EE382N-4
Advanced Microcontroller Systems

Boot Loaders and Device Trees

Mark McDermott

Boot Terminology

- **Loader:**
  - Program that moves bits from disk (usually) to memory and then transfers CPU control to the newly "loaded" bits (executable).
- **Boot-loader / Bootstrap:**
  - Program that loads the "first program" (the kernel).
- **Boot PROM / PROM Monitor / BIOS:**
  - Persistent code that is "already loaded" on power-up.
- **Boot Manager:**
  - Program that lets you choose the "first program" to load.

Linux Open Source Boot-Loaders

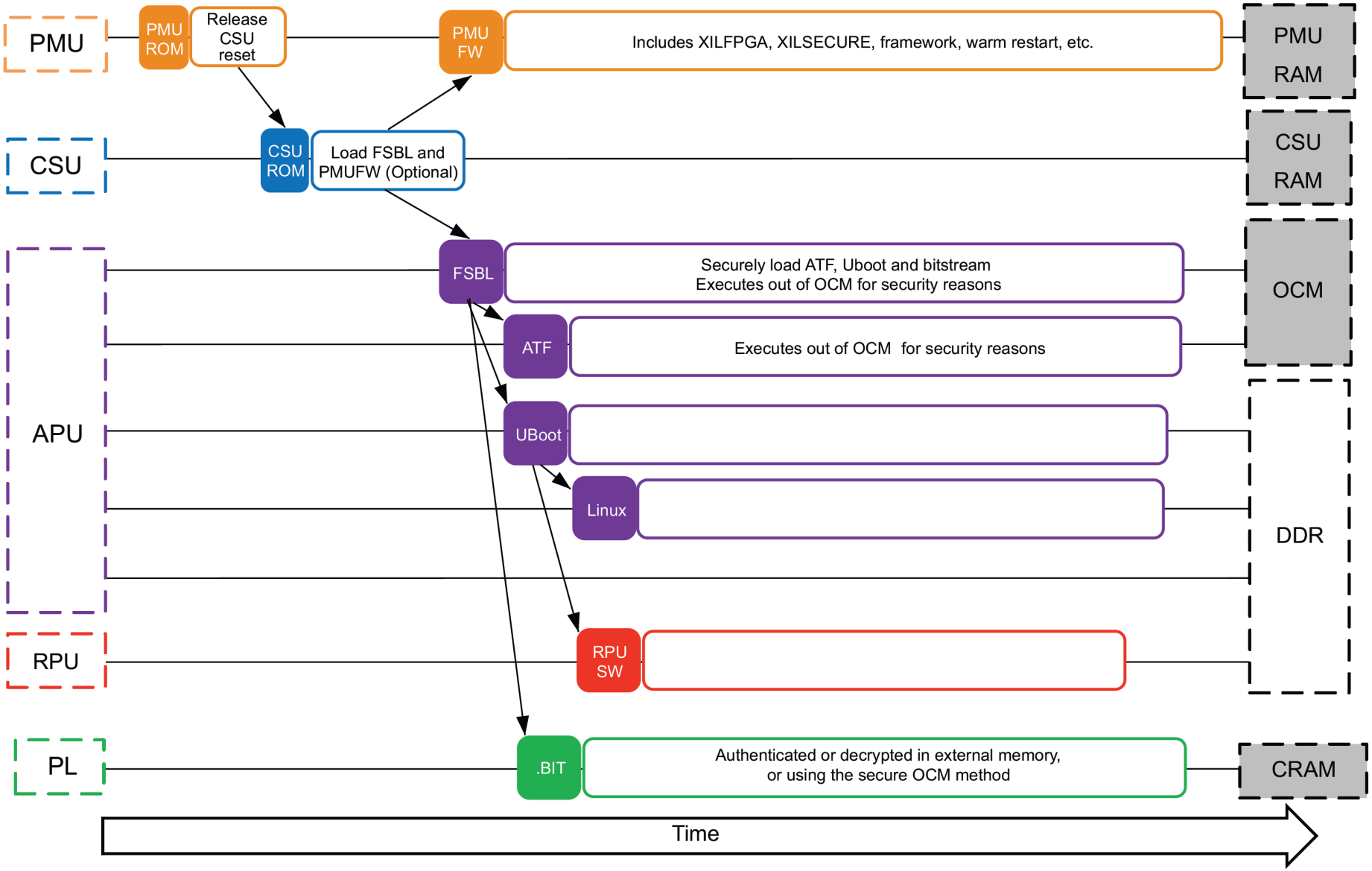
Boot-Loader	Monitor	Description	X86	ARM
LILO	No	The main disk bootloader for Linux	X	
GRUB	No	GNU's successor to LILO	X	
ROLO	No	Loads Linux from ROM without a BIOS	X	
LinuxBIOS	No	Linux-based BIOS replacement	X	
U-Boot	Yes	Universal loader	X	X
RedBOOT	Yes	eCos-based loader	X	X

Boot-loaders

- **The boot-loader is a piece of code responsible for**
  - Basic hardware initialization
  - Loading of an application binary, usually an operating system kernel, from flash storage, from the network, or from another type of non-volatile storage.
  - Possibly decompression of the application binary
  - Execution of the application

- **Besides these basic functions, most boot loaders provide a shell with various commands implementing different operations.**
  - Loading of data from storage or network, memory inspection, hardware diagnostics and testing, etc.

Zynq UltraScale Boot-up Flow



UltraScale96 Board U-boot prompt

```
Xilinx Zynq MP First Stage Boot Loader
Release 2022.2   Oct  7 2022  -  04:56:16
NOTICE:  BL31: v2.6(release):xlnx_rebase_v2.6_2022.1_update3-18-g0897efd45
NOTICE:  BL31: Built : 03:55:03, Sep  9 2022

U-Boot 2022.01 (Sep 20 2022 - 06:35:33 +0000)

CPU:   ZynqMP
Silicon: v3
Board: Xilinx ZynqMP
DRAM:  2 GiB
PMUFW:  v1.1
PMUFW no permission to change config object
EL Level:       EL2
Chip ID:        zu3eg
NAND:  0 MiB
MMC:   mmc@ff160000: 0, mmc@ff170000: 1
In:    serial
Out:   serial
Err:   serial
Bootmode: SD_MODE
Reset reason:   SOFT
Net:   No ethernet found.
scanning bus for devices...
starting USB...
Bus usb@fe300000: probe failed, error -2
No working controllers found
Hit any key to stop autoboot:  0
```

U-Boot Commands

```
ZynqMP> bdinfo
arch_number = 0x00000000
boot_params = 0x00000000
DRAM bank   = 0x00000000
-> start    = 0x00000000
-> size     = 0x7FF00000
current eth = unknown
ip_addr     = <NULL>
baudrate    = 115200 bps
TLB addr    = 0x7FEE0000
relocaddr   = 0x7FE1A000
reloc off   = 0x77E1A000
irq_sp      = 0x7DD99DD0
sp start    = 0x7DD99DD0
ARM frequency = 100 MHz
DSP frequency = 0 MHz
DDR frequency = 0 MHz
Early malloc usage: dd8 / 8000
fdt_blob = 000000007fe830c0
```

- **Information Commands**
  - bdinfo – list board information on console
  - coninfo – list console information
  - flinfo – list flash memory information
  - iminfo – list application image information
  - help – lists commands , for help on specific command type help command

- **Commands to manipulate environment variables:**
  - printenv - shows all variables
  - printenv <variable-name> - shows the value of a variable
  - setenv <variable-name> <variable-value> - Changes the value of a variable, only in RAM
  - editenv <variable-name> edits the value of a variable, only in RAM
  - saveenv - saves the current state of the environment to flash

