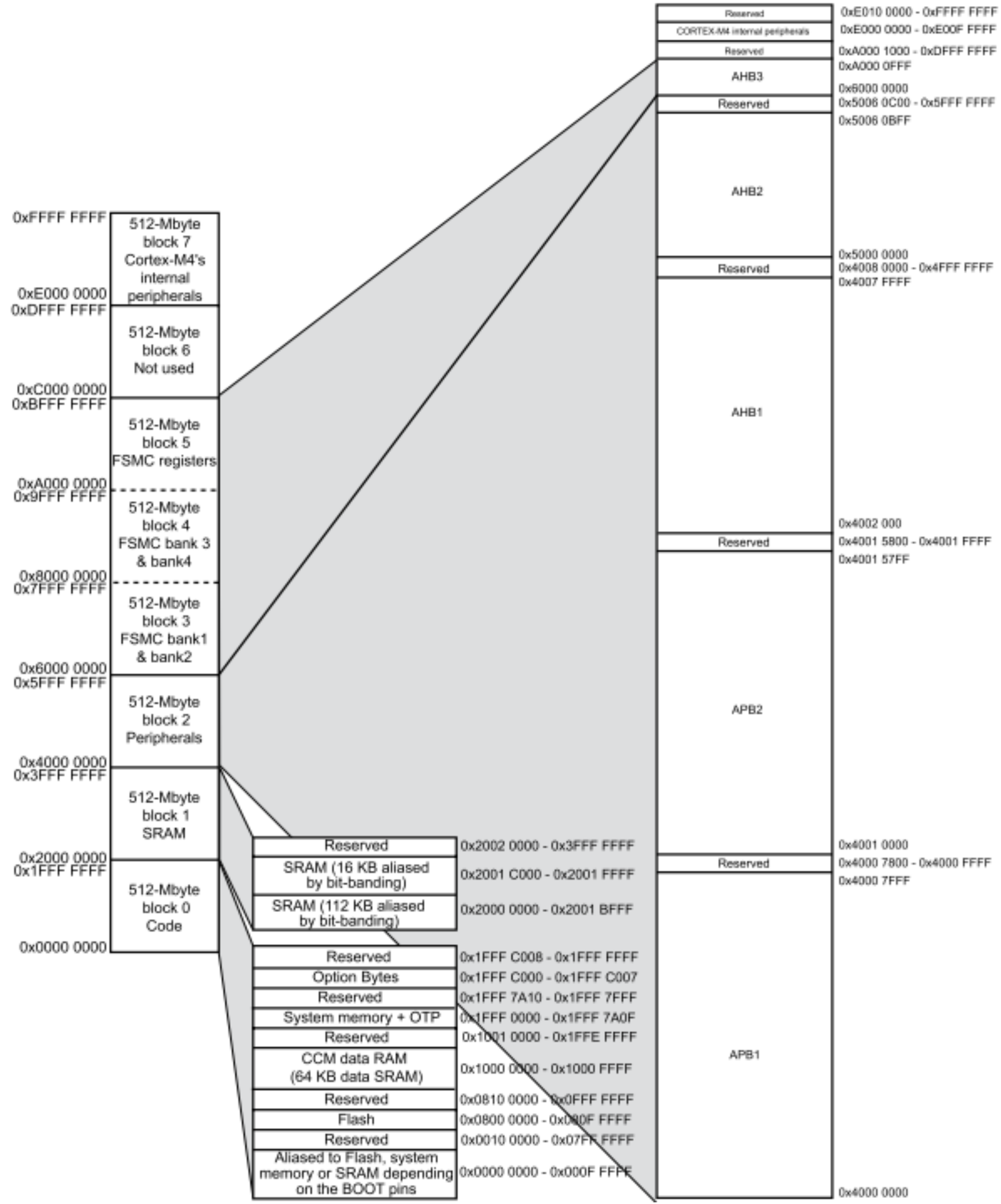


x86


CPU
Clock Generator
Front-side bus
Graphics card slot
Chipset
Memory Slots
High-speed graphics bus (AGP or PCI Express)
Northbridge
(memory controller hub)
Memory bus
Internal Bus
PCI Bus
Onboard graphics controller
PCI Bus
Southbridge
(I/O controller hub)
IDE
SATA
USB
Ethernet
Audio Codec
CMOS Memory
Cables and ports leading off-board
PCI Slots
LPC Bus
Super I/O
Serial Port
Parallel Port
Floppy Disk
Keyboard
Mouse
Flash ROM (BIOS)

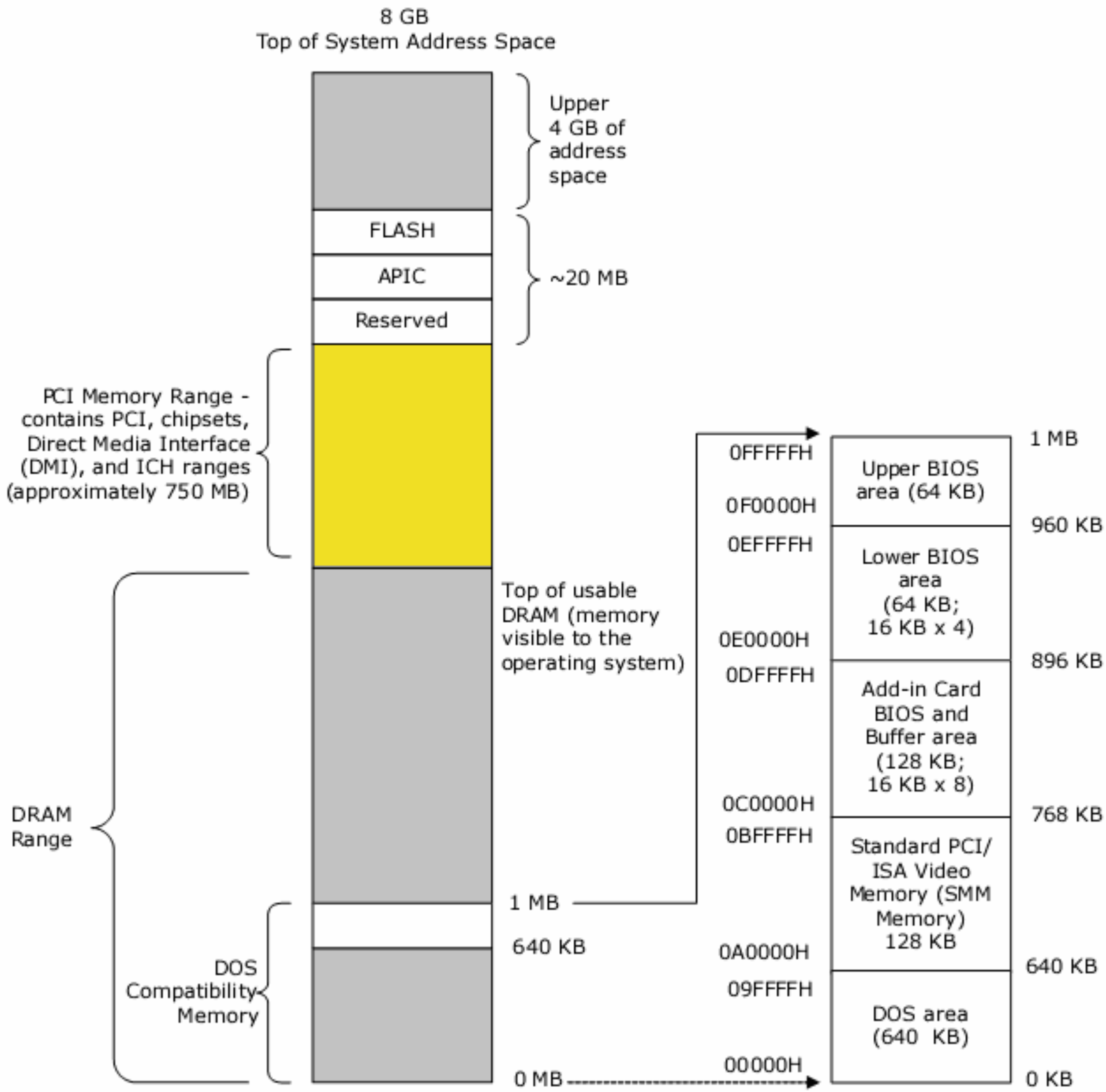

SATA Connector (x4)
BIOS Flash Chip in PLCC Socket
Southbridge (with heatsink)
Floppy Drive Connector
IDE Connector (x2)
CMOS Backup Battery
24-pin ATX Power Connector
Integrated graphics processor (with heatsink)
Super IO Chip
PCI Slot (×3)
DIMM Memory Slots (×4)
CPU Fan Connector
CPU Fan & Heatsink Mount
Integrated audio codec chip
Integrated Gigabit Ethernet chip
PCI Express Slot
CPU Socket (Socket 939)
Connectors For Integrated Peripherals
PS/2 Keyboard and Mouse, Serial Port, Parallel Port, VGA, Firewire/IEEE 1394a, USB (×4), Ethernet, Audio (×6)

STM32



0xFFFF FFFF
512-Mbyte block 7 Cortex-M4's internal peripherals
0xE000 0000
0xDFFF FFFF
512-Mbyte block 6 Not used
0xC000 0000
0xBFFF FFFF
512-Mbyte block 5 FSMC registers
0xA000 0000
0x9FFF FFFF
512-Mbyte block 4 FSMC bank 3 & bank4
0x8000 0000
0x7FFF FFFF
512-Mbyte block 3 FSMC bank1 & bank2
0x6000 0000
0x5FFF FFFF
512-Mbyte block 2 Peripherals
0x4000 0000
0x3FFF FFFF
512-Mbyte block 1 SRAM
0x2000 0000
0x1FFF FFFF
512-Mbyte block 0 Code
0x0000 0000
Reserved 0x2002 0000 - 0x3FFF FFFF
SRAM (16 KB aliased by bit-banding) 0x2001 C000 - 0x2001 FFFF
SRAM (112 KB aliased by bit-banding) 0x2000 0000 - 0x2001 BFFF
Reserved 0x1FFF C008 - 0x1FFF FFFF
Option Bytes 0x1FFF C000 - 0x1FFF C007
Reserved 0x1FFF 7A10 - 0x1FFF 7FFF
System memory + OTP 0x1FFF 0000 - 0x1FFF 7A0F
Reserved 0x1001 0000 - 0x1FFE FFFF
CCM data RAM (64 KB data SRAM) 0x1000 0000 - 0x1000 FFFF
Reserved 0x0810 0000 - 0x0FFF FFFF
Flash 0x0800 0000 - 0x080F FFFF
Reserved 0x0010 0000 - 0x07FF FFFF
Aliased to Flash, system memory or SRAM depending on the BOOT pins 0x0000 0000 - 0x000F FFFF
Reserved 0xE010 0000 - 0xFFFF FFFF
CORTEX-M4 internal peripherals 0xE000 0000 - 0xE00F FFFF
Reserved 0xA000 1000 - 0xDFFF FFFF
0xA000 0FFF
AHB3
0x6000 0000
Reserved 0x5006 0C00 - 0x5FFF FFFF
0x5006 0BFF
AHB2
0x5000 0000
Reserved 0x4008 0000 - 0x4FFF FFFF
0x4007 FFFF
AHB1
0x4002 000
Reserved 0x4001 5800 - 0x4001 FFFF
0x4001 57FF
APB2
0x4001 0000
Reserved 0x4000 7800 - 0x4000 FFFF
0x4000 7FFF
APB1
0x4000 0000

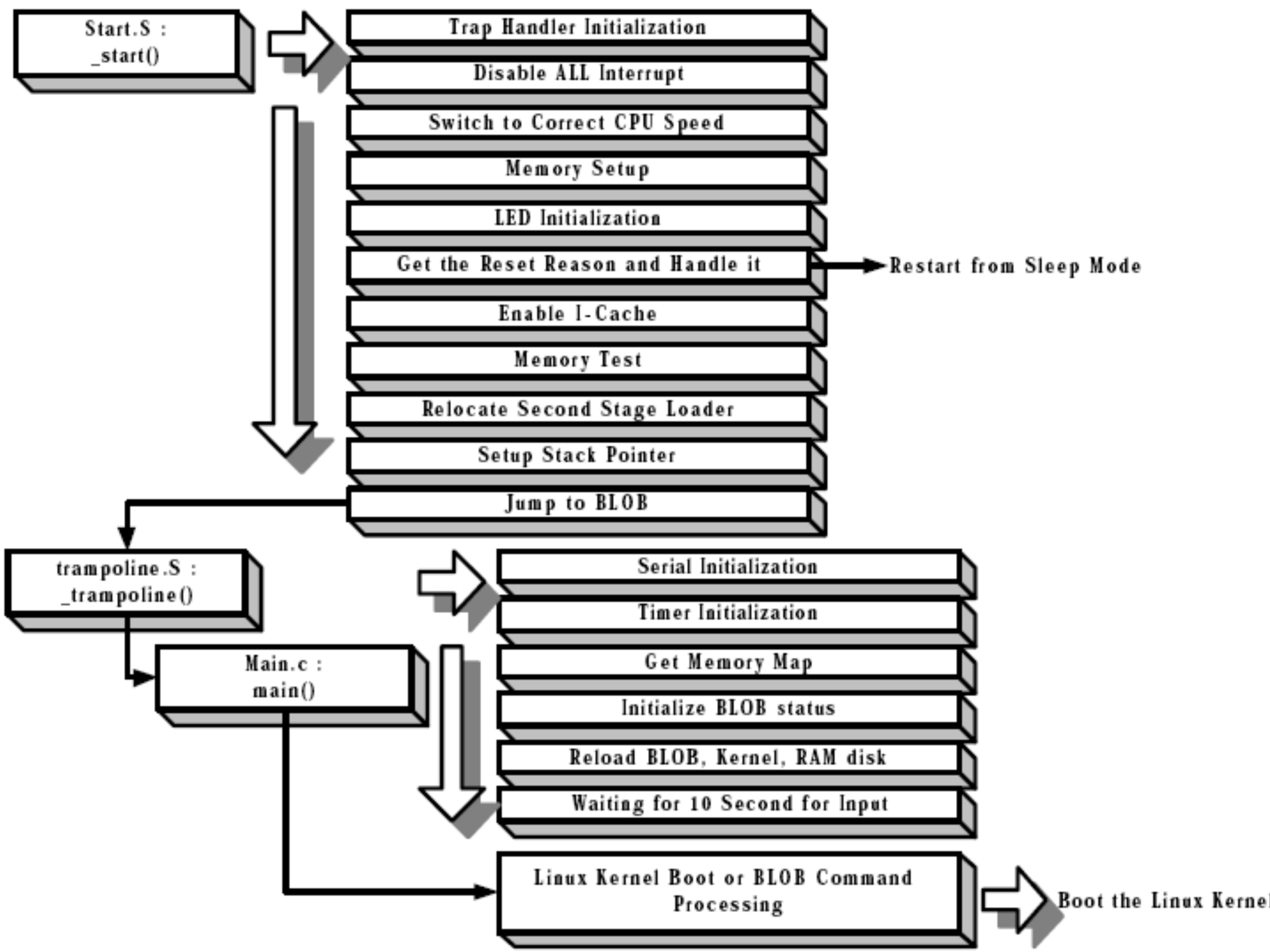
x86


8 GB
Top of System Address Space
Upper
4 GB of
address
space
FLASH
APIC
Reserved
~20 MB
PCI Memory Range -
contains PCI, chipsets,
Direct Media Interface
(DMI), and ICH ranges
(approximately 750 MB)
Top of usable
DRAM (memory
visible to the
operating system)
DRAM
Range
1 MB
640 KB
DOS
Compatibility
Memory
0 MB
0FFFFFH
0F0000H
0EFFFFH
0E0000H
0DFFFFH
0C0000H
0BFFFFH
0A0000H
09FFFFH
00000H
Upper BIOS
area (64 KB)
Lower BIOS
area
(64 KB;
16 KB x 4)
Add-in Card
BIOS and
Buffer area
(128 KB;
16 KB x 8)
Standard PCI/
ISA Video
Memory (SMM
Memory)
128 KB
DOS area
(640 KB)
1 MB
960 KB
896 KB
768 KB
640 KB
0 KB