U-Boot Commands

- **Memory Commands**
  - base – print or set base address
  - crc32 – calculate the crc32 checksum over an address range
  - cmp – compare two memory ranges (cmp staddr1 staddr2 size (hex))
  - cp – copy memory (cp source targer size (hex))
  - md – display memory (md (.b,.w,.l) addr size (hex))
  - mm – modify memory (mm (.b,.w,.l) addr (hex)) will prompt for new value
  - mtest – simple memory test (mtest start end pattern (hex))
  - mw – memory write (mw [.b, .w, .l] address value [count] (hex))
  - nm – memory modify (nm [.b, .w, .l] address (hex))
  - loop – infinite loop on address range (loop [.b, .w, .l] address number_of_objects (hex))

U-Boot Commands

- **Flash Memory Commands**
  - cp – copy memory (cp [.b, .w, .l] source target count (hex))
  - flinfo – print Flash memory information
  - erase – erase flash memory (see help for options)
  - protect – enable or disable flash protection
  - mtdparts – define a Linux compatible Memory Technology Device (MTD) partition scheme (see manual for more info on this command)

U-Boot Commands

- **Execution Control Commands**
  - autoscr - run script from memory
  - bootm - boot application image from memory
  - go - start application at address 'addr‘

- **Download Commands**
  - bootp - boot image via network using BOOTP/TFTP protocol
  - dhcp - invoke DHCP client to obtain IP/boot params
  - loadb - load binary file over serial line (kermit mode)
  - loads - load S-Record file over serial line
  - rarpboot- boot image via network using RARP/TFTP protocol
  - tftpboot- boot image via network using TFTP protocol
  - fatload, loads a file from a FAT filesystem to RAM
    - also fatls and fatinfo

Important U-Boot env variables

- bootcmd; specifies the commands that U-Boot will automatically execute at boot time after a configurable delay (bootdelay), if the process is not interrupted
- bootargs; contains the arguments passed to the Linux kernel
- serverip; the IP address of the server that U-Boot will contact for network related commands
- ipaddr; the IP address that U-Boot will use
- netmask; the network mask to contact the server
- ethaddr; the MAC address, can only be set once
- autostart; if yes, U-Boot starts automatically an image that has been loaded into memory
- filesize; the size of the latest copy to memory (from tftp, fatload, nand read...)

Scripts in environment variables

- **Environment variables can contain small scripts, to execute several commands and test the results of commands.**
  - Useful to automate booting or upgrade processes
  - Several commands can be chained using the ; operator
  - Tests can be done using
    if command ; then … ; else … ; fi
- **Scripts are executed using run <variable-name>**
  - You can reference other variables using ${variable-name}
- **Example (boot.cmd)**

```
kernel_image=image.ub
devicetree_image=system.dtb
fatload mmc 0 0x1000000 system.bit
fpga loadb 0 0x1000000 0x54f91a
fatload mmc 0 0x3000000 ${kernel_image}
fatload mmc 0 0x2A00000 ${devicetree_image}
bootm 0x3000000 - 0x2A00000
```

- **Build .scr file using:**

```
mkimage -A arm -T script -O linux -d boot.cmd boot.scr
```

Transferring files to the target

- U-Boot is mostly used to load and boot a kernel image, but it also provides the ability to change the kernel image and the root filesystem stored in flash.

- Files must be exchanged between the target and the development workstation. This is possible via:
  - the network if the target has an Ethernet connection, and U-Boot contains a driver for the Ethernet chip. This is the fastest and most efficient solution.
  - a USB key, if U-Boot supports the USB controller of your platform
  - an SD or microSD card, if U-Boot supports the MMC controller of your platform
  - the serial port

TFTP

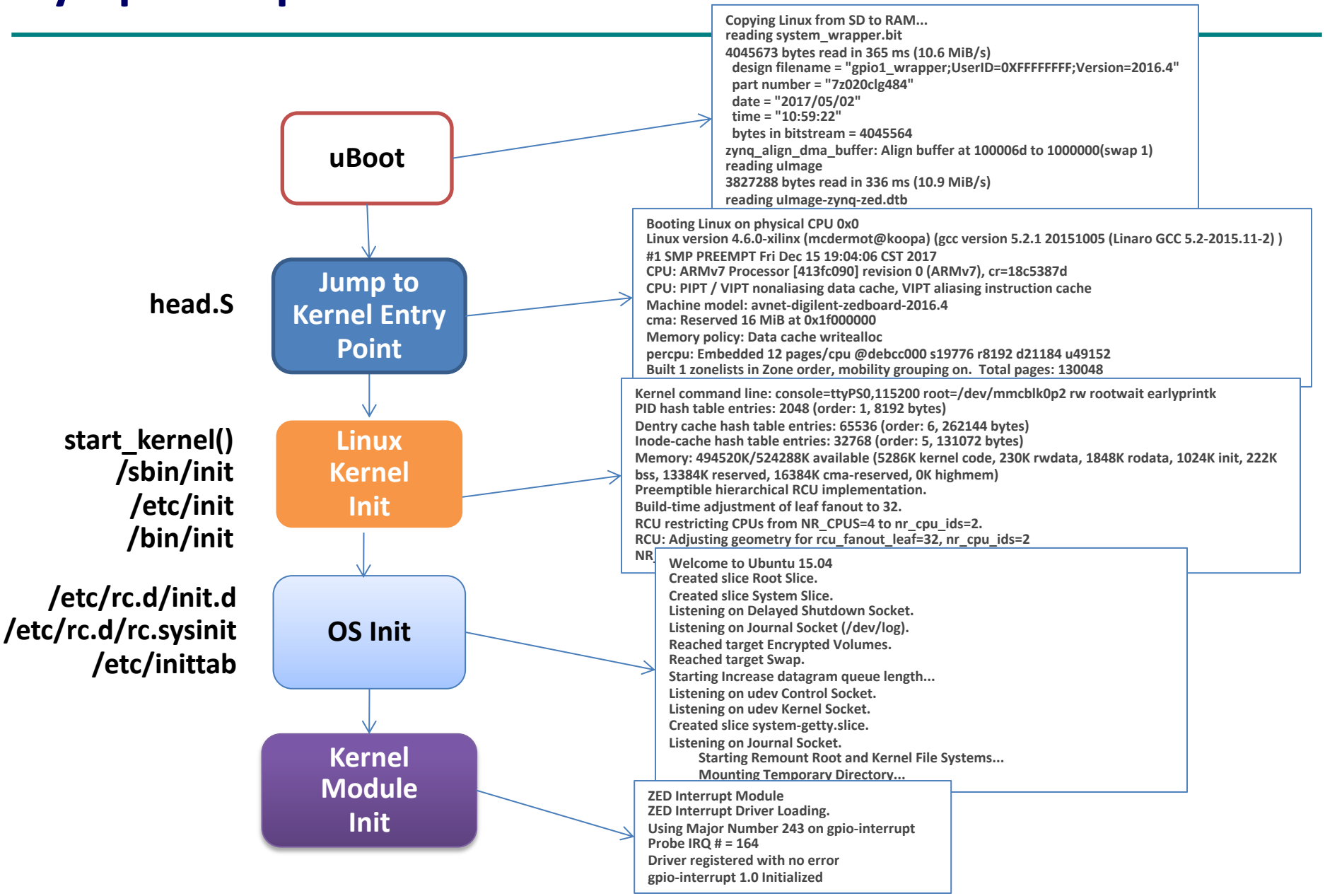
- **Network transfer from the development workstation to U-Boot on the target takes place through TFTP**
  - Trivial File Transfer Protocol
  - Somewhat like FTP, but without authentication and over UDP

- **A TFTP server is needed on the development workstation**
  - sudo apt-get install tftpd-hpa
  - All files in /var/lib/tftpboot are then visible through TFTP
  - A TFTP client is available in the tftp-hpa package, for testing

- **A TFTP client is integrated into U-Boot**
  - Configure the ipaddr and serverip environment variables
  - Use tftp <address> <filename> to load a file

Booting Embedded Linux

Zynq Bootup flow



Kernel Entry Point: head.S

- **Performs the following functions:**
  - Checks for valid processor and architecture
  - Creates the initial page table entries
  - Enables the processors memory management unit
  - Establishes limited error detection and reporting
  - Jumps to the start of kernel proper, main.c

```
Linux version 4.6.0-xilinx (mcdermot@koopa) (gcc version 5.2.1
20151005 (Linaro GCC 5.2-2015.11-2) ) #1 SMP PREEMPT Fri Dec 15
19:04:06 CST 2017
CPU: ARMv7 Processor [413fc090] revision 0 (ARMv7), cr=18c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction
cache
Machine model: avnet-digilent-zedboard-2016.4
   PID hash table entries: 2048 (order: 1, 8192 bytes)
   Dentry cache hash table entries: 65536 (order: 6, 262144 bytes)
   Inode-cache hash table entries: 32768 (order: 5, 131072 bytes)
Virtual kernel memory layout:
  vector  : 0xffff0000 - 0xffff1000   (   4 kB)
  fixmap  : 0xffc00000 - 0xfff00000   (3072 kB)
  vmalloc : 0xe0800000 - 0xff800000   ( 496 MB)
  lowmem  : 0xc0000000 - 0xe0000000   ( 512 MB)
  pkmap   : 0xbfe00000 - 0xc0000000   (   2 MB)
  modules : 0xbf000000 - 0xbfe00000   (  14 MB)
```

```
/*
 * Kernel startup entry point.
 * ---------------------------
 *
 * This is normally called from the decompressor code.  The requirements
 * are: MMU = off, D-cache = off, I-cache = dont care, r0 = 0,
 * r1 = machine nr.
 *
 * This code is mostly position independent, so if you link the kernel at
 * 0xc0008000, you call this at __pa(0xc0008000).
 *
 * See linux/arch/arm/tools/mach-types for the complete list of machine
 * numbers for r1.
 *
 * We're trying to keep crap to a minimum; DO NOT add any machine specific
 * crap here - that's what the boot loader (or in extreme, well justified
 * circumstances, zImage) is for.
 */
    __INIT
    .type   stext, %function
ENTRY(stext)
    msr     cpsr_c, #PSR_F_BIT | PSR_I_BIT | MODE_SVC       @ ensure svc mode
                                                            @ and irqs disabled
    bl      __lookup_processor_type                         @ r5=procinfo r9=cpuid
    movs    r10, r5                                         @ invalid processor (r5=0)?
    beq     __error_p                                       @ yes, error 'p'
    bl      __lookup_machine_type                           @ r5=machinfo
    movs    r8, r5                                          @ invalid machine (r5=0)?
    beq     __error_a                                       @ yes, error 'a'
    bl      __create_page_tables

    /*
     * The following calls CPU specific code in a position independent
     * manner.  See arch/arm/mm/proc-*.S for details.  r10 = base of
     * xxx_proc_info structure selected by __lookup_machine_type
     * above.  On return, the CPU will be ready for the MMU to be
     * turned on, and r0 will hold the CPU control register value.
     */
    ldr     r13, __switch_data                              @ address to jump to after
                                                            @ mmu has been enabled
    adr     lr, __enable_mmu                                @ return (PIC) address
    add     pc, r10, #PROCINFO_INITFUNC
```

Kernel Entry Point: …/init/main.c

- **Performs the following functions:**
  - This code starts up the kernel.
  - The "command line" from the boot loader is parsed and executed.
  - The cache is configured at this time
  - The scheduler initialized while preemption is turned off.
  - The various timers and interrupts are then initialized.

```
Kernel command line:
ip=128.83.52.221:$GIPADD:$GIPADD:$NETMASK::eth0:off mem=64M
console=ttyS0:230400 console=tty  root=/dev/mtdblock1 rootfstype=jffs2
PID hash table entries: 512 (order: 9, 8192 bytes)
Console: colour dummy device 80x30
Serial: Console IMX rounded baud rate from 115193 to 115200
Dentry cache hash table entries: 16384 (order: 4, 65536 bytes)
Inode-cache hash table entries: 8192 (order: 3, 32768 bytes)
Memory: 64MB = 64MB total
Memory: 62208KB available (2260K code, 247K data, 96K init)
Mount-cache hash table entries: 512
CPU: Testing write buffer coherency: ok
NET: Registered protocol family 16
Initializing system clocks.........CSCR: 0x17000e07
```

```
asmlinkage void __init start_kernel(void)
/*
 * Interrupts are still disabled. Do necessary setups, then
 * enable them
 */
    lock_kernel();
    page_address_init();
    printk(KERN_NOTICE);
    printk(linux_banner);
    setup_arch(&command_line);
    setup_per_cpu_areas();

    /*
     * Mark the boot cpu "online" so that it can call console drivers in printk() and can
      access its per-cpu storage.
     */
    smp_prepare_boot_cpu();

    /*
     * Set up the scheduler prior starting any interrupts (such as the timer interrupt). Full
      * topology setup happens at smp_init() time - but meanwhile we still have a
      * functioning scheduler.
     */
    sched_init();
    /*
     * Disable preemption - early bootup scheduling is extremely
     * fragile until we cpu_idle() for the first time.
     */
    preempt_disable();
    build_all_zonelists();
    page_alloc_init();
    printk(KERN_NOTICE "Kernel command line: %s\n", saved_command_line);
    parse_early_param();
    parse_args("Booting kernel", command_line, __start___param,
            __stop___param - __start___param,
            &unknown_bootoption);
    sort_main_extable();
    trap_init();
    rcu_init();
    init_IRQ();
    pidhash_init();
    init_timers();
    hrtimers_init();
    softirq_init();
    time_init();
```

/sbin/init

- Ancestor of all processes (except idle/swapper process).
- Init parses the /etc/inittab file to determine the specifics of what programs to run and at what level.
  - 0: used to halt the system. The system performs an init 0 command and the system is halted.
  - 1: Puts the system into single-user mode.
  - 2: Puts the system into a multiuser mode but does not support networking.
  - 3: Puts the system into the standard full multiuser mode but does not automatically start X-windows.
  - 4: User defined.
  - 5: Puts the system into standard multiuser mode with a graphical (X-based) login.
  - 6: Shutdown DO NOT use inittab to set this
- Starts the /etc/rc.d/rc script with the appropriate run level.
  - The rc script executes all of the scripts pointed to by the symboblic links contained in the directory for that run level.
  - For example, if the run level is 3, the scripts pointed to by the links in /etc/rc.d/rc3.d are run.