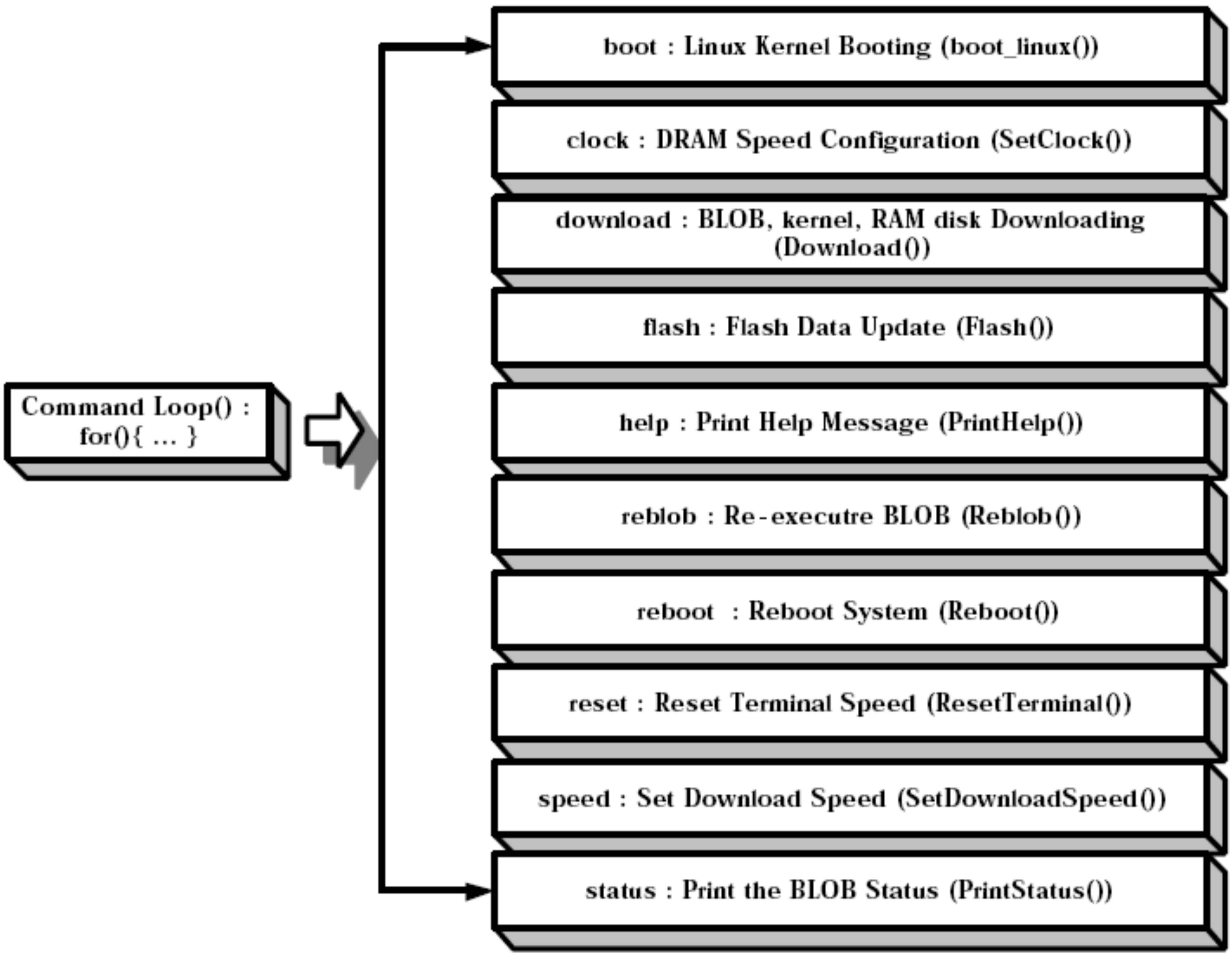
Boot-loader functions

- Initialize the hardware setting
- Basic monitor and debugger
- Pass the control to OS

Boot-loader Example



Boot-loader Example



ARM Examples

- Vector table

```
.text

/* Jump vector table as in table 3.1 in [1] */
.globl _start
_start:        b       reset
        b       undefined_instruction
        b       software_interrupt
        b       prefetch_abort
        b       data_abort
        b       not_used
        b       irq
        b       fiq
```

Reset_Handler

```
/* the actual reset code */
reset:
        /* First, mask **ALL** interrupts */
        ldr     r0, IC_BASE
        mov     r1, #0x00
        str     r1, [r0, #ICMR]


        /* switch CPU to correct speed */
        ldr     r0, PWR_BASE
        LDR     r1, cpuspeed
        str     r1, [r0, #PPCR]


        /* setup memory */
        bl      memsetup

        /* init LED */
        bl      ledinit
```

Reset_Handler (Cont.)

```
        /* check if this is a wake-up from sleep */
        ldr     r0, RST_BASE
        ldr     r1, [r0, #RCSR]      Reset status register
        and     r1, r1, #0x0f
        teq     r1, #0x08
        bne     normal_boot   /* no, continue booting */

        /* yes, a wake-up. clear RCSR by writing a 1 (see
9.6.2.1 from [1]) */
        mov     r1, #0x08
        str     r1, [r0, #RCSR]      ;

        /* get the value from the PSPR and jump to it */
        ldr     r0, PWR_BASE
        ldr     r1, [r0, #PSPR]      Power manager scratch pad register
        mov     pc, r1
```

Reset_Handler (Cont.)

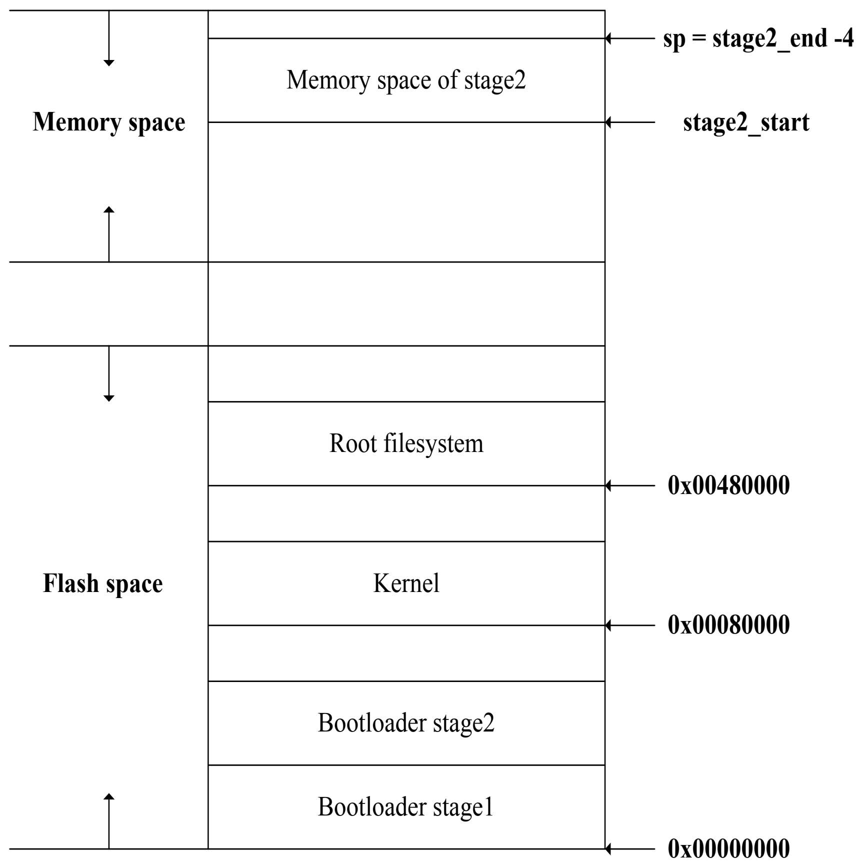
```
normal_boot:
      /* enable I-cache */
      mrc    p15, 0, r1, c1, c0, 0      @ read control reg
      orr    r1, r1, #0x1000            @ set Icache
      mcr    p15, 0, r1, c1, c0, 0      @ write it back

      /* check the first 1MB  in increments of 4k */
      mov    r7, #0x1000
      mov    r6, r7, lsl #8       /* 4k << 2^8 = 1MB */
      ldr    r5, MEM_START

mem_test_loop:
      mov    r0, r5
      bl     testram
      teq    r0, #1
      beq    badram

      add    r5, r5, r7
      subs   r6, r6, r7
      bne    mem_test_loop
```

```
        /* the first megabyte is OK, so let's clear it */
        mov     r0, #((1024 * 1024) / (8 * 4))    /* 1MB in
steps of 32 bytes */
        ldr     r1, MEM_START
…
clear_loop:
        stmia  r1!, {r2-r9}
        subs   r0, r0, #(8 * 4)
        bne    clear_loop

        /* relocate the second stage loader */
        add    r2, r0, #(128 * 1024)      /* blob is 128kB */
        add    r0, r0, #0x400             /* skip first 1024
bytes */
        ldr    r1, MEM_START
        add    r1, r1, #0x400             /* skip over here
as well */
..
copy_loop:
        ldmia  r0!, {r3-r10}
        stmia  r1!, {r3-r10}
        cmp    r0, r2
        ble    copy_loop
```

Reset_Handler (Cont.)

```
/* set up the stack pointer */
ldr    r0, MEM_START
add    r1, r0, #(1024 * 1024)
sub    sp, r1, #0x04


/* blob is copied to ram, so jump to it */
add    r0, r0, #0x400
mov    pc, r0
```

Bootloader memory map



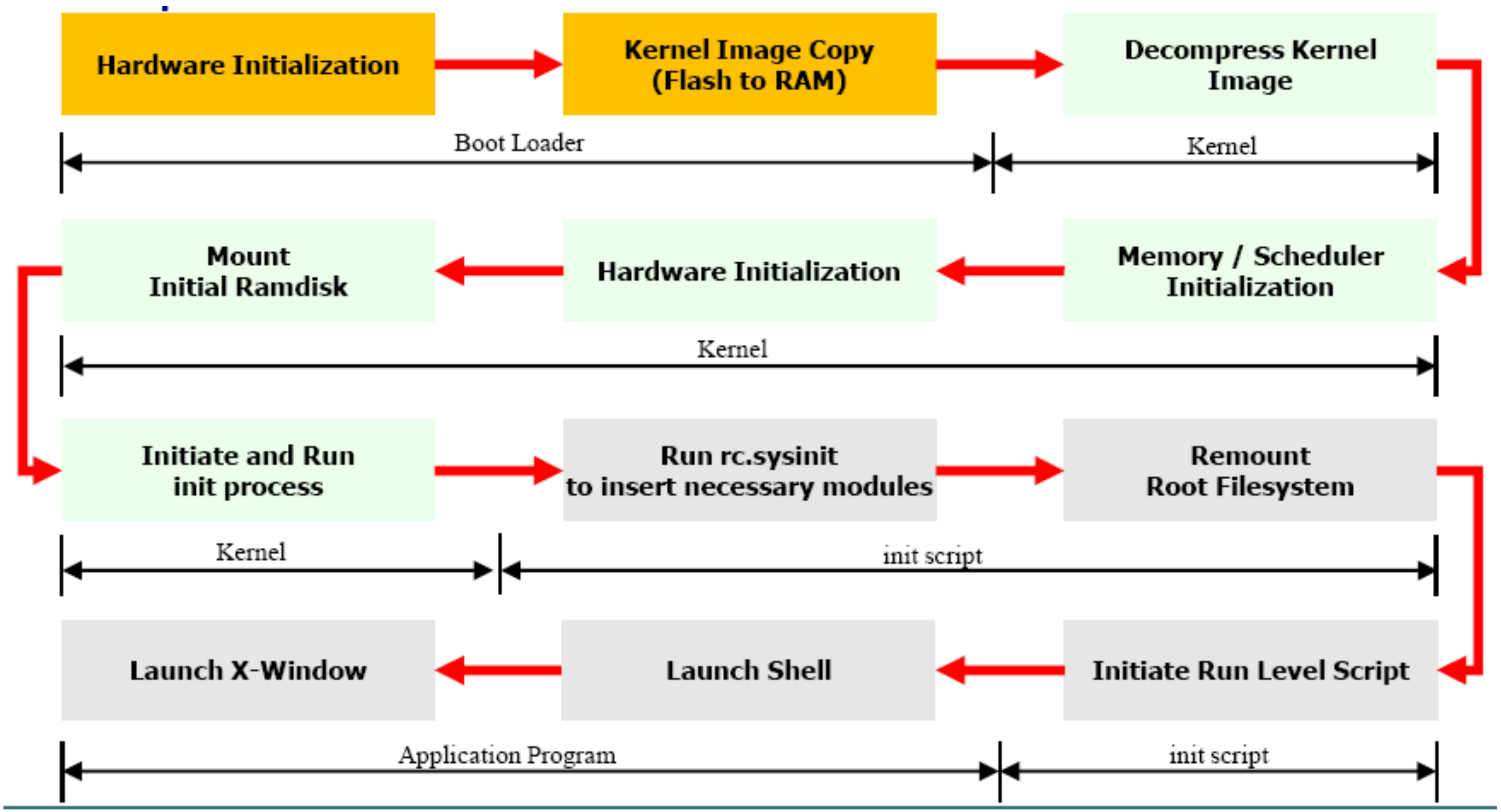
Boot-loader C program

```
int main(void)
{
~
   led_on();
~
   SerialInit(baud9k6);
   TimerInit();
~
   SerialOutputString(PACKAGE " version " VERSION  "\n"
                "Copyright (C) 1999 2000 2001 "
~
   get_memory_map();
~
   SerialOutputString("Running from ");
   if(RunningFromInternal())
       SerialOutputString("internal");
   else
       SerialOutputString("external");
…
```

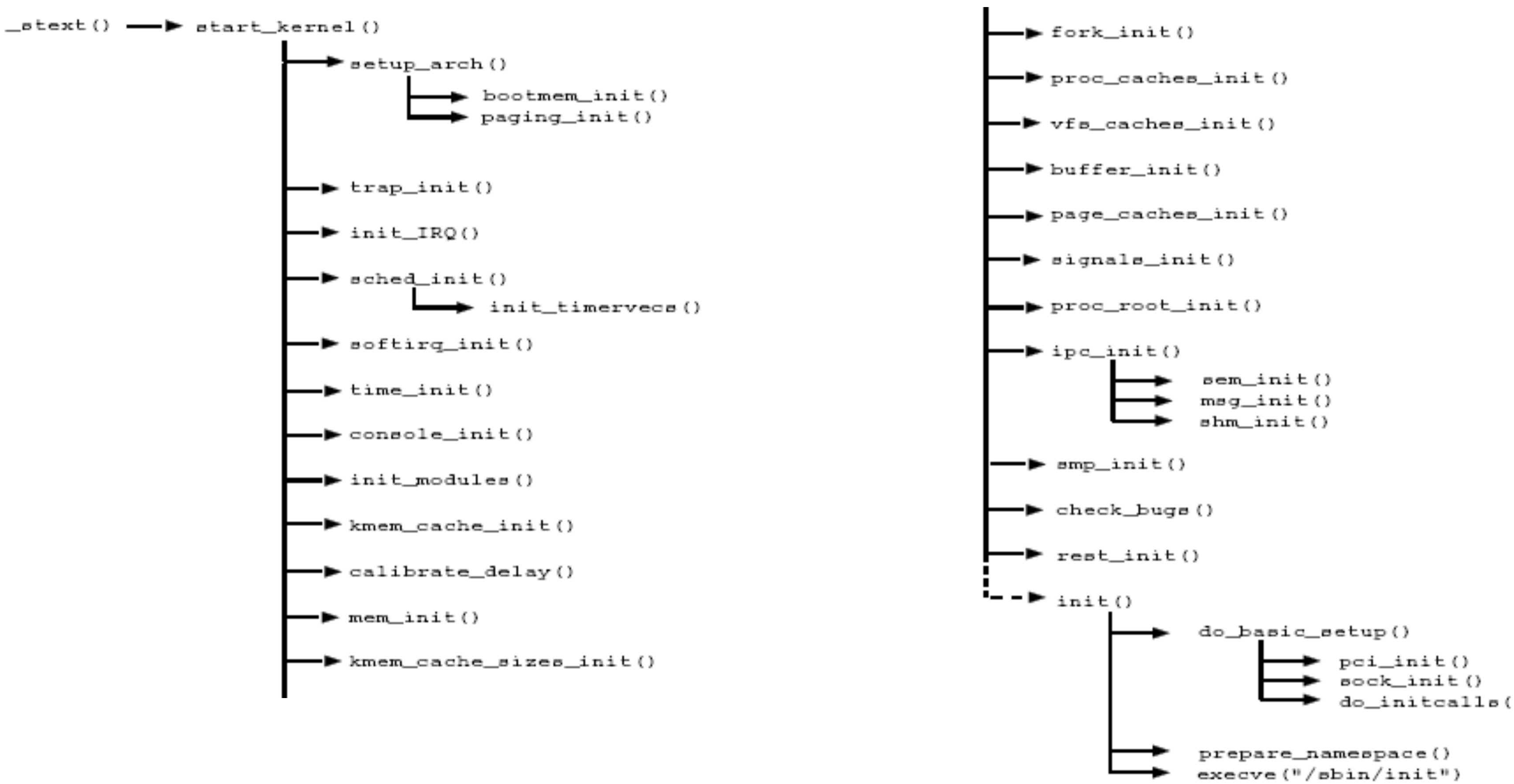
Boot-loader C program

```
   /* wait 10 seconds before starting autoboot */
SerialOutputString("Autoboot in progress, press any
   key…");
   for(i = 0; i < 10; i++) {
       SerialOutputByte('.');
       retval = SerialInputBlock(commandline, 1, 1);
~
   if(retval == 0) {
~
       boot_linux(commandline);   }
~
   for(;;) {
~
       if(numRead > 0) {
       if(MyStrNCmp(commandline, "boot", 4) == 0) {
             boot_linux(commandline + 4);
       } else if(MyStrNCmp(commandline, "clock", 5) == 0) {
             SetClock(commandline + 5);
   } else if(MyStrNCmp(commandline, "download ", 9) == 0) {
~  return 0;
} /* main */
```