/etc/inittab

```
# /etc/inittab
#
# Copyright (C) 2001 Erik Andersen <andersen@codepoet.org>
#
# Note: BusyBox init doesn't support runlevels.
# The runlevels field is completely ignored by BusyBox init.
# If you want runlevels, use sysvinit.
#
# Format for each entry: <id>:<runlevels>:<action>:<process>
#
# id       == tty to run on, or empty for /dev/console
# runlevels == ignored
# action    == one of sysinit, respawn, askfirst, wait, and once
# process   == program to run

# Startup the system
#null::sysinit:/bin/mount -o remount,rw /
null::sysinit:/bin/mount -t proc proc /proc
null::sysinit:/bin/mount -a
null::sysinit:/bin/hostname -F /etc/hostname
null::sysinit:/sbin/ifconfig lo 127.0.0.1 up
null::sysinit:/sbin/route add -net 127.0.0.0 netmask 255.0.0.0 lo
# now run any rc scripts
::sysinit:/etc/init.d/rcS

# Set up a couple of getty's
#tty1::respawn:/sbin/getty 38400 tty1
#tty2::respawn:/sbin/getty 38400 tty2

# Put a getty on the serial port
#ttyAM0::respawn:/sbin/getty -L ttyAM0 38400 vt100

# Put a getty on the serial port
#ttyAM0::respawn:/sbin/getty -L ttyAM0 38400 vt100

# Put a shell on the default console
ttyS0::respawn:/bin/sh

# Logging junk
#null::sysinit:/bin/touch /var/log/messages
#null::respawn:/sbin/syslogd -n -m 0
#null::respawn:/sbin/klogd -n
#tty3::respawn:/usr/bin/tail -f /var/log/messages

# Stuff to do for the 3-finger salute
::ctrlaltdel:/sbin/reboot

# Stuff to do before rebooting
null::shutdown:/usr/bin/killall klogd
null::shutdown:/usr/bin/killall syslogd
null::shutdown:/bin/umount -a -r
null::shutdown:/sbin/swapoff -a
```

Normally run from the rc.sysinit script

/etc/rc.d/rc3.d

- All the files here are only symbolic links to the actual scripts that exist in /etc/rc.d/init.d.
- The system first runs the scripts whose names start with K to kill the associated processes → /etc/rc.d/init.d/<command> stop
- The system runs the scripts whose names start with S to start the processes → /etc/rc.d/init.d/<command> start
- Changing a K name to start with S (e.g., K20nfs → S20nfs) makes Linux start the process rather than kill it.

/etc/init.d

- Run once at boot time
- Sets the path and the hostname and checking whether networking is activated.
- Mounts the /proc filesystem
- Sets the kernel parameters
- Sets the system clock
- Loading keymaps and fonts
- Start swapping
- Initializing the USB controller along with the attached devices.
- Checks the root filesystem for consistency (if applicable)
- Remounts the root filesystem as read-write.
- Loads kernel modules as appropriate.

Shutdown

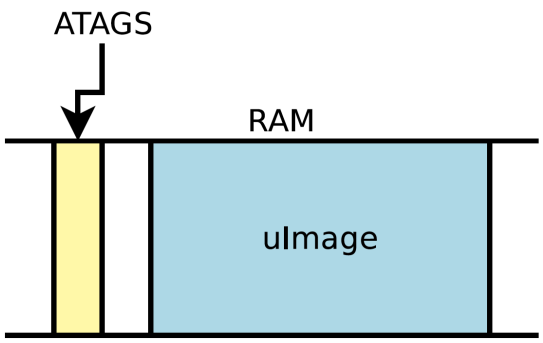
- Use /sbin/shutdown to avoid data loss and filesystem corruption.
- Shutdown inhibits login, asks init to send SIGTERM to all processes, then SIGKILL
  - Clears out all buffered writes using sync command. Not guaranteed to work due to the non-deterministic nature of disk sub-systems.
- Low-level commands: halt, reboot, poweroff.
  - Use -h, -r or -p options to shutdown instead.
- Ctrl-Alt-Delete "Vulcan neck pinch":
  - defined by a line in /etc/inittab
  - ca::ctrlaltdel:/sbin/shutdown -t3 -r now.

Device Trees

User perspective: before the Device Tree

- The kernel contains the entire description of the hardware.
- The bootloader loads a single binary, the kernel image, and executes it.
  - uImage or zImage
- The bootloader prepares some additional information, called ATAGS, which address is passed to the kernel through register r2
  - Contains information such as memory size and location, kernel command line, etc.
  - The bootloader tells the kernel on which board it is being booted through a machine type integer, passed in register r1.
- U-Boot command:

```
bootm <kernel img addr>
```




```
r1 = <machine type>
r2 = <pointer to ATAGS>
```

User perspective: booting with a Device Tree

- The kernel no longer contains the description of the hardware, it is located in a separate binary: the device tree blob (DTB)
- The bootloader loads two binaries: the kernel image and the DTB
  - Kernel image is: `uImage.ub` or `Image.gz`
  - DTB image is: `system.dtb`
- The bootloader passes the DTB address through r2. It is supposed to adjust the DTB with memory information, kernel command line, and potentially other info.
- Machine type has been deprecated
- U-Boot command:

```
bootm <kernel img addr> - <dtb addr>
```

Basic Device Tree Syntax

Node name

Unit address

Property name

Property value

Properties of node@0

```
/ {
    node@0 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        a-byte-data-property = [0x01 0x23 0x34 0x56];

        child-node@0 {
            first-child-property;
            second-child-property = <1>;
            a-reference-to-something = <&node1>;
        };

        child-node@1 {
        };
    };

    node1: node@1 {
        an-empty-property;
        a-cell-property = <1 2 3 4>;

        child-node@0 {
        };
    };
};
```

Bytestring

A phandle
(reference to another node)

Label

Four cells (32 bits values)

From source to binary

- On ARM, all Device Tree Source files (DTS) are for now located in `arch/arm64/boot/dts/Xilinx/`
  - .dts files for board-level definitions
  - .dtsi files for included files, generally containing SoC-level definitions
- A tool, the Device Tree Compiler compiles the source into a binary form.
  - Source code located in …/scripts/dtc
  - Need to source Petalinux shell script to use dtc
- The Device Tree Blob is produced by the Device Tree Compiler (dtc) and is the binary that gets loaded by the bootloader and parsed by the kernel at boot time.

```
dtc -I dts -O dtb -o system.dtb system.dts
```

- To decompile a dtb back to source use the following command:

```
dtc -I dtb -O dts -o system.dts system.dtb
```

A simple example, DT side

```
auart0: serial@8006a000 {
```

Defines the "programming model" for the device. Allows the operating system to identify the corresponding device driver.

```
compatible = "fsl,imx28-auart", "fsl,imx23-auart";
```

Address and length of the register area.

```
reg = <0x8006a000 0x2000>;
```

Interrupt number

```
interrupts = <112>;
```

DMA engine and channels, with names.

```
dmas = <&dma_apbx 8>, <&dma_apbx 9>;
dma-names = "rx", "tx";
```

Reference to the clock

```
clocks = <&clks 45>;
```

The device is not enabled.

```
status = "disabled";
};
```

Taken from arch/arm/boot/dts/imx28.dtsi

A simple example, driver side (1)

The compatible string used to bind a device with the driver

```
static struct of_device_id mxs_auart_dt_ids[] = {
	{
		.compatible = "fsl,imx28-auart",
		.data = &mxs_auart_devtype[IMX28_AUART]
	}, {
		.compatible = "fsl,imx23-auart",
		.data = &mxs_auart_devtype[IMX23_AUART]
	}, { /* sentinel */ }
};

MODULE_DEVICE_TABLE(of, mxs_auart_dt_ids); [...]
static struct platform_driver mxs_auart_driver = {
	.probe = mxs_auart_probe,
	.remove = mxs_auart_remove,
	.driver = {
		.name = "mxs-auart",
		.of_match_table = mxs_auart_dt_ids,
	},
};
```

A simple example, driver side (2)

`of_match_device` gets the matching entry in the `mxs_auart_dt_ids` table.

Useful to get the driver-specific `data` field, typically used to alter the behavior of the driver depending on the variant of the detected device.

```
static int mxs_auart_probe(struct platform_device *pdev)
{
        const struct of_device_id *of_id =
                        of_match_device(mxs_auart_dt_ids, &pdev->dev);

        if (of_id) {
                /* Use of_id->data here */ [...]
        } [...]
}
```

A simple example, driver side (3)

- **Getting a reference to the clock**
  - described by the clocks property
  - s->clk = clk_get(&pdev->dev, NULL);
- **Getting the I/O registers resource**
  - described by the reg property
  - r = platform_get_resource(pdev, IORESOURCE_MEM, 0);
- **Getting the interrupt**
  - described by the interrupts property
  - s->irq = platform_get_irq(pdev, 0);
- **Get a DMA channel**
  - described by the dmas property
  - s->rx_dma_chan = dma_request_slave_channel(s->dev, "rx");
  - s->tx_dma_chan = dma_request_slave_channel(s->dev, "tx");
- **Check some custom property**
  - struct device_node *np = pdev->dev.of_node;
  - if (of_get_property(np, "fsl,uart-has-rtscts", NULL))

Device Tree inclusion

- Device Tree files are not monolithic, they can be split in several files, including each other.
  - `.dtsi` files are included files, while `.dts` files are *final* Device Trees
- Typically, `.dtsi` will contain definition of SoC-level information (or sometimes definitions common to several almost identical boards).
- The `.dts` file contains the board-level information.
- The inclusion works by overlaying the tree of the including file over the tree of the included file.
- Inclusion using the DT operator `/include/`, or since a few kernel releases, the DTS go through the C preprocessor, so `#include` is recommended.

Device Tree inclusion example

Definition of the AM33xx SoC

```
/ {
     compatible = "ti,am33xx";
     [...]
     ocp {
          uart0: serial@44e09000 {
               compatible = "ti,omap3-uart";
               reg = <0x44e09000 0x2000>;
               interrupts = <72>;
               status = "disabled";
          };
     };
};
                                        am33xx.dtsi
```

+

Definition of the BeagleBone board

```
#include "am33xx.dtsi"

/ {
     compatible = "ti,am335x-bone", "ti,am33xx";
     [...]
     ocp {
          uart0: serial@44e09000 {
               pinctrl-names = "default";
               pinctrl-0 = <&uart0_pins>;
               status = "okay";
          };
     };
};
                                        am335x-bone.dts
```

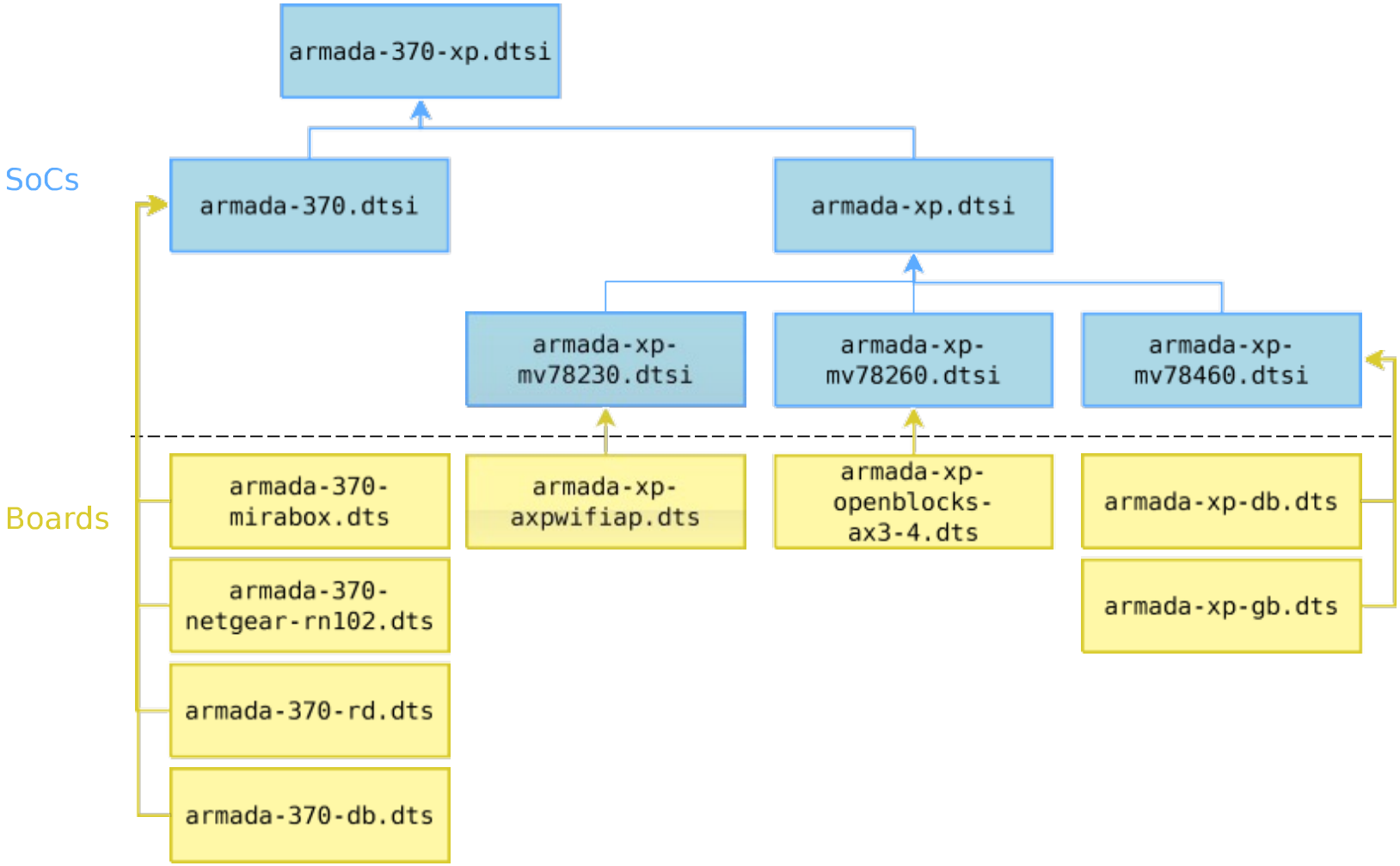
=

Compiled DTB

```
/ {
     compatible = "ti,am335x-bone", "ti,am33xx";
     [...]
     ocp {
          uart0: serial@44e09000 {
               compatible = "ti,omap3-uart";
               reg = <0x44e09000 0x2000>;
               interrupts = <72>;
               pinctrl-names = "default";
               pinctrl-0 = <&uart0_pins>;
               status = "okay";
          };
     };
};
                                        am335x-bone.dtb
```

Note: the real DTB is in binary format. Here we show the text equivalent of the DTB contents;

Device Tree inclusion example (2)



Device Tree binding documentation example

```
* Freescale MXS Application UART (AUART) Required properties:
- compatible : Should be "fsl,<soc>-auart". The supported SoCs include imx23 and imx28.
- reg : Address and length of the register set for the device
- interrupts : Should contain the auart interrupt numbers
- dmas: DMA specifier, consisting of a phandle to DMA controller node and AUART DMA
  channel ID.
  Refer to dma.txt and fsl-mxs-dma.txt for details.
- dma-names: "rx" for RX channel, "tx" for TX channel.

Example:

auart0: serial@8006a000 {
        compatible = "fsl,imx28-auart", "fsl,imx23-auart"; reg = <0x8006a000
        0x2000>;
        interrupts = <112>;
        dmas = <&dma_apbx 8>, <&dma_apbx 9>; dma-names = "rx",
        "tx";
};

Note: Each auart port should have an alias correctly numbered in "aliases" node.

Example: [...]
```

Documentation/devicetree/bindings/tty/serial/fsl-mxs-auart.txt

Device Tree organization: top-level nodes

- **Under the root of the Device Tree, one typically finds the following top-level nodes:**
  - A ‘cpus’ node, which sub-nodes describing each CPU in the system.
  - A memory node, which defines the location and size of the RAM.
  - A chosen node, which defines parameters chosen or defined by the system firmware at boot time. In practice, one of its usage is to pass the kernel command line.
  - An aliases node, to define shortcuts to certain nodes.
  - One or more nodes defining the buses in the SoC.
  - One or mode nodes defining on-board devices.