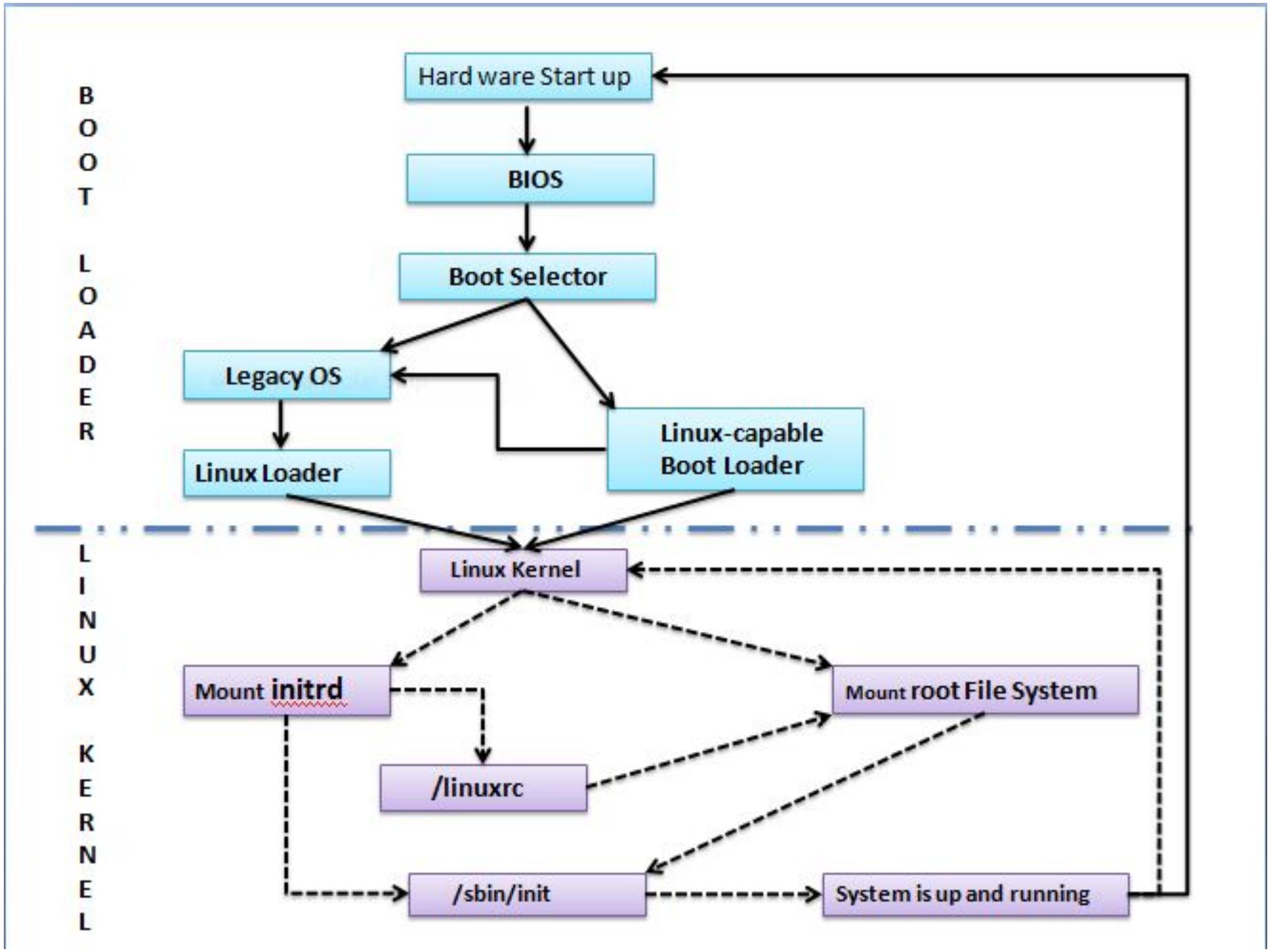
How does OS boot?



Linux booting



Mount root disk

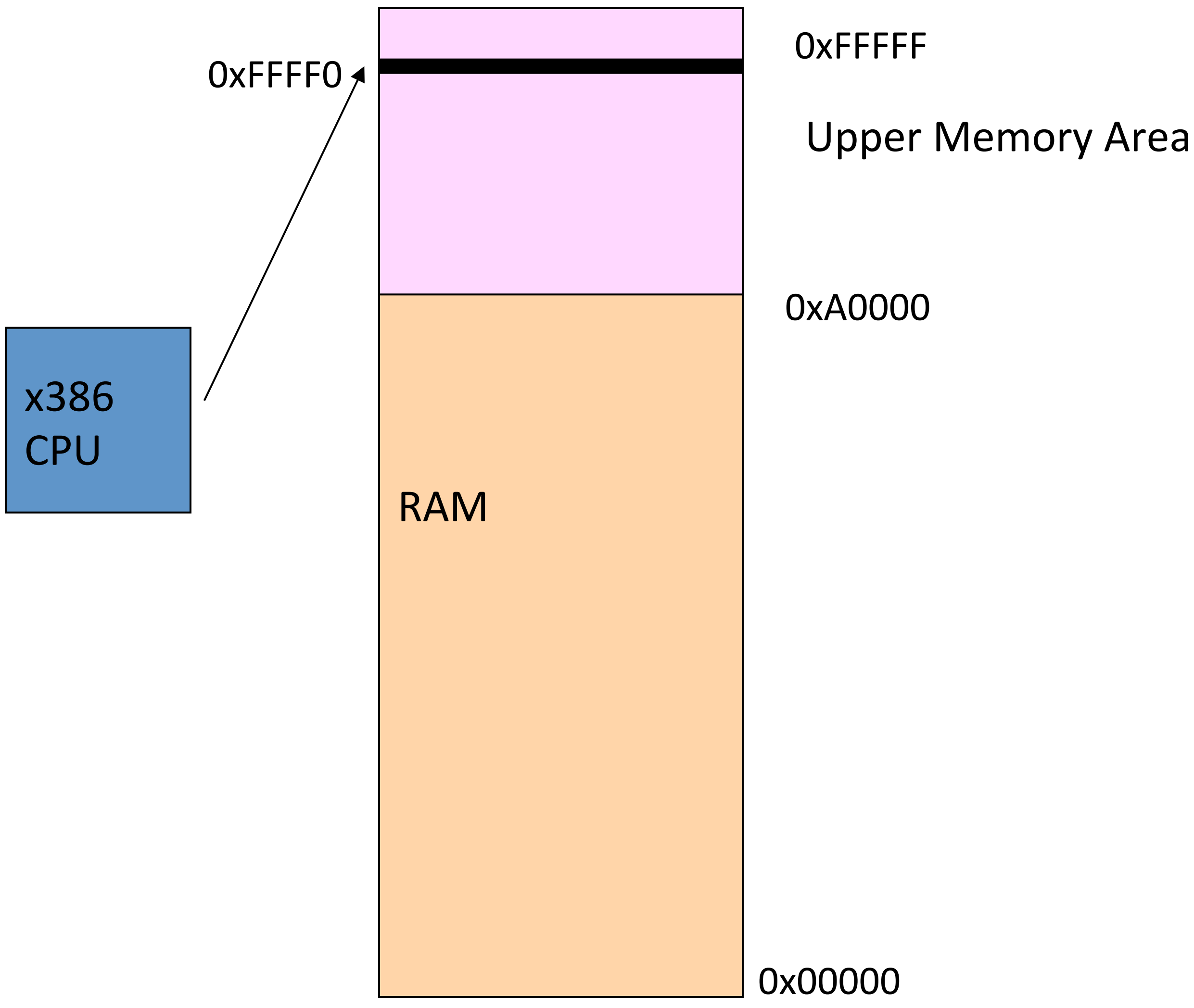


initrd (initramfs): boot loader initialized RAM disk

A case study of PC and Linux 0.11

- Basic Input and Output System
- Firmware
- For Intel boot architecture
  - Old but backward compatible
- Functions include
  - H/W initialization and parameter configurations
  - Device drivers
  - Debuggers/monitor
  - Load OS/another bootloader

PC Booting (Cont)

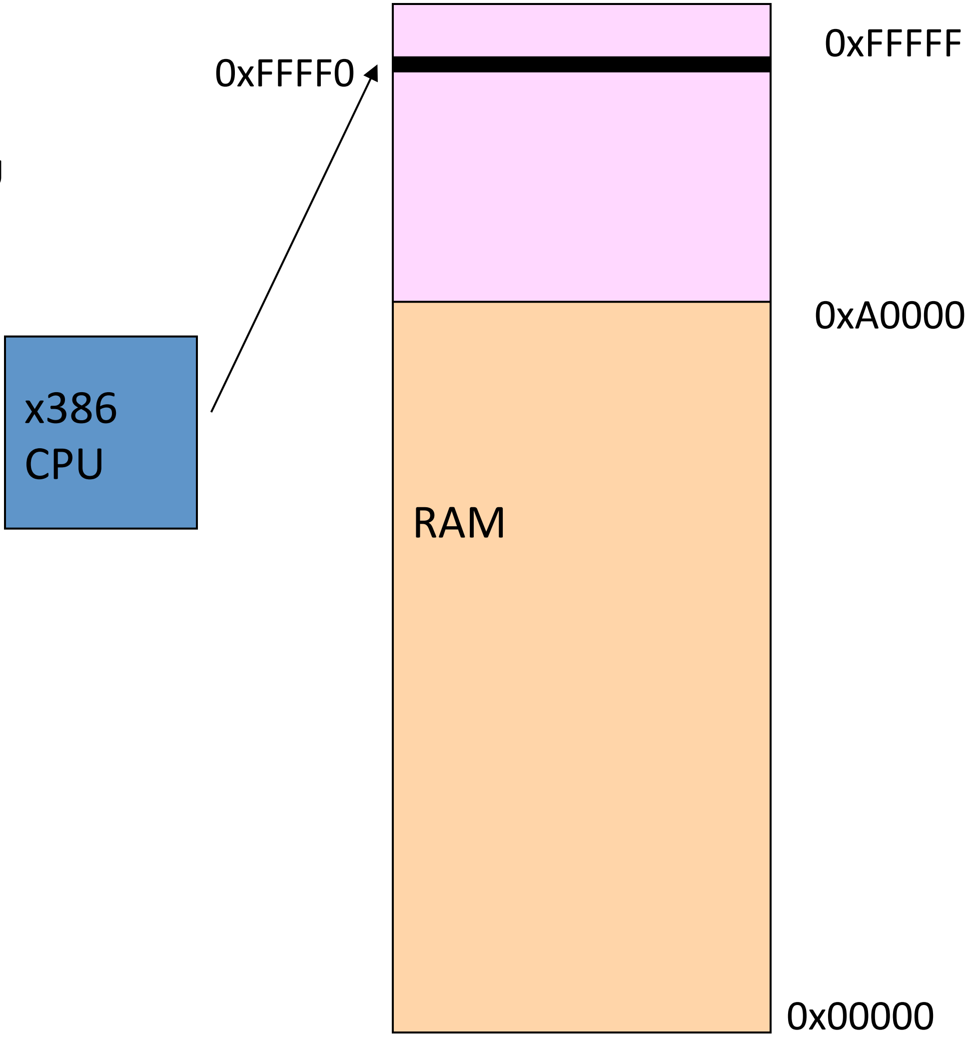


UMA

Address	First 16K (x0000h-x3FFFh)	Second 16K (x4000h-x7FFFh)	Third 16K (x8000h-xBFFFh)	Fourth 16K (xC000h-xFFFFh)
A0000-AFFFFh	VGA Graphics Mode Video			
B0000- BFFFFh	VGA Monochrome Text Mode Video RAM		VGA Color Text Mode Video RAM	
C0000- CFFFFh	VGA Video BIOS ROM		IDE Hard Disk BIOS ROM	Optional Adapter ROM BIOS or RAM UMBs
D0000- DFFFFh	Optional Adapter ROM BIOS or RAM UMBs			
F0000- FFFFFh	System BIOS ROM			

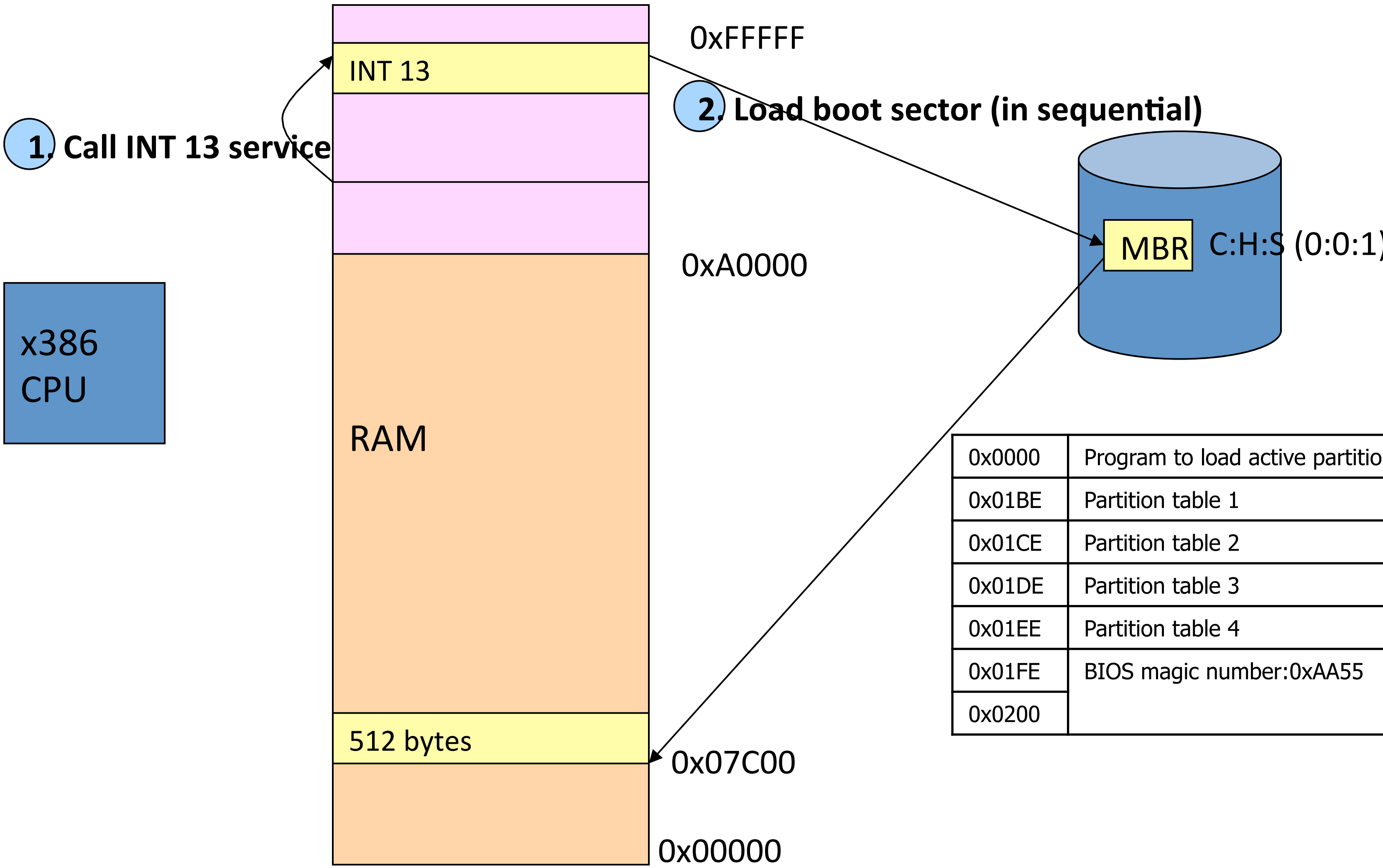
PC Booting (Cont)

1. Power supply sends POWER GOOD to CPU
2. CPU resets
3. Run FFFF:0000 @ BIOS ROM
4. Jump to a real BIOS start address
5. POST
6. Beep if there is an error
7. Read CMOS data/settings
8. Run 2nd-stage boot



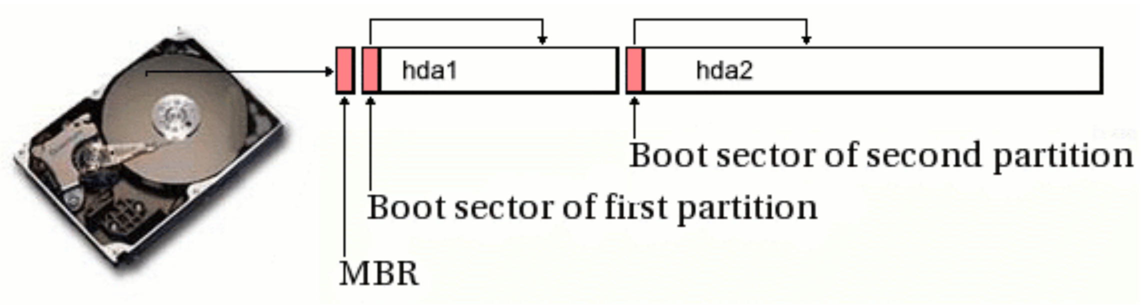
INT	Address	Type	Description
00h	0000:0000h	Processor	Divide by zero
01h	0000:0004h	Processor	Single step
02h	0000:0008h	Processor	Non maskable interrupt (NMI)
03h	0000:000Ch	Processor	Breakpoint
04h	0000:0010h	Processor	Arithmetic overflow
05h	0000:0014h	Software	Print screen
06h	0000:0018h	Processor	Invalid op code
07h	0000:001Ch	Processor	Coprocessor not available
08h	0000:0020h	Hardware	System timer service routine
09h	0000:0024h	Hardware	Keyboard device service routine
0Ah	0000:0028h	Hardware	Cascade from 2nd programmable interrupt controller
0Bh	0000:002Ch	Hardware	Serial port service - COM post 2
0Ch	0000:0030h	Hardware	Serial port service - COM port 1
0Dh	0000:0034h	Hardware	Parallel printer service - LPT 2
0Eh	0000:0038h	Hardware	Floppy disk service
0Fh	0000:003Ch	Hardware	Parallel printer service - LPT 1
10h	0000:0040h	Software	Video service routine
11h	0000:0044h	Software	Equipment list service routine
12h	0000:0048H	Software	Memory size service routine
13h	0000:004Ch	Software	Hard disk drive service
14h	0000:0050h	Software	Serial communications service routines
15h	0000:0054h	Software	System services support routines
16h	0000:0058h	Software	Keyboard support service routines
17h	0000:005Ch	Software	Parallel printer support services

PC Booting (Cont)

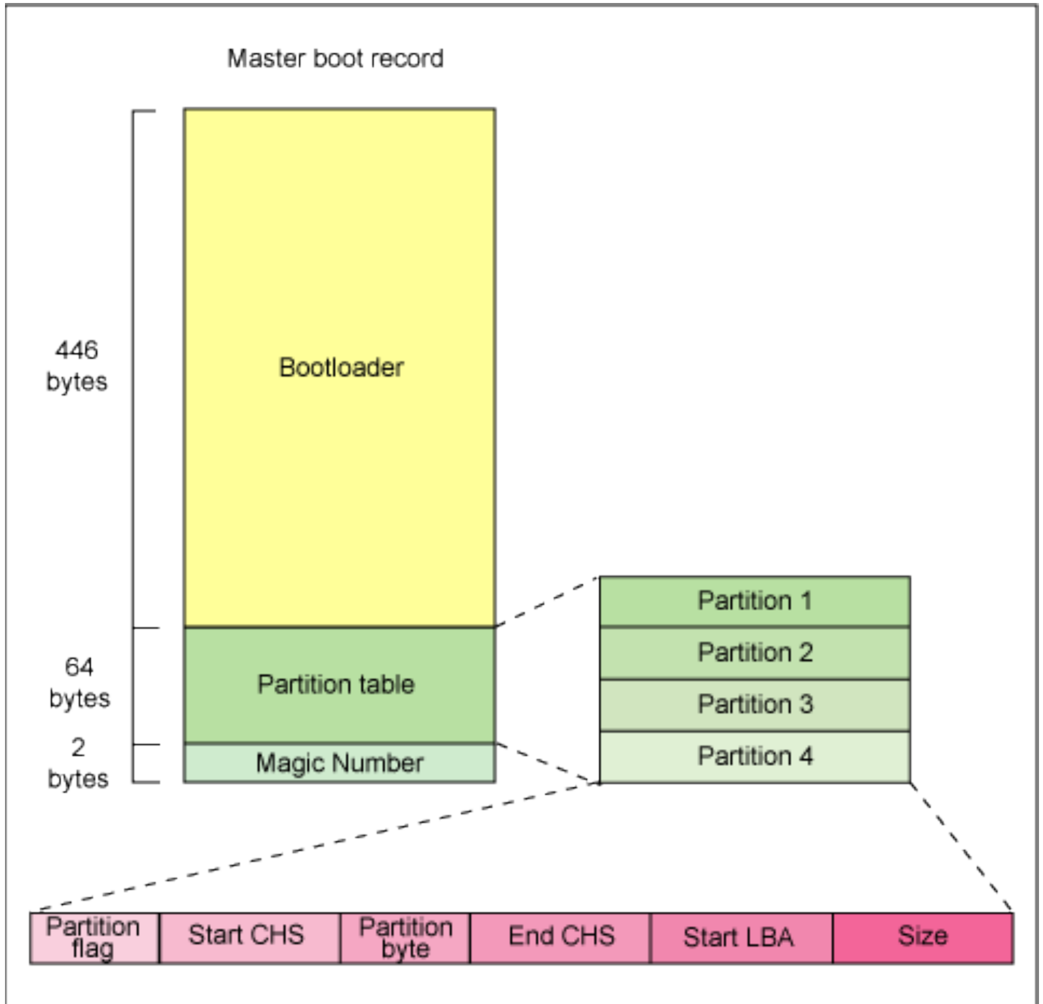


0x0000	Program to load active partition
0x01BE	Partition table 1
0x01CE	Partition table 2
0x01DE	Partition table 3
0x01EE	Partition table 4
0x01FE	BIOS magic number:0xAA55
0x0200	

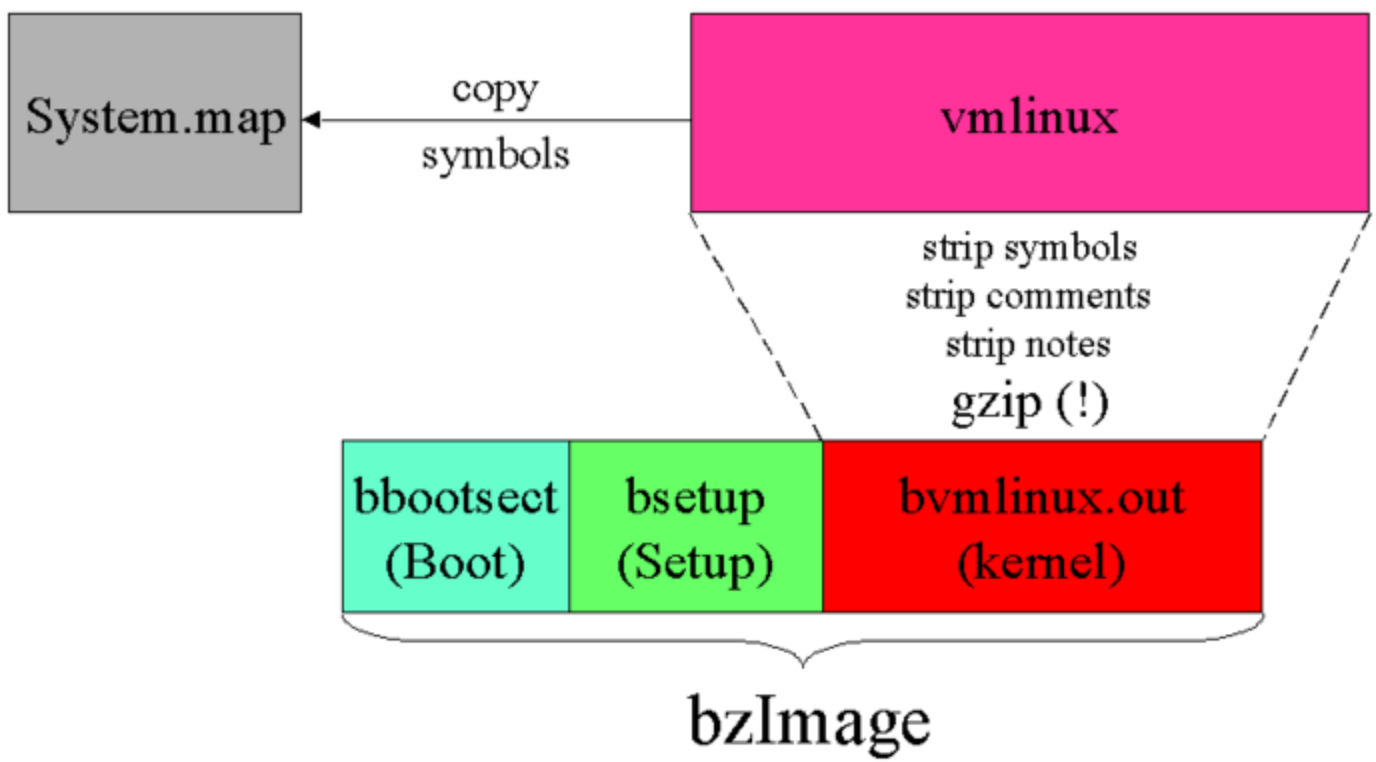
MBR


hda1
hda2
Boot sector of second partition
Boot sector of first partition
MBR

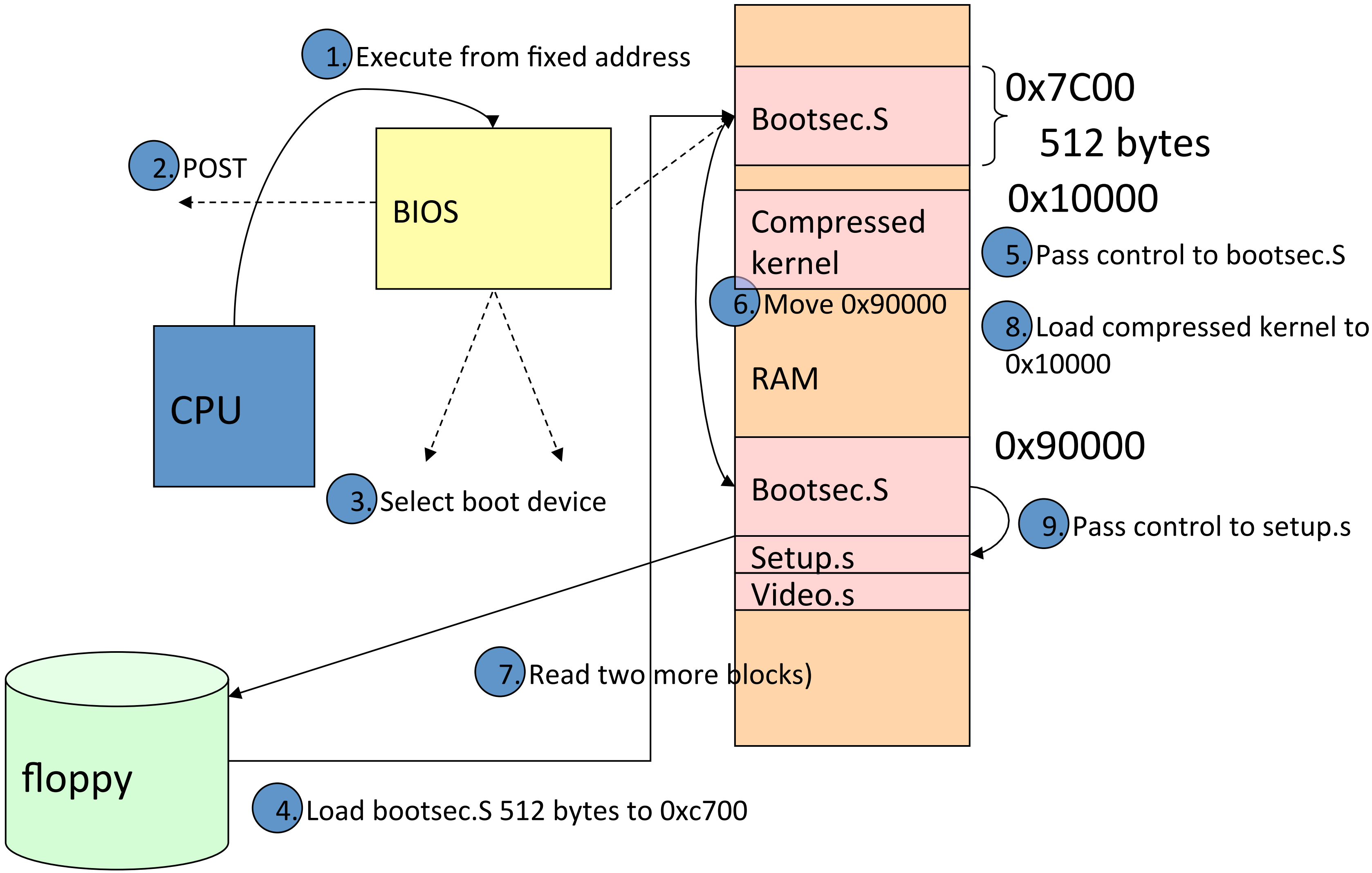
MBR (Master Boot Record)