Top-level nodes

```
compatible = "avnet,ultra96-rev1", "avnet,ultra96", "xlnx,zynqmp-zcu100-revC", "xlnx,zynqmp-zcu100",
"xlnx,zynqmp";
            #address-cells = <0x2>;
            #size-cells = <0x2>;
            model = "Avnet Ultra96 Rev1";

            cpus {
                        #address-cells = <0x1>;
                        #size-cells = <0x0>;

                        cpu@0 {
                                    compatible = "arm,cortex-a53", "arm,armv8";
                                    device_type = "cpu";
                                    enable-method = "psci";
                                    operating-points-v2 = <0x1>;
                                    reg = <0x0>;
                                    cpu-idle-states = <0x2>;
                                    clocks = <0x3 0xa>;
                        };

                        cpu@1 {
                                    compatible = "arm,cortex-a53", "arm,armv8";
                                    device_type = "cpu";
                                    enable-method = "psci";
                                    reg = <0x1>;
                                    operating-points-v2 = <0x1>;
                                    cpu-idle-states = <0x2>;
                        };
```

Device Tree organization: imx28.dtsi

arch/arm/boot/dts/imx28.dtsi

```
/ {
        aliases { ... };
        cpus { ... };

        apb@80000000 {
                apbh@80000000 {
                        /* Some devices */
                };

                apbx@80040000 {
                        /* Some devices */
                };
        };

        ahb@80080000 {
                /* Some devices */
        };
};
```

i.MX28 buses organization

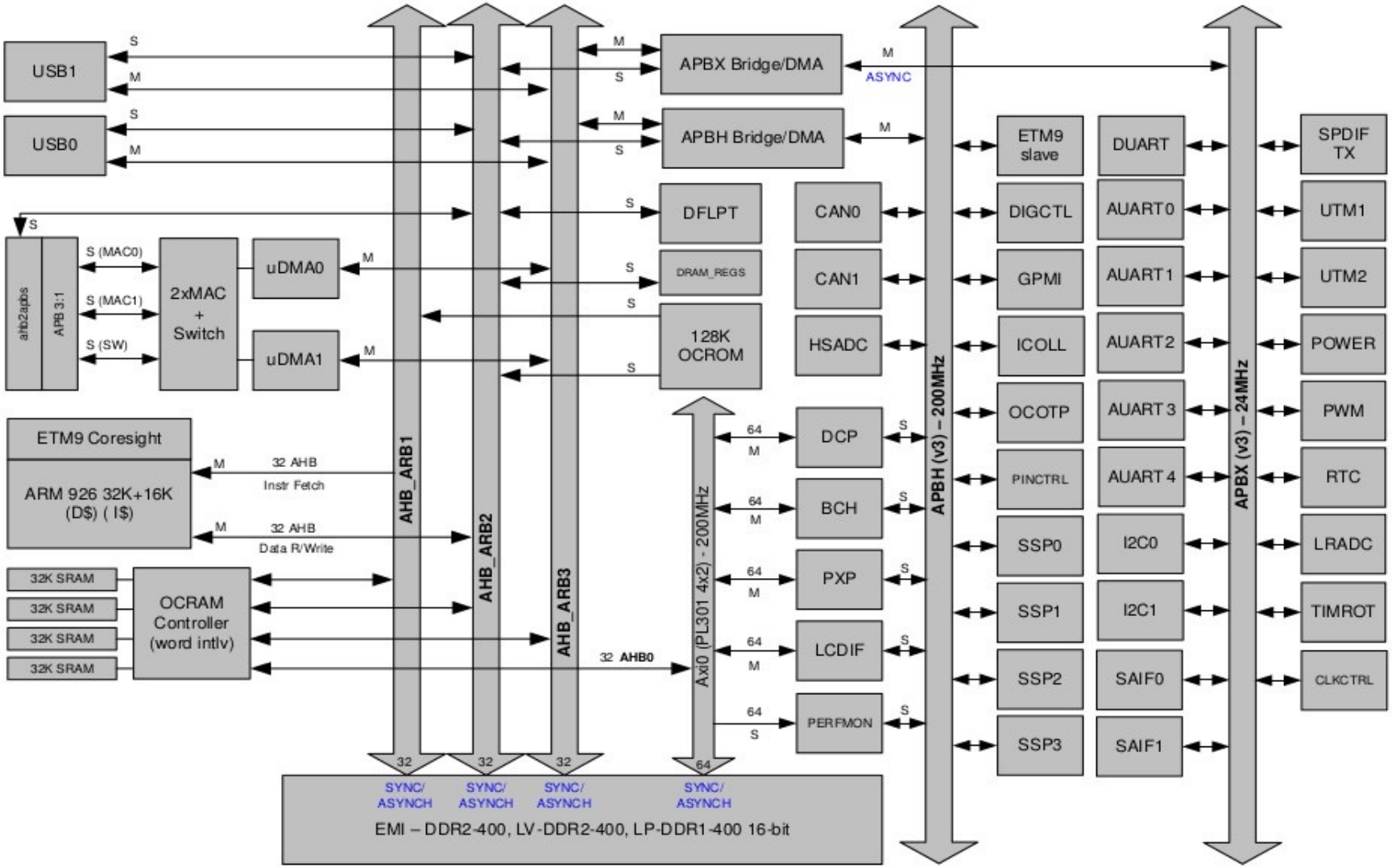


Figure 1-2. i.MX28 SOC System Buses

Device Tree organization: imx28-evk.dts

arch/arm/boot/dts/imx28-evk.dts

```
{
        model = "Freescale i.MX28 Evaluation Kit";
        compatible = "fsl,imx28-evk", "fsl,imx28";

        memory {
                reg = <0x40000000 0x08000000>;
        };


        apb@80000000 {
                apbh@80000000 { ... };
                apbx@80040000 { ... };
        };

        ahb@80080000 { ... };

        sound { ... };
        leds { ... };
        backlight { ... };
};
```

Top-level compatible property

- The top-level compatible property typically defines a compatible string for the board, and then for the SoC.
- Values always given with the most-specific first, to least-specific last.
- Used to match with the dt_compat field of the DT_MACHINE structure:

```
static const char *mxs_dt_compat[] __initdata = {
    "fsl,imx28",
    "fsl,imx23", NULL,
};

DT_MACHINE_START(MXS, "Freescale MXS (Device Tree)")
```

- Can also be used within code to test the machine:

```
 if (of_machine_is_compatible("fsl,imx28-evk"))
        imx28_evk_init();
```

Bus, address cells and size cells

- **Inside a bus, one typically needs to define the following properties:**
  - A compatible property, which identifies the bus controller (in case of I2C, SPI, PCI, etc.). A special value `compatible = "simple-bus"` means a simple memory-mapped bus with no specific handling or driver.
  - Child nodes will be registered as platform devices.
- **The `#address-cells` property indicate how many cells (i.e 32 bits values) are needed to form the base address part in the reg property.**
- **The `#size-cells` is the same, for the size part of the reg property.**
- **The ranges property can describe an address translation between the child bus and the parent bus. When simply defined as ranges; it means that the translation is an identity translation.**

simple-bus, address cells and size cells

```
apbh@80000000 {
        compatible = "simple-bus";
        #address-cells = <1>;
        #size-cells = <1>;
        reg = <0x80000000 0x3c900>;
        ranges;

        [...]

        hsadc: hsadc@80002000 {
                reg = <0x80002000 0x2000>;
                interrupts = <13>; dmas = <&dma_apbh 12>; dma-names = "rx";
                status = "disabled";
        };

        [...]
};
```

I2C bus, address cells and size cells

```
i2c0: i2c@80058000 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "fsl,imx28-i2c"; reg = <0x80058000 0x2000>;
        interrupts = <111>; [...]

        sgtl5000: codec@0a {
                compatible = "fsl,sgtl5000"; reg = <0x0a>;
                VDDA-supply = <&reg_3p3v>; VDDIO-supply = <&reg_3p3v>;
                        clocks = <&saif0>;
        };

        at24@51 {
                compatible = "at24,24c32"; pagesize = <32>;
                reg = <0x51>;
        };
};
```

Interrupt handling

- `interrupt-controller`; is a Boolean property that indicates that the current node is an interrupt controller.
- `#interrupt-cells` indicates the number of cells in the interrupts property for the interrupts managed by the selected interrupt controller.
- `interrupt-parent` is a `phandle` that points to the interrupt controller for the current node. There is generally a top-level interrupt-parent definition for the main interrupt controller.