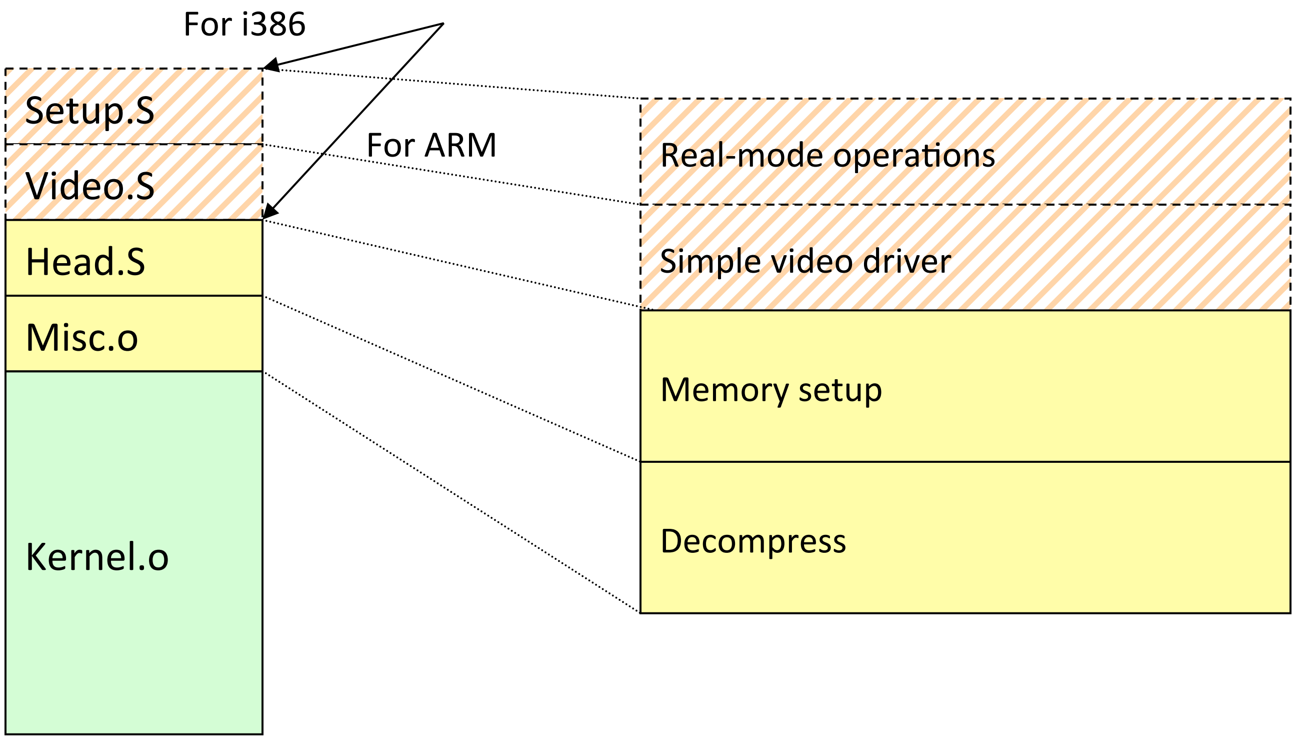
Anatomy of bzImage



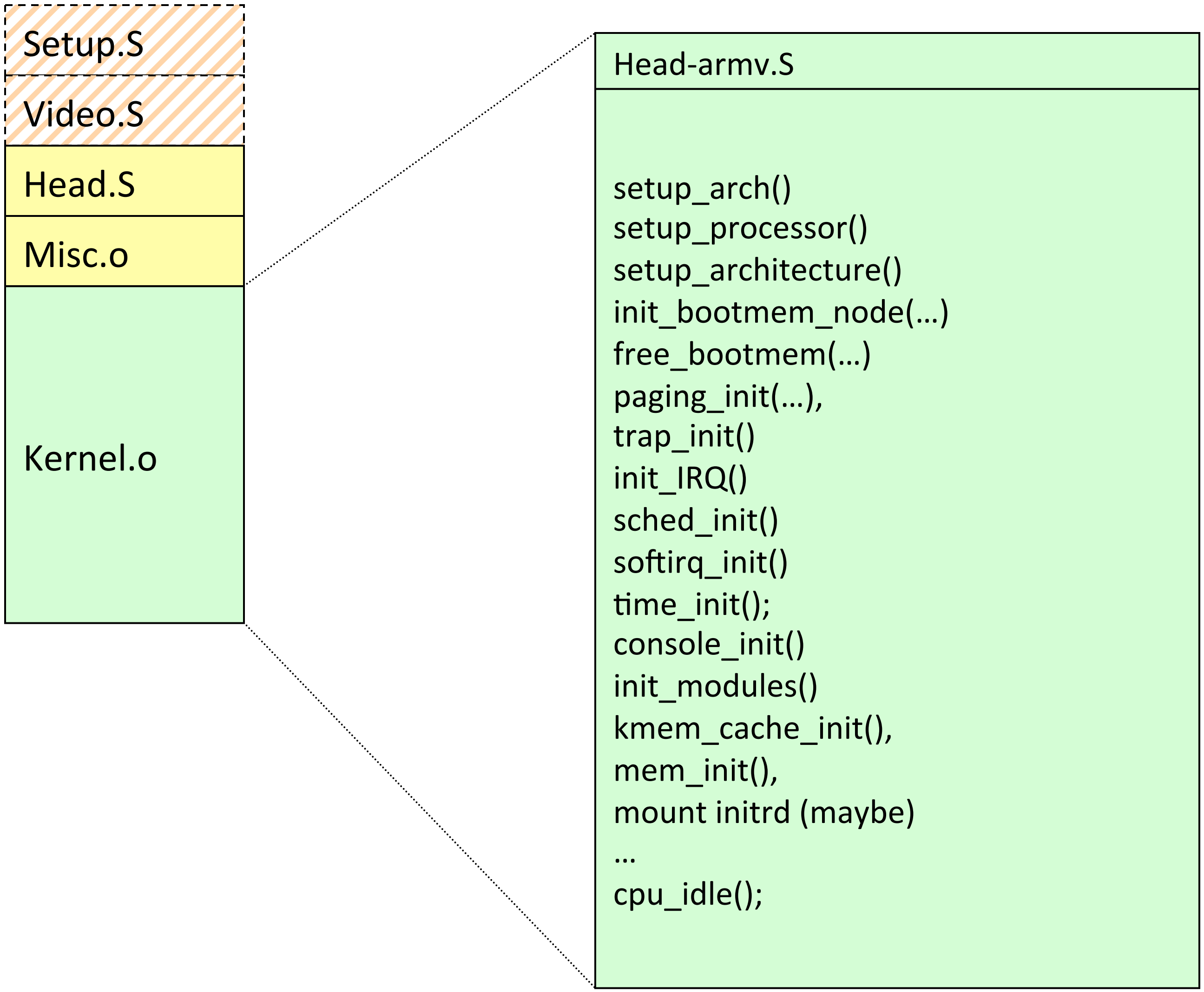
Linux Boot Example



Kernel Image Structure



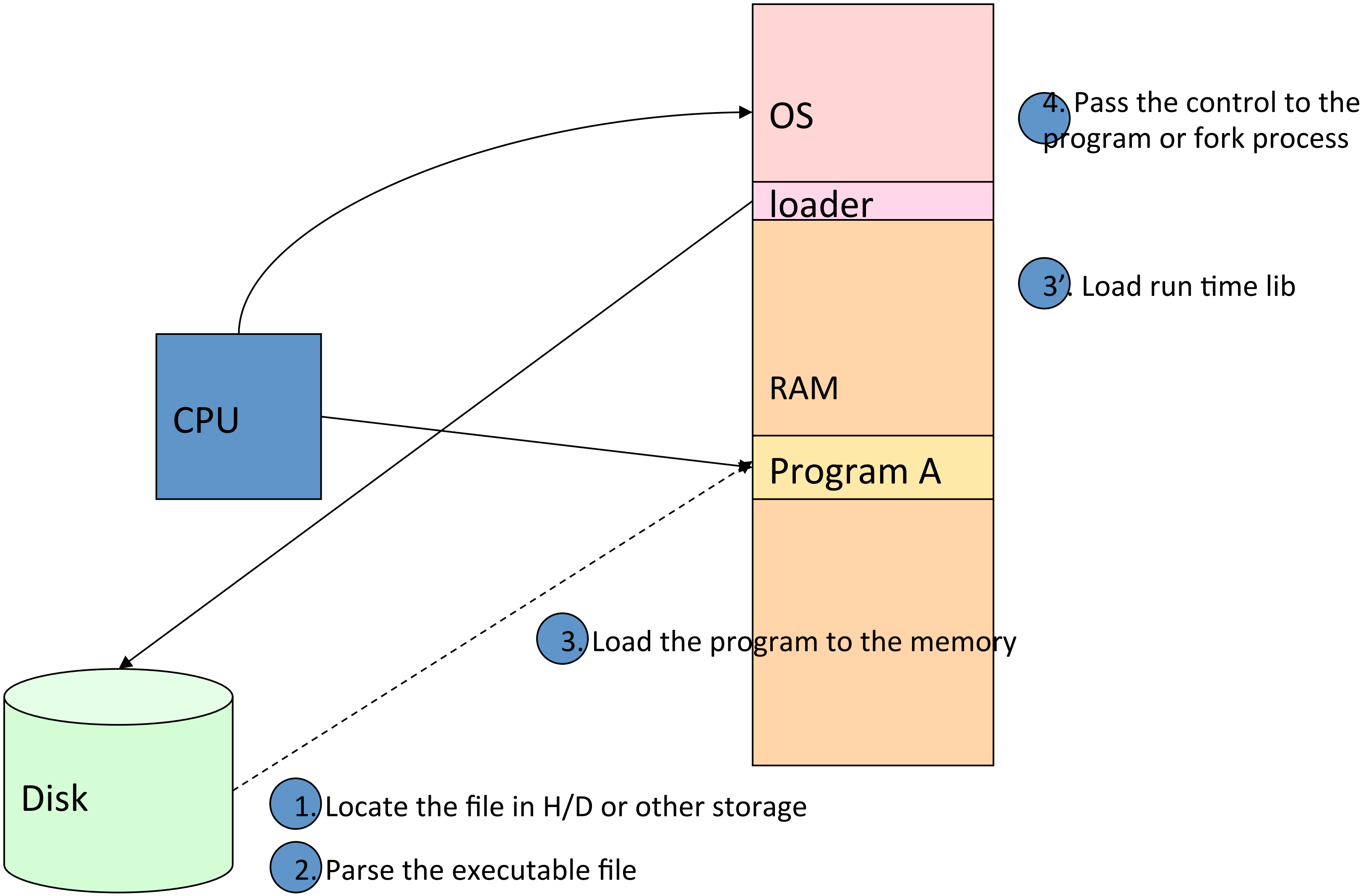
Kernel Image Structure



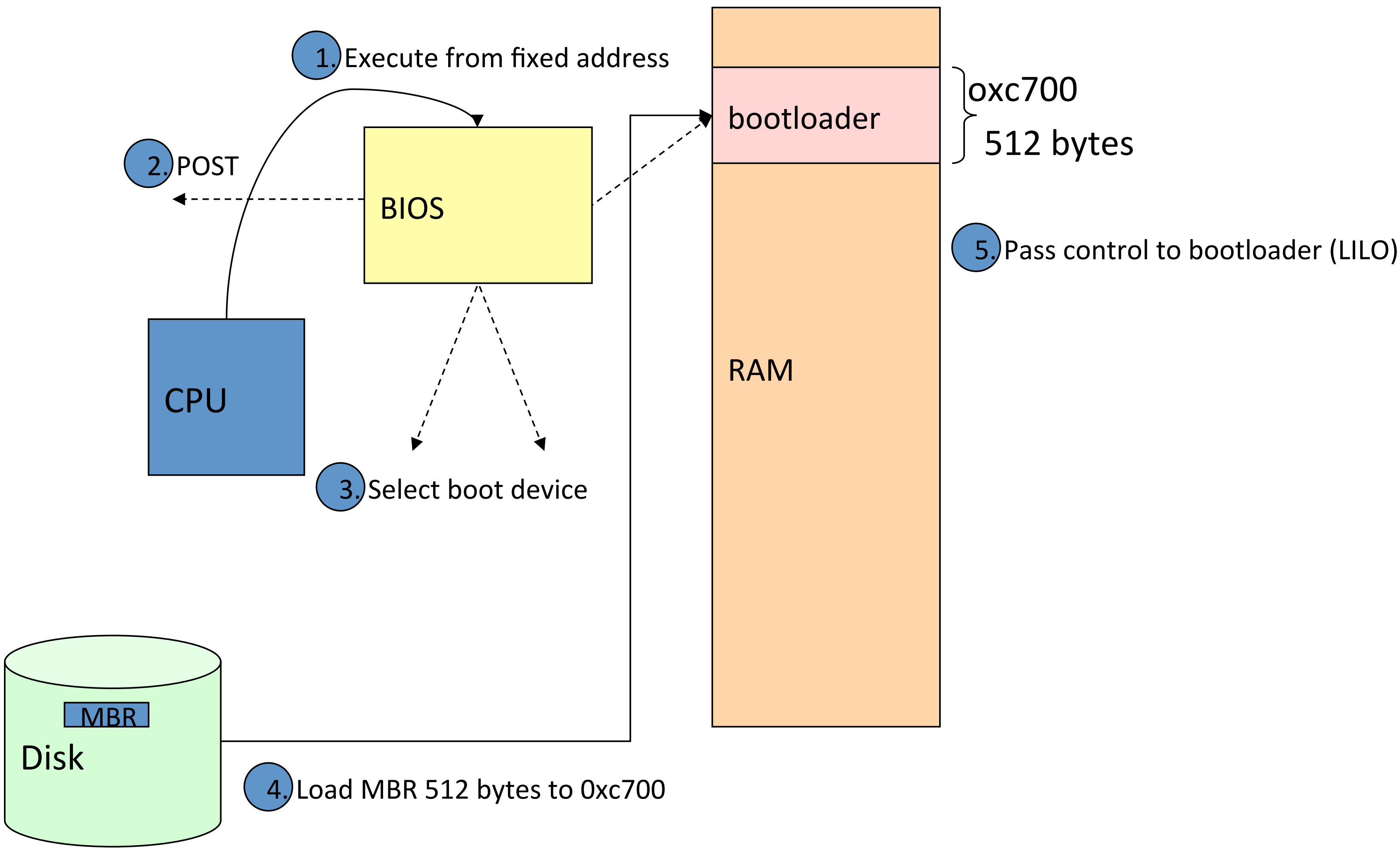
Loader

- Normally a user level program
- Might use OS system services to access the storage and control the memory allocation
- Load and interpret executable files to memory and ask OS to run it

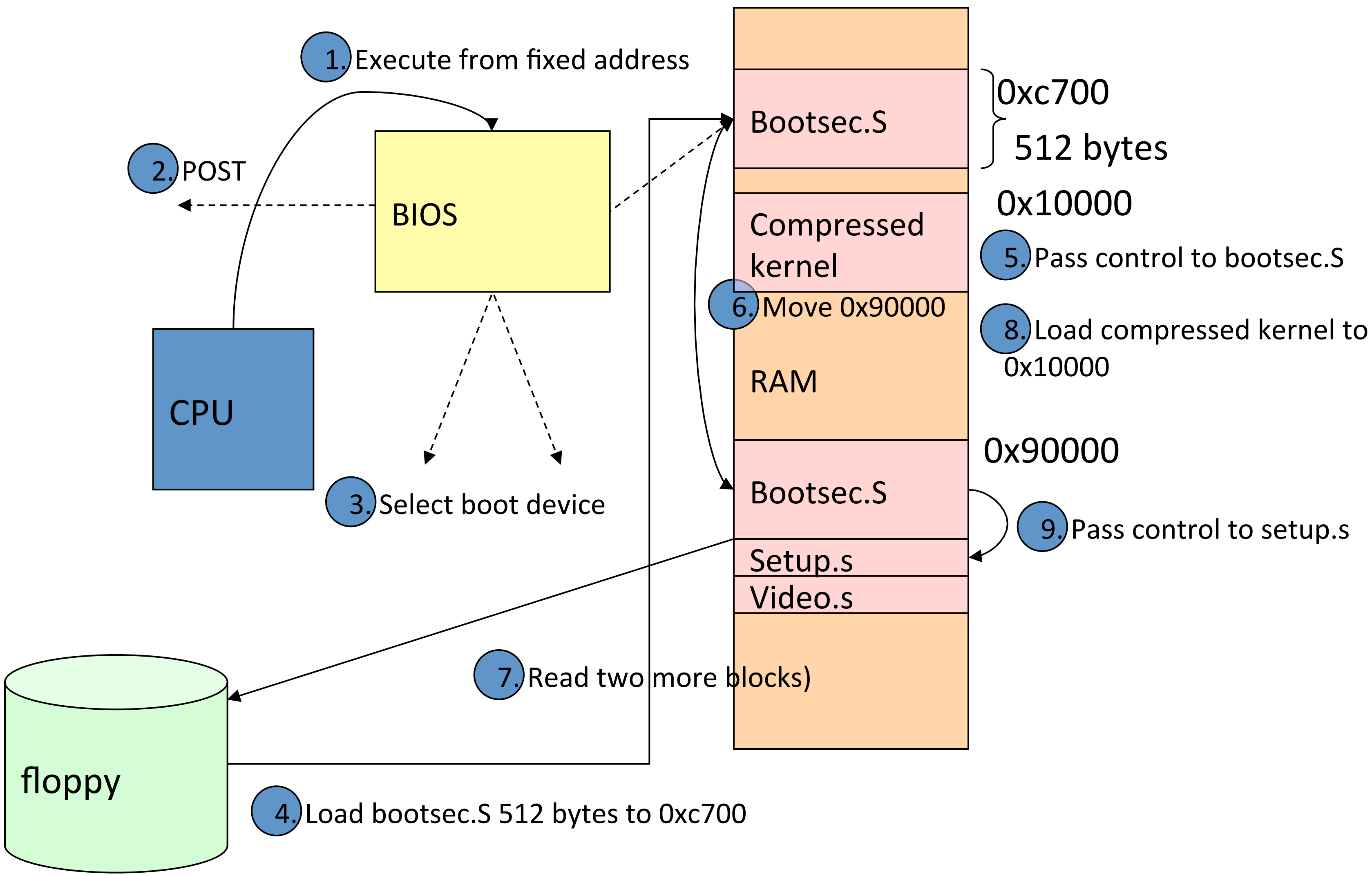
Loader


OS
loader
RAM
Program A
CPU
Disk
4. Pass the control to the program or fork process
3'. Load run time lib
3. Load the program to the memory
1. Locate the file in H/D or other storage
2. Parse the executable file

Linux Boot Example



Linux Boot Example



Bootloader Speedups

- Remove waiting time
- Removing unnecessary initialization routines
- Uncompressed kernel
- DMA Copy Of Kernel On Startup
- Fast Kernel Decompression
- Kernel XIP

Remove waiting time

- Boot loader await a couple of seconds to connect debug port
- Since boot loader cannot detect connection of debug port, boot loader awaits a couple of seconds on normal boot sequence.
- U-boot : setenv bootdelay 0

Removing unnecessary initialization routines

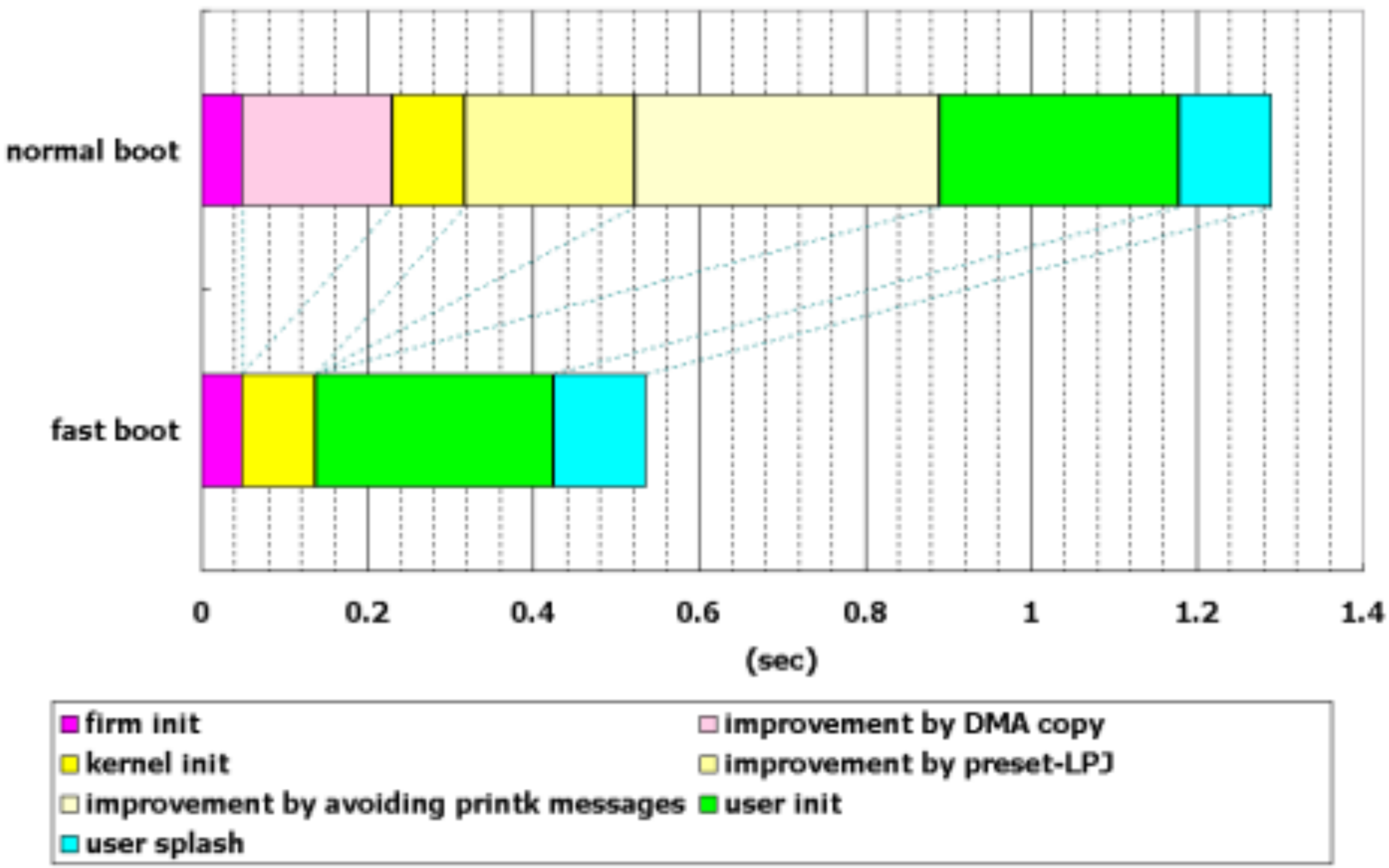
- Should not perform initialize other devices except for necessaries to load kernel
- Examples
  - Removing initialization of LCD
  - Removing initialization of timer