Interrupt example: imx28.dtsi

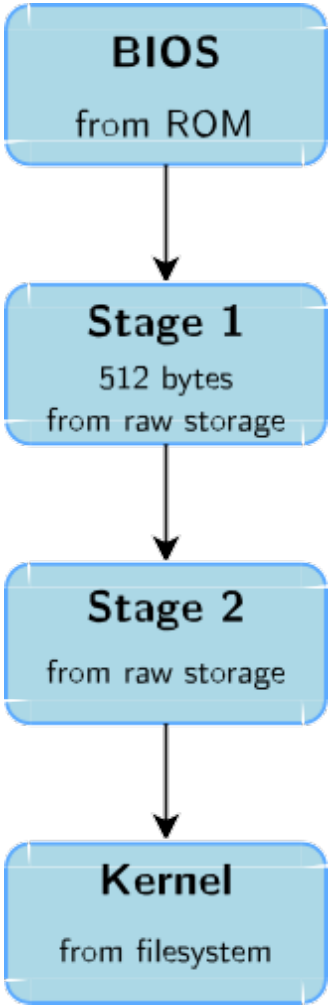
```
{
    interrupt-parent = <&icoll>; apb@80000000 {
        apbh@80000000 {
            icoll: interrupt-controller@80000000 { compatible = "fsl,imx28-
                icoll", "fsl,icoll"; interrupt-controller;
                #interrupt-cells = <1>; reg = <0x80000000
                0x2000>;
            };

            ssp0: ssp@80010000 { [...]
                interrupts = <96>;
            };
        };
    };
};
```

Questions

Backup

Bootloaders on BIOS-based x86 (1)

- The x86 processors are typically bundled on a board with a non-volatile memory containing a program, the BIOS.
- On old BIOS-based x86 platforms: the BIOS is responsible for basic hardware initialization and loading of a very small piece of code from non-volatile storage.
- This piece of code is typically a 1st stage boot loader, which will load the full boot loader itself.
- It typically understands file system formats so that the kernel file can be loaded directly from a normal file system.
- This sequence is different for modern EFI-based systems.

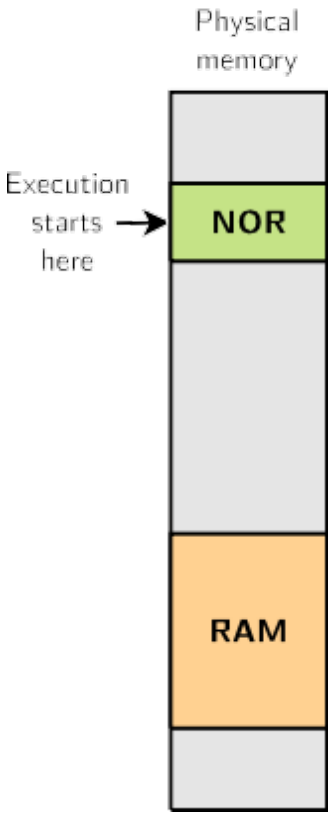


Bootloaders on x86 (2)

- **GRUB, Grand Unified Bootloader, the most powerful one.**
  - http://www.gnu.org/software/grub/
  - Can read many file system formats to load the kernel image and the configuration, provides a powerful shell with various commands, can load kernel images over the network, etc.
- **Syslinux, for network and removable media booting (USB key, CD-ROM)**
  - http://www.kernel.org/pub/linux/utils/boot/syslinux/

Booting on embedded CPUs: case 1

- When powered, the CPU starts executing code at a fixed address
- There is no other booting mechanism provided by the CPU
- The hardware design must ensure that a NOR flash chip is wired so that it is accessible at the address at which the CPU starts executing instructions
- The first stage bootloader must be programmed at this address in the NOR
- NOR is mandatory, because it allows random access, which NAND doesn't allow
- Not very common anymore (unpractical, and requires NOR flash)

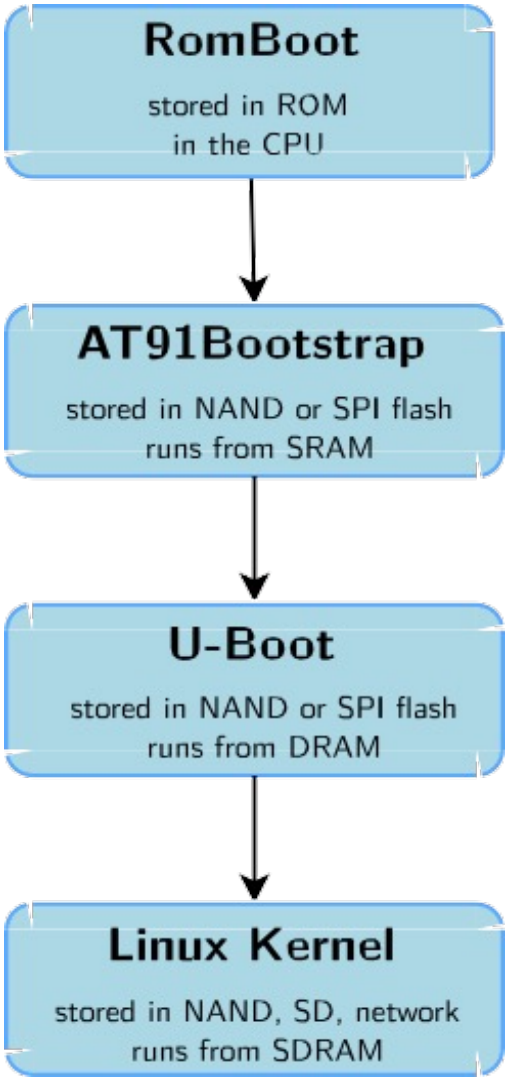


Booting on embedded CPUs: case 2

- The CPU has an integrated boot code in ROM
  - BootROM on AT91 CPUs, "ROM code" on OMAP, etc.
  - Exact details are CPU-dependent
- This boot code is able to load a first stage bootloader from a storage device into an internal SRAM (DRAM not initialized yet)
  - Storage device can typically be: MMC, NAND, SPI flash, UART (transmitting data over the serial line), etc.
- The first stage bootloader is:
  - Limited in size due to hardware constraints (SRAM size)
  - Provided either by the CPU vendor or through community projects
- This first stage bootloader must initialize DRAM and other hardware devices and load a second stage bootloader into DRAM