Uncompressed kernel

- Fast boot, but imagine size has been larger
  - 2MB – 2.5MB non-compressed (ARM)
  - 1MB – 1.5MB compressed (ARM)
- It should be different results, performance of CPUs, speed of flash memory
  - Profiling is required
- Make Image vs. make zImage

DMA Copy Of Kernel On Startup

- Turn on DMA in bootloader



Fast Kernel Decompression

- Use other compression/decompression algorithm
  - Slow compression, good compression ratio, fast decompression
- GZIP vs. Sony UCL
- Small kernel size, fast kernel loading time

Fast Kernel Decompression

Image file:	initrd-2.6.5-1.358		Power PC
method	UCL	GZIP	improved %
parameter	-b4194304	-8	.
source file size	819200	819200	.
compressed size	187853	189447	.
compression rate	77.1%	76.9%	0.3%
compression time: user (sec)	5.13	2.03	-152.5%
sys (sec)	0.09	0.06	-36.5%
total (sec)	5.22	2.09	-149.0%
decompression time: user (sec)	0.12	0.3	59.7%
sys (sec)	0.1	0.08	-16.9%
total (sec)	0.22	0.39	43.0%

Fast Kernel Decompression

Image file:	vmlinux-2.4.20 for ibm-440gp		PowerPC
method	UCL	GZIP	improved %
parameter	-b4194304	-8	.
source file size	1810351	1810351	.
compressed size	790250	776807	.
compression rate	56.3%	57.1%	-1.3%
compression time: user (sec)	17.29	6.07	-185.0%
sys (sec)	0.04	0.02	-92.4%
total (sec)	17.33	6.09	-184.6%
decompression time: user (sec)	0.12	0.16	26.1%
sys (sec)	0.03	0.04	35.8%
total (sec)	0.15	0.2	28.2%

Kernel XIP

- Direct addressing on NOR flash memory
- Cannot compress kernel
- PowerPC 405LP/266 MHZ

Boot Stage	Non-XIP Time	XIP Time
Copy kernel to RAM	85 ms	12 ms *
Decompress kernel	453 ms	0 ms
Kernel time to initialize (time to first user space program)	819 ms	882 ms
Total kernel boot time	1357 ms	894 ms
Reduction:	*	**463 ms**

Kernel XIP (Cont.)

- TI OMAP 5912/196 MHZ

Boot Stage	Non-XIP Time Kernel compressed	Non-XIP Time Kernel not compressed	XIP Time
Copy kernel to RAM	56 ms	120 ms	0 ms
Decompress kernel	545 ms	0 ms	0 ms
Kernel time to initialize (time to first user space program)	88 ms	208 ms	110 ms
Total kernel boot time	689 ms	208 ms	110 ms
Reduction:	*	**481 ms**	**579 ms**

Embedded Linux Speedups

- Removing unnecessary message printout
- Remove unnecessary functions and device drivers
- Modularization of device driver
- Asynchronous function calls
- Avoid performance measurement routine
- RTC no sync
- Using read-only file system
- Using lazy mount technique on R/W file systems
- Deferred Initcalls

Removing unnecessary message printout

- Add “quiet” to kernel command line
- Example 1

Hardware	KMC SH board, using VGA console
Kernel Version	CELF-1 (040126)
Configuration	relatively small configuration (details not available)
Time without "quiet" option	637878 usec
Time with "quiet" option	461893 usec
Time savings	176 milliseconds

Example 2

Hardware	TI OMAP board, using serial console
Kernel Version	CELF-1 (040126)
Configuration	Kernel booted with XIP, CRAMFS root file system, with preset-LPJ
Time without "quiet" option	551735 usec
Time with "quiet" option	280676 usec
Time savings	271 milliseconds

Embedded Linux Speedups

- Remove unnecessary functions and device drivers
  - Reduce kernel size
- Modularization of device driver
  - Upload module after boot sequence finished
    - e.g. CAM device

Asynchronous function calls

- For some initialization functions, it could be done asynchronously
- Asynchronous function calls has been merged in mainline starting from 2.6.29

Avoid performance measurement routine

- kernel updates loops_per_jiffy value at every booting to using timer
  - This value should be fixed until hardware changes or modify setting
  - The problem is this routine uses some loops (Delaying)
- Method examples
  - Find out loops_per_jiffy after basic boot sequence
  - Modify init/calibrate.c in kernel source
- TI OMAP 1510/158MHz
  - 212 milliseconds

RTC no sync

- get_cmos_time() is used to read the value of the external real-time clock (RTC) when the kernel boots
- This routine delays until the edge of the next second rollover, in order to ensure that the time value in the kernel is accurate with respect to the RTC
- Introduces up to 1 second of variability in the total bootup time
- Reconfigure kernel with "CONFIG_RTC_NO_SYNC_ON_READ" is enabled
- “Fast boot options” labeled as “No SYNC on read of Real Time Clock”

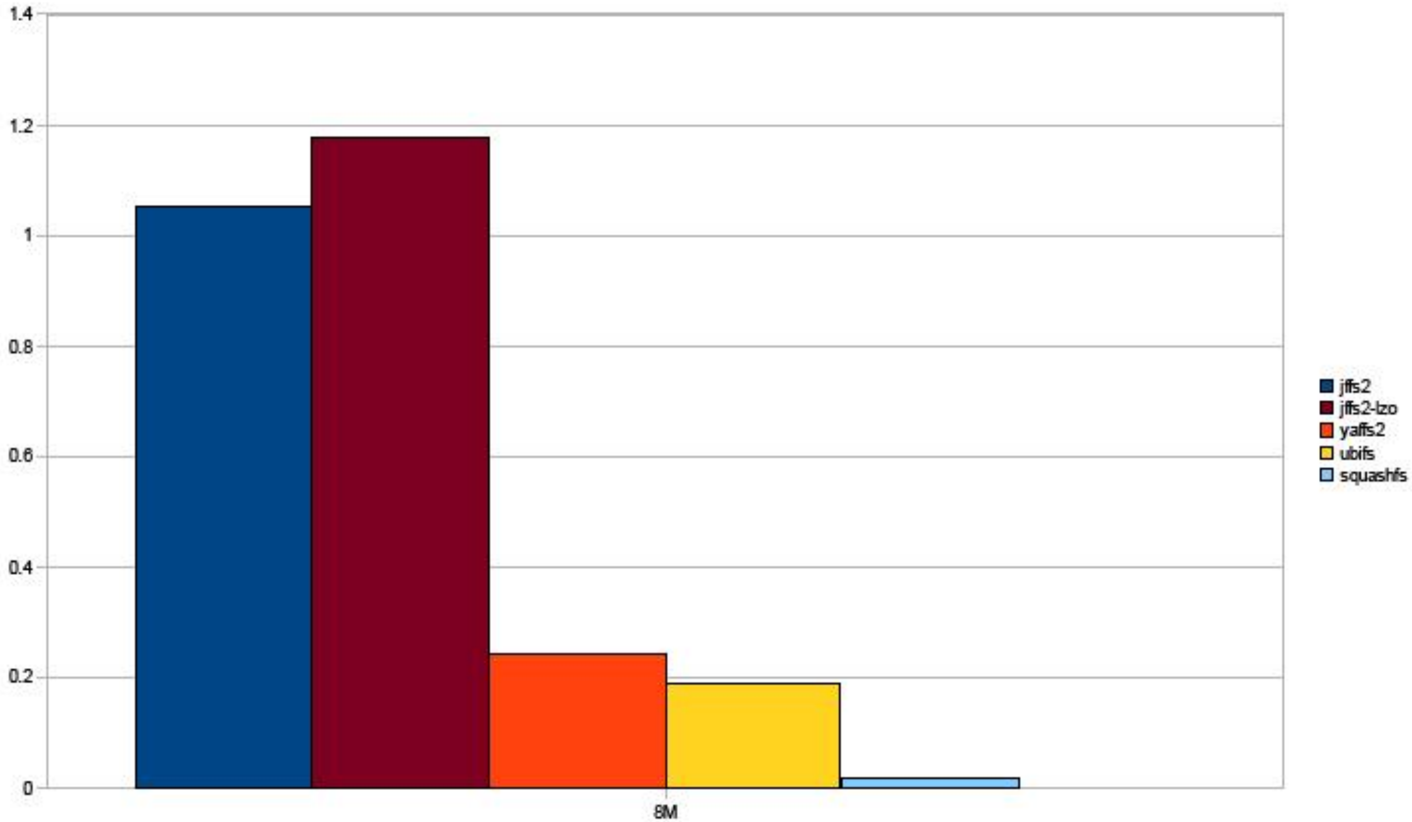
Using read-only file system

- Without using write functionality, simplify using file system
- There are no delay time on mount read-only file systems
- One of simplest, RomFS
- The cramfs has compression functionality
- Method examples
  - Classify read-only, R/W, temporary files that will include file system
  - Position cramfs read-only files
  - Using lazy mount technique, to include R/W files
  - Temporary file uses tmpfs

Using lazy mount technique on R/W file systems

- slow mount speed file systems like jffs2 mount after finishing boot sequence on user request
- On booting, using read-only file instead
- JFFS2 mount of 8M filesystem partition is over 1 second
- UBIFS mount of 8M partition is under .2 seconds
- Squashfs mount of 8M partition is under 50 milliseconds

Zoom - Mount time (seconds) - 8M



Deferred Initcalls

- Find modules that are not required for core functionality of product
  - Ex: USB on a camera – uhci_hcd_usb, ehci_hcd_init
- Change module init routine declaration
  - module_init(foo_init) to
  - deferred_module_init(foo_init)
- Modules marked like this are not initialized during kernel boot
- After main init, do:
  - echo 1 >/proc/deferred_initcalls
- Deferred initcalls are run

Application Speedups

- Using binary script, not shell script
- Using init process with simplified and optimized
- Parallel RC scripts
- Application XIP
- Using pre-link shared lib
- Optimize of application programs
- Move from glibc to uClibc

Using binary script, not shell script

- Disadvantages performance of shell script
  - Interpreter
    - Interpret every line and perform
    - Interpret meaningless line and phrases
  - Use fork & exec technique when perform commands
- Make binary script without fork & exec technique

Using init process with simplified and optimized

- BusyBox optimizations
  - The modified init scripts must be run with "bash" as well as the BusyBox "ash" shell
- Follow the rules listed below to reduce execution time for the init scripts:
  - Do not use unnecessary codes in the scripts.
  - Replace external commands and utilities with the BusyBox built-ins as far as possible.
  - Do not use the piped commands as far as possible.
  - Reduce the number of commands within a pipe.
  - Do not use the back-quoted commands as far as possible.
  - The main goal of such optimization is to reduce the number of the “fork/exec” calls during a script execution.

Parallel RC scripts

- run RC scripts in parallel
  - to take advantage of the multi-processing capabilities of the OS (such as overlapping execution with I/O, etc.)
- Need shell/shell script support

Pre-linking

- CPU: ARM9
- Base OS : MontaVistaLinux CEE 3.1 (2.4.20 kernel, glibc-2.3)
- GUI:X Window, GTK+(1.2.10) / Motif like toolkit

CE Linux Forum

Panasonic
ideas for life

User Response Time/Boot up time

Application of prelink

- Normal dynamic linked ELF: Over 2 sec. to start up a multimedia application process (fork ~ main)

Stages of Processing in Process Start up	elapsed time
1. Layout and map shared libs to virtual address space	96ms
2. Resolve symbol references	2354ms
3. Init of each ELF file	29ms

- Prelink eliminates symbol reference resolution proceses.

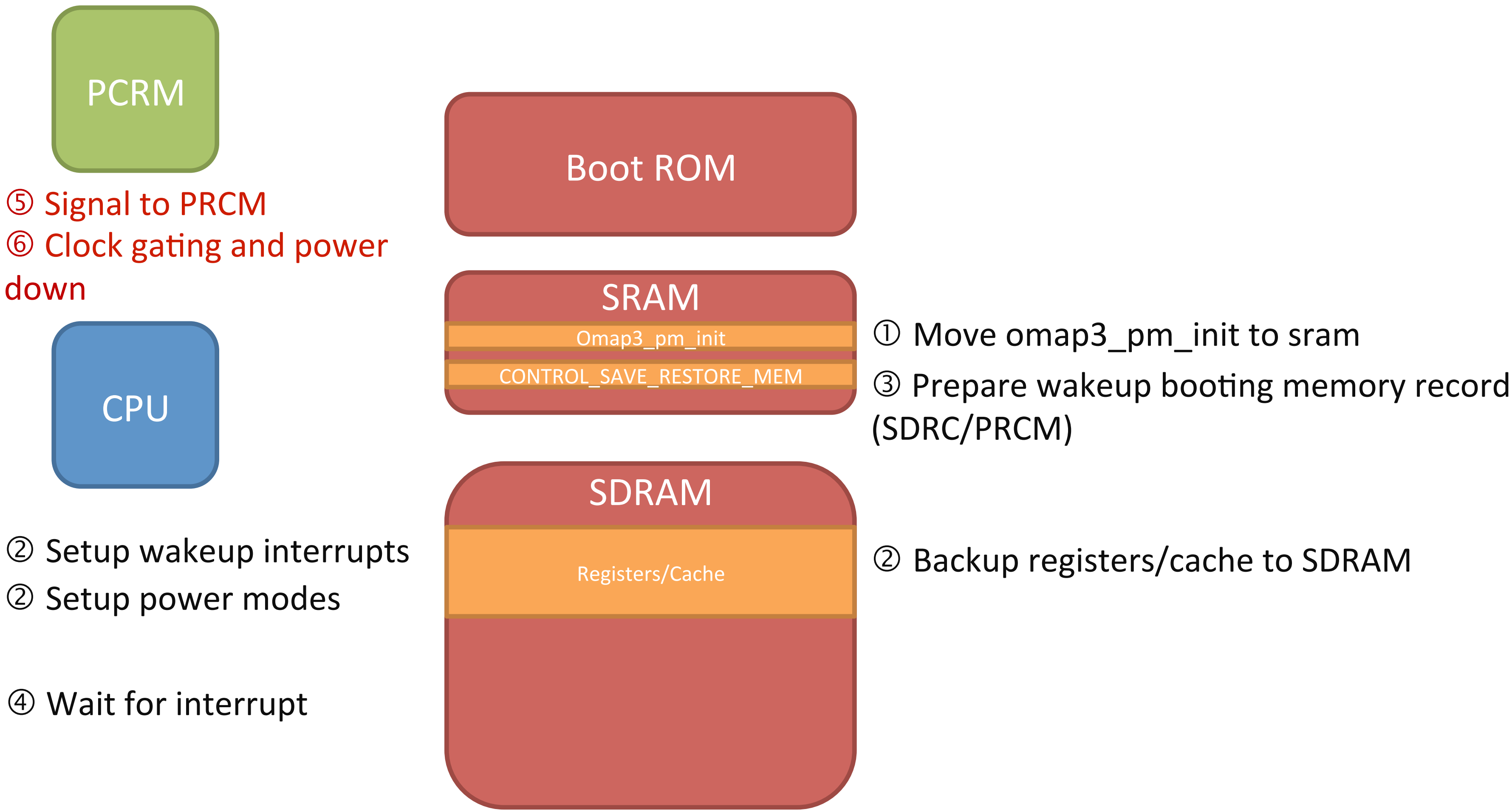
  2,479msec → 125msec

- Boot up time also reduced: Over 1min. → 20sec.

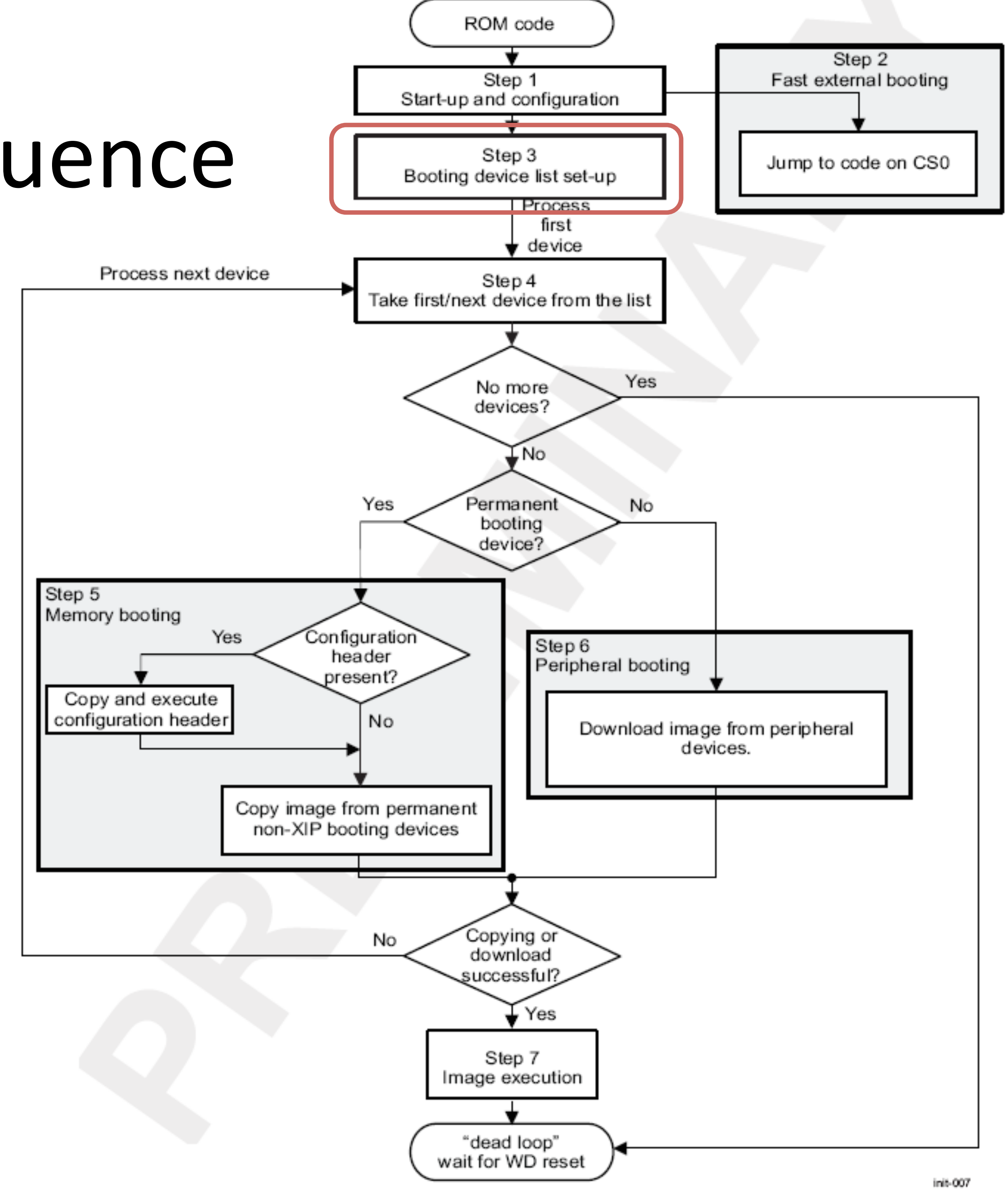
Suspend to RAM

OMAP3430

- Enter dormant mode

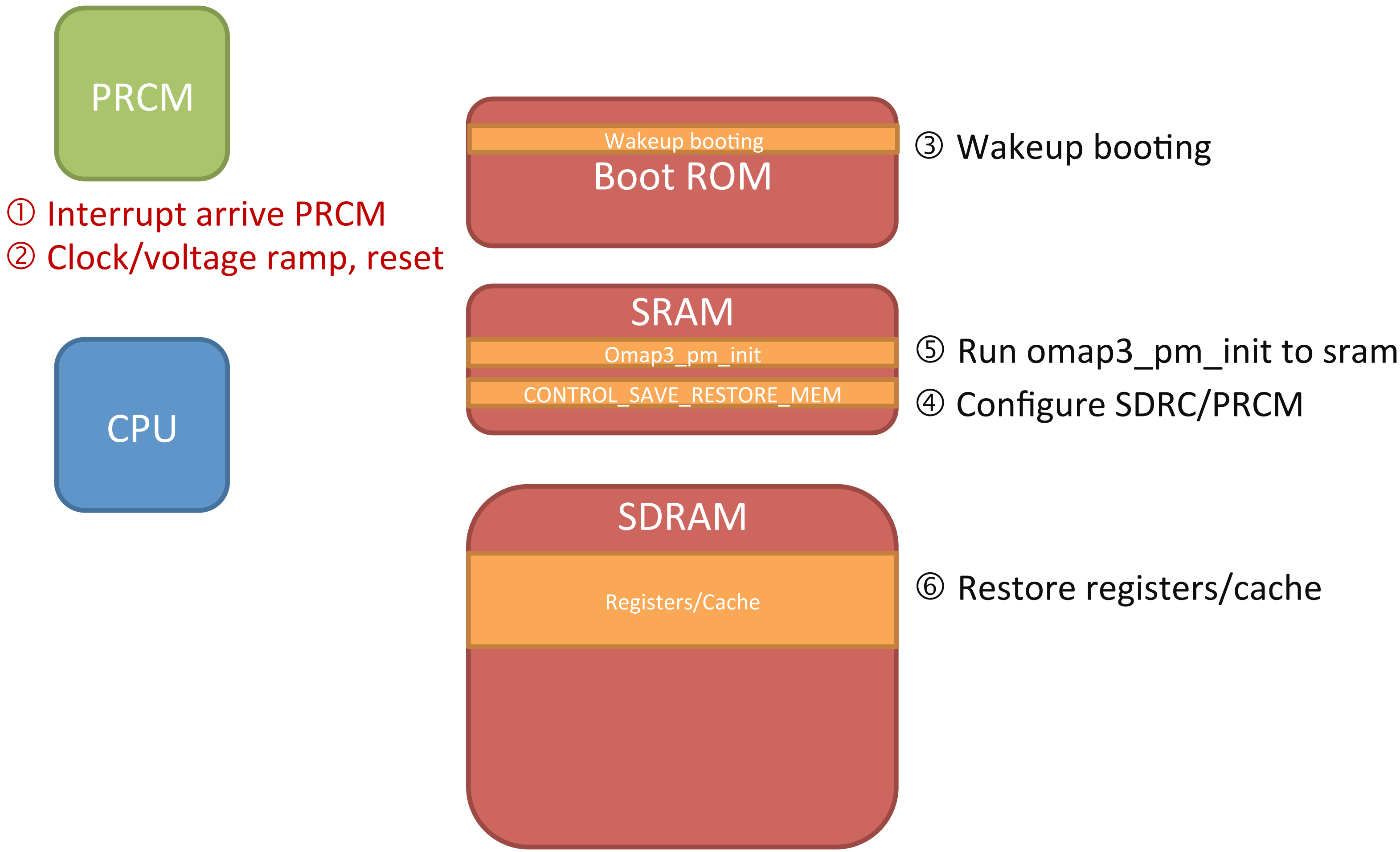


Booting sequence



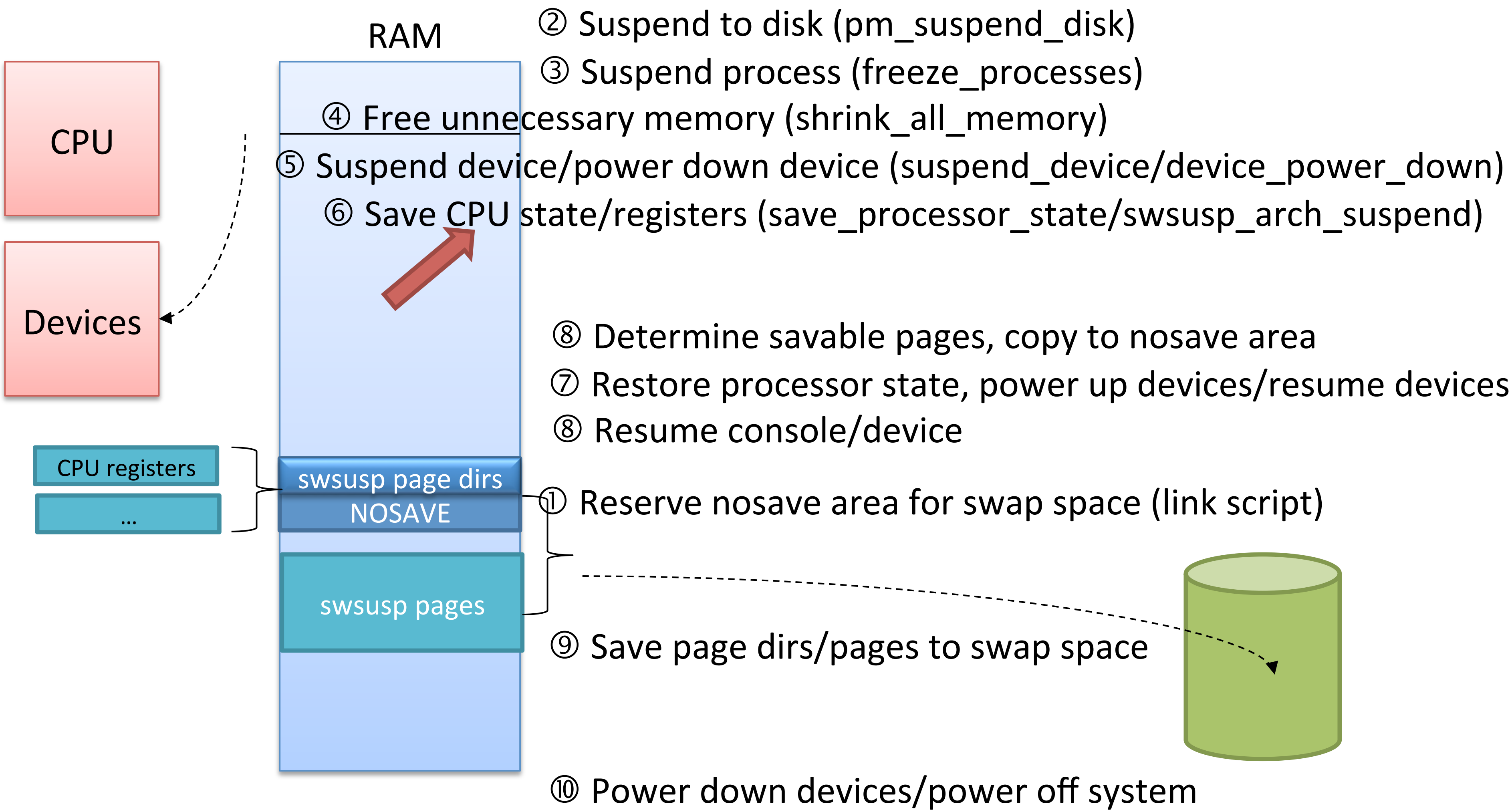
OMAP3430

- Exit dormant mode



Suspend to Disk

Suspend Flow



Resume Flow

① CPU booting/bootloader/kernel loading/decompress kernel/start_kernel

② late_initcall(software_resume)

③ Check signature and header of swsusp image

④ Allocate memory for swsusp image

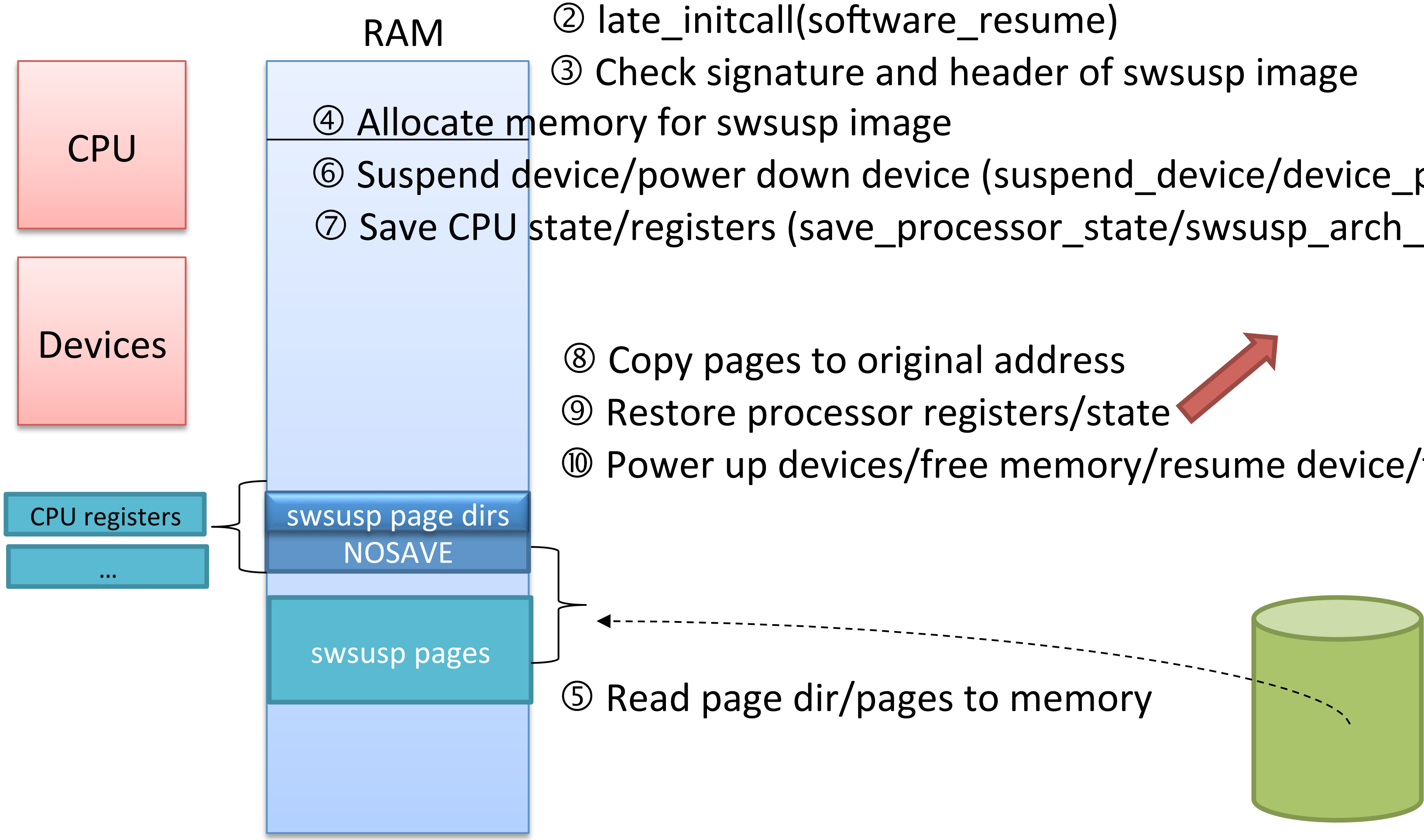
⑥ Suspend device/power down device (suspend_device/device_power_down)

⑦ Save CPU state/registers (save_processor_state/swsusp_arch_suspend)

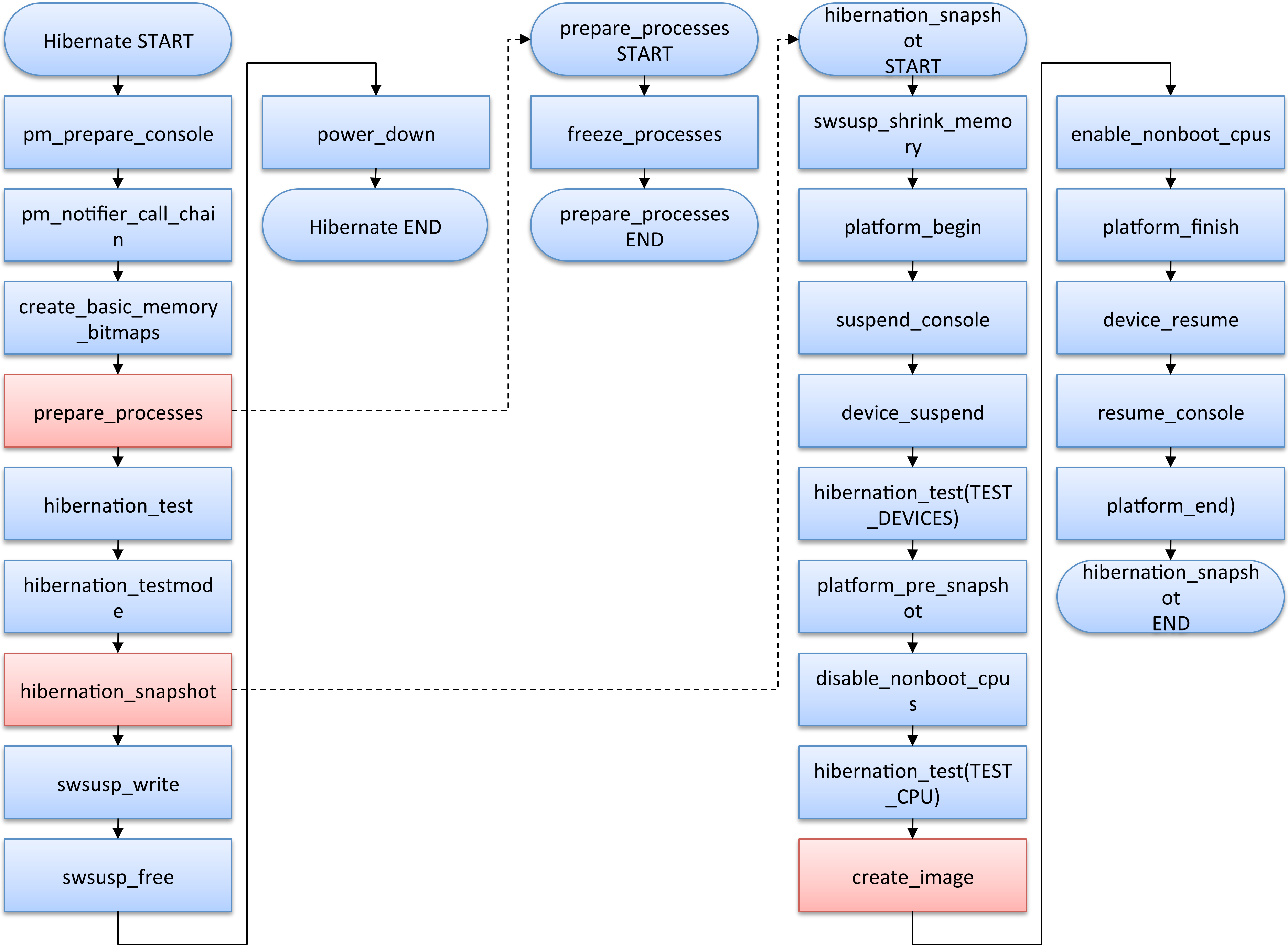
⑧ Copy pages to original address

⑨ Restore processor registers/state

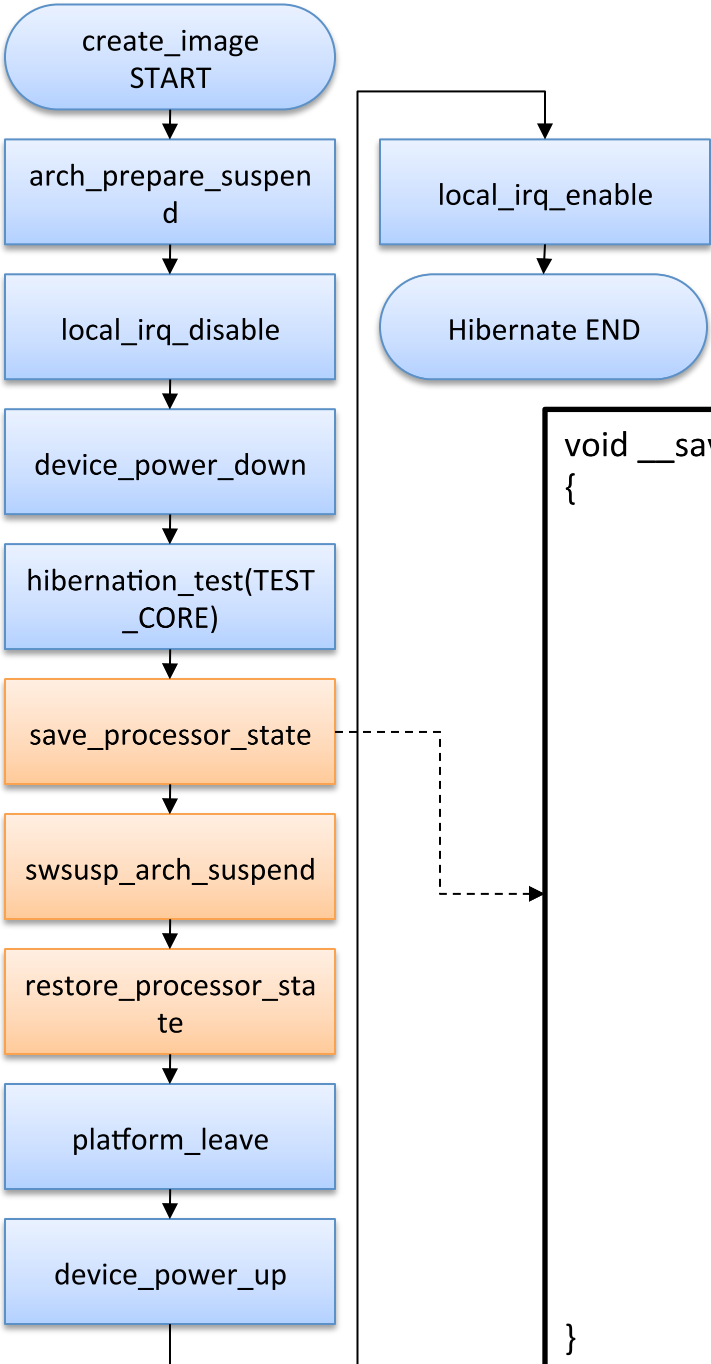
⑩ Power up devices/free memory/resume device/thaw processe



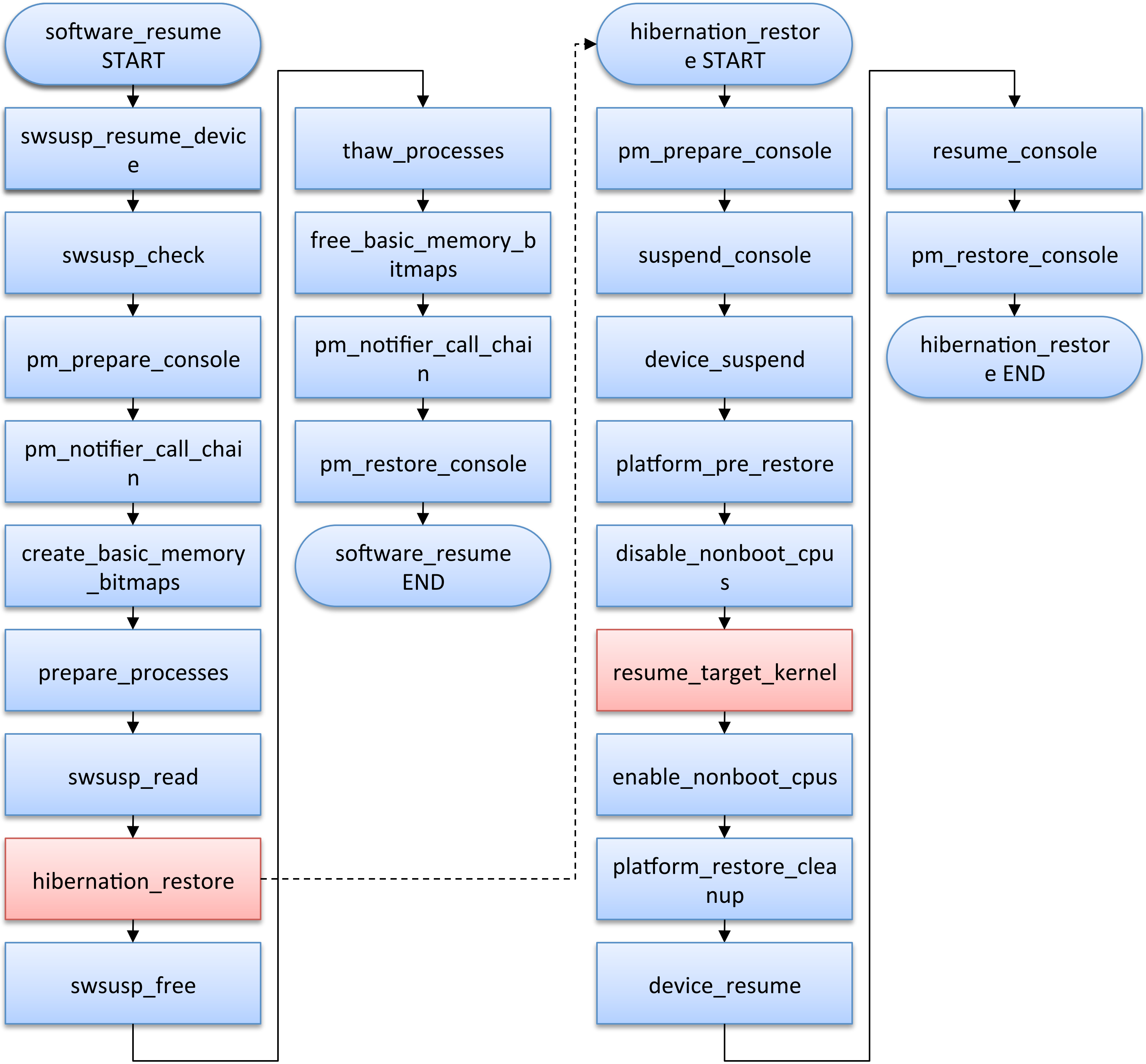
⑤ Read page dir/pages to memory



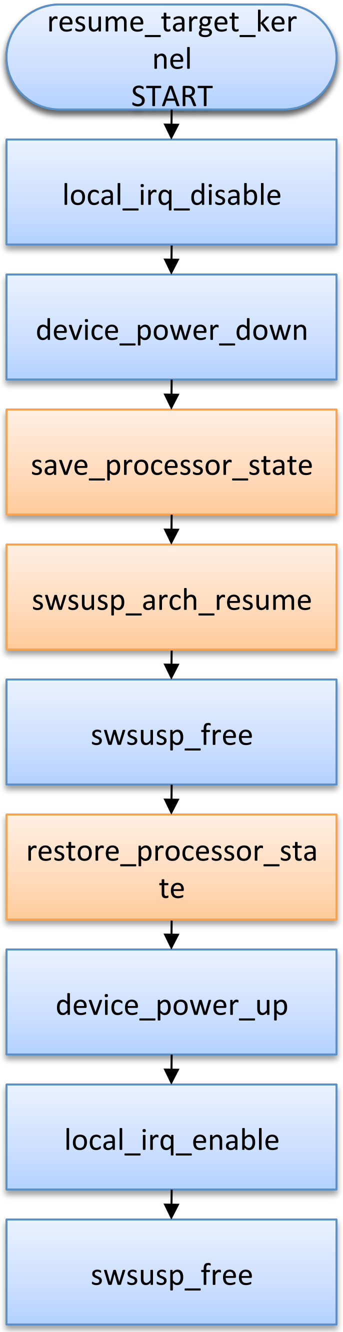
Hibernate START
pm_prepare_console
pm_notifier_call_chain
create_basic_memory_bitmaps
prepare_processes
hibernation_test
hibernation_testmode
hibernation_snapshot
swsusp_write
swsusp_free
power_down
Hibernate END
prepare_processes START
freeze_processes
prepare_processes END
hibernation_snapshot START
swsusp_shrink_memory
platform_begin
suspend_console
device_suspend
hibernation_test(TEST_DEVICES)
platform_pre_snapshot
disable_nonboot_cpus
hibernation_test(TEST_CPU)
create_image
enable_nonboot_cpus
platform_finish
device_resume
resume_console
platform_end)
hibernation_snapshot END




```
void __save_processor_state(struct saved_context *ctxt)
{
        /* save preempt state and disable it */
        preempt_disable();

        /* save coprocessor 15 registers */
        asm volatile ("mrc p15, 0, %0, c1, c0, 0" : "=r" (ctxt->CR));
        asm volatile ("mrc p15, 0, %0, c3, c0, 0" : "=r" (ctxt->DACR));
        asm volatile ("mrc p15, 0, %0, c5, c0, 0" : "=r" (ctxt->D_FSR));
        asm volatile ("mrc p15, 0, %0, c5, c0, 1" : "=r" (ctxt->I_FSR));
        asm volatile ("mrc p15, 0, %0, c6, c0, 0" : "=r" (ctxt->FAR));
        asm volatile ("mrc p15, 0, %0, c9, c0, 0" : "=r" (ctxt->D_CLR));
        asm volatile ("mrc p15, 0, %0, c9, c0, 1" : "=r" (ctxt->I_CLR));
        asm volatile ("mrc p15, 0, %0, c9, c1, 0" : "=r" (ctxt->D_TCMRR));
        asm volatile ("mrc p15, 0, %0, c9, c1, 1" : "=r" (ctxt->I_TCMRR));
        asm volatile ("mrc p15, 0, %0, c10, c0, 0" : "=r" (ctxt->TLBLR));
        asm volatile ("mrc p15, 0, %0, c13, c0, 0" : "=r" (ctxt->FCSE));
        asm volatile ("mrc p15, 0, %0, c13, c0, 1" : "=r" (ctxt->CID));
        asm volatile ("mrc p15, 0, %0, c2, c0, 0" : "=r" (ctxt->TTBR));

}
```

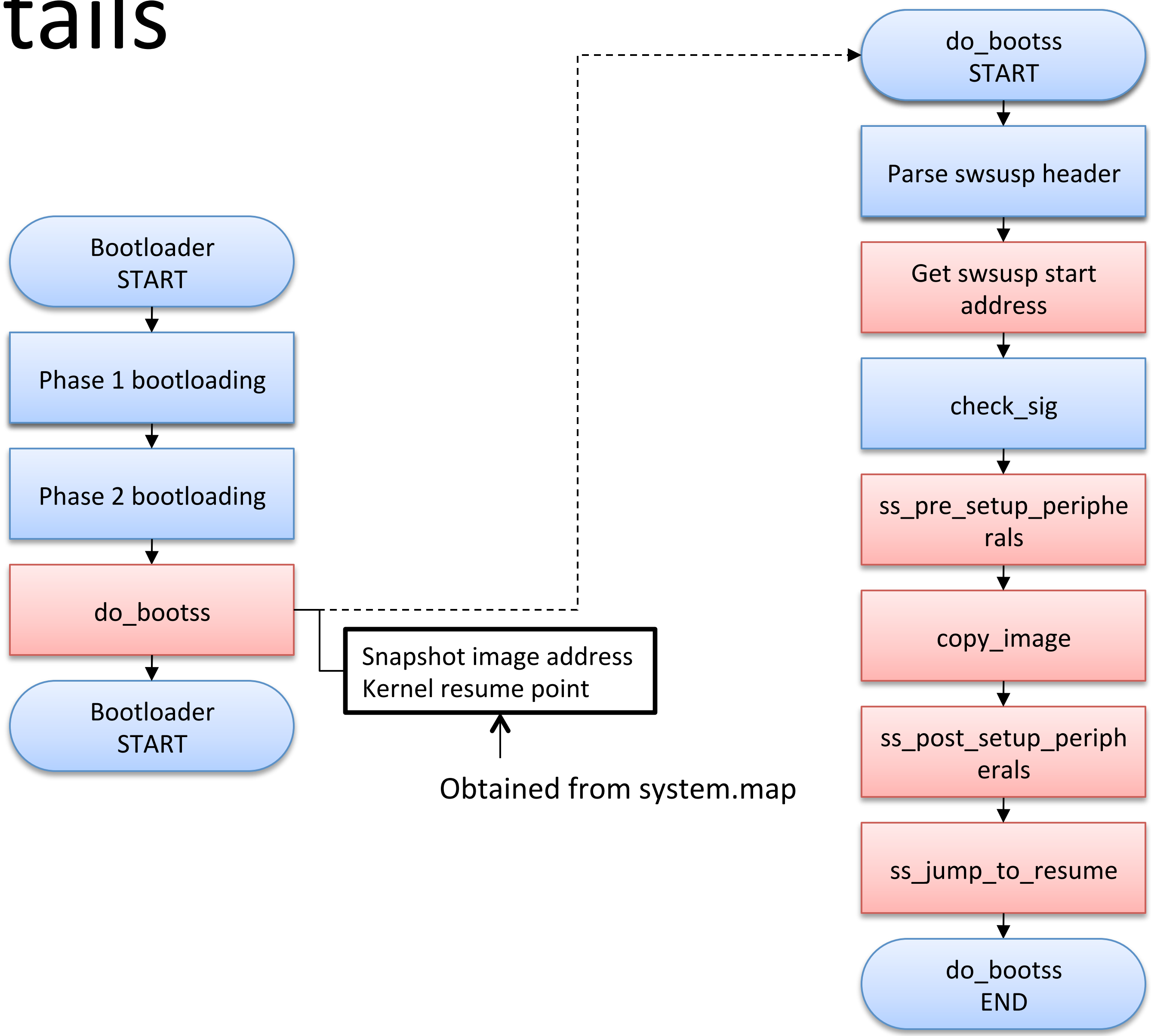


software_resume START
swsusp_resume_device
swsusp_check
pm_prepare_console
pm_notifier_call_chain
create_basic_memory_bitmaps
prepare_processes
swsusp_read
hibernation_restore
swsusp_free
thaw_processes
free_basic_memory_bitmaps
pm_notifier_call_chain
pm_restore_console
software_resume END
hibernation_restore START
pm_prepare_console
suspend_console
device_suspend
platform_pre_restore
disable_nonboot_cpus
resume_target_kernel
enable_nonboot_cpus
platform_restore_cleanup
device_resume
resume_console
pm_restore_console
hibernation_restore END


resume_target_kernel START
local_irq_disable
device_power_down
save_processor_state
swsusp_arch_resume
swsusp_free
restore_processor_state
device_power_up
local_irq_enable
swsusp_free

Snapshot Booting Implementation

Details



Bootloader START
Phase 1 bootloading
Phase 2 bootloading
do_bootss
Bootloader START
Snapshot image address
Kernel resume point
Obtained from system.map
do_bootss START
Parse swsusp header
Get swsusp start address
check_sig
ss_pre_setup_peripherals
copy_image
ss_post_setup_peripherals
ss_jump_to_resume
do_bootss END

Preserving hibernation system image