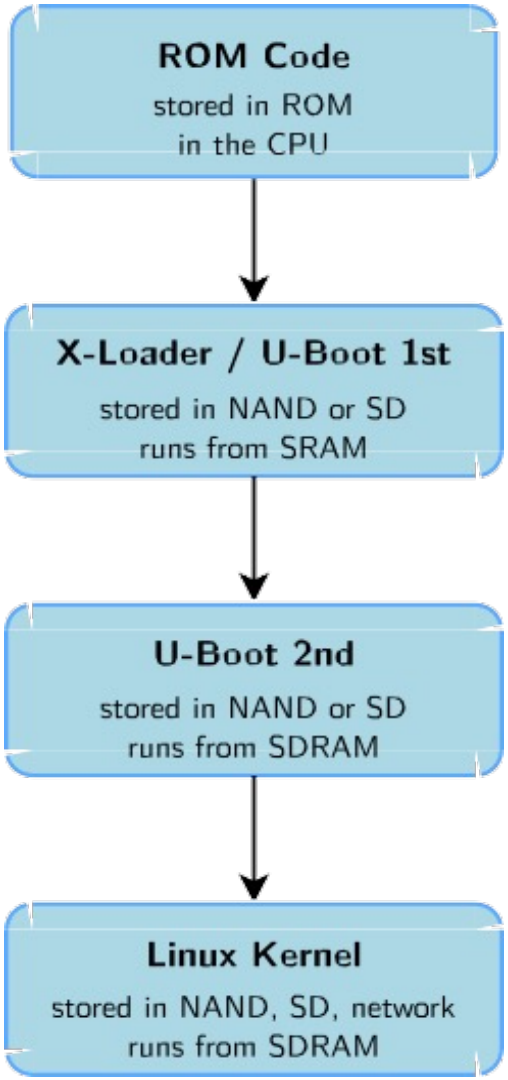
Booting on ARM Atmel AT91

- **RomBoot: tries to find a valid bootstrap image from various storage sources, and load it into SRAM (DRAM not initialized yet). Size limited to 4 KB. No user interaction possible in standard boot mode.**
- **AT91Bootstrap: runs from SRAM. Initializes the DRAM, the NAND or SPI controller, and loads the secondary bootloader into RAM and starts it. No user interaction possible.**
- **U-Boot: runs from RAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to RAM and starts it. Shell with commands provided.**
- **Linux Kernel: runs from RAM. Takes over the system completely (the bootloader no longer exists).**



Booting on ARM TI OMAP3

- ROM Code: tries to find a valid bootstrap image from various storage sources, and load it into SRAM or RAM (RAM can be initialized by ROM code through a configuration header). Size limited to <64 KB. No user interaction possible.
- X-Loader or U-Boot: runs from SRAM. Initializes the DRAM, the NAND or MMC controller, and loads the secondary bootloader into RAM and starts it. No user interaction possible. File called MLO.
- U-Boot: runs from RAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to RAM and starts it. Shell with commands provided. File called u-boot.bin or u-boot.img.
- Linux Kernel: runs from RAM. Takes over the system completely (bootloaders no longer exists).

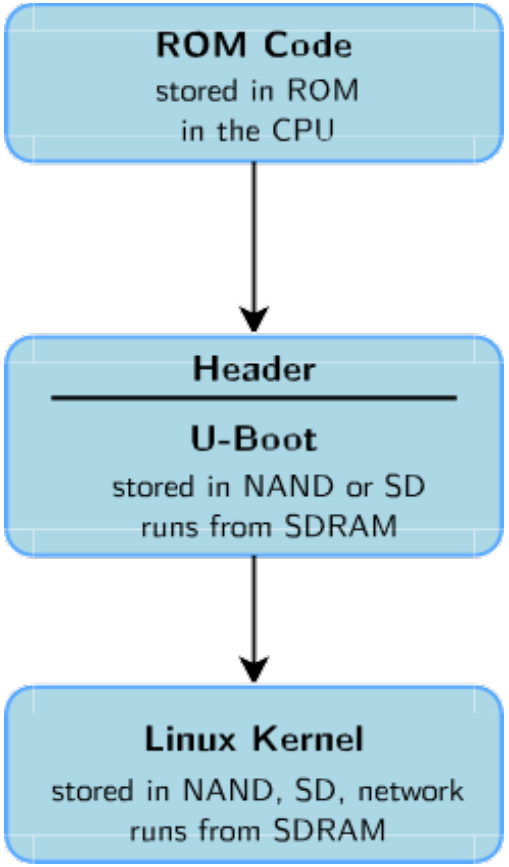


Booting on Marvell SoC

- **ROM Code: tries to find a valid bootstrap image from various storage sources, and load it into RAM. The RAM configuration is described in a CPU-specific header, prepended to the bootloader image.**

- **U-Boot: runs from RAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to RAM and starts it. Shell with commands provided. File called u-boot.kwb.**

- **Linux Kernel: runs from RAM. Takes over the system completely (bootloaders no longer exists).**



Generic bootloaders for embedded CPUs

- **We will focus on the generic part, the main bootloader, offering the most important features.**
- **There are several open-source generic bootloaders.**
- **Here are the most popular ones:**
  - **U-Boot, the universal bootloader by Denx**
  - **The most used on ARM, also used on PPC, MIPS, x86, m68k, NIOS, etc. The de-facto standard nowadays.**

  - **Barebox, an architecture-neutral bootloader, written as a successor of U-Boot. It doesn't have as much hardware support as U-Boot yet. U-Boot has improved quite a lot thanks to this competitor.**

Zynq environment variables commands - Example

```
Zynq> printenv
aa=fatload mmc 0 0x1000000 system.bit
baudrate=115200
bb=fpga loadb 0 0x1000000 0x03dbb6a
bitstream_image=system.bit.bin
boot_image=BOOT.bin
boot_size=0xF00000
bootargs=console=ttyPS0,115200 root=/dev/mmcblk0p2 rw rootwait earlyprintk
bootcmd=run $modeboot
bootdelay=2
bootenv=uEnv.txt
devicetree_image=system.dtb
devicetree_load_address=0x2000000
devicetree_size=0x20000
dfu_mmc=run dfu_mmc_info && dfu 0 mmc 0
dfu_mmc_info=set dfu_alt_info ${kernel_image} fat 0 1\\;${devicetree_image} fat 0 1\\;${ramdisk_image} fat 0 1
dfu_ram=run dfu_ram_info && dfu 0 ram 0
dfu_ram_info=set dfu_alt_info ${kernel_image} ram 0x3000000 0x500000\\;${devicetree_image} ram
0x2A00000 0x20000\\;${ramdisk_image} ram 0x2000000 0x600000
ethaddr=00:0a:55:00:aa:23
```

Zynq environment variables commands - Example

```
fdt_high=0x20000000
fdtcontroladdr=1ff960c0
fileaddr=2000000
filesize=1a8
importbootenv=echo Importing environment from SD ...; env import -t ${loadbootenv_addr} $filesize
initrd_high=0x20000000
jtagboot=echo TFTPing Linux to RAM... && tftpboot ${kernel_load_address} ${kernel_image} && tftpboot
${devicetree_load_address} ${devicetree_image} && tftpboot ${ramdisk_load_address} ${ramdisk_image} &&
bootm ${kernel_load_address} ${ramdisk_load_address} ${devicetree_load_address}
kernel_image=uImage
kernel_load_address=0x2080000
kernel_size=0x500000
loadbit_addr=0x100000
loadbootenv=load mmc 0 ${loadbootenv_addr} ${bootenv}
loadbootenv_addr=0x2000000
mmc_loadbit=echo Loading bitstream from SD/MMC/eMMC to RAM.. && mmcinfo && load mmc 0
${loadbit_addr} ${bitstream_image} && fpga load 0 ${loadbit_addr} ${filesize}
modeboot=sdboot
nandboot=echo Copying Linux from NAND flash to RAM... && nand read ${kernel_load_address} 0x100000
${kernel_size} && nand read ${devicetree_load_address} 0x600000 ${devicetree_size} && echo Copying
ramdisk... && nand read ${ramdisk_load_address} 0x620000 ${ramdisk_size} && bootm ${kernel_load_address}
${ramdisk_load_address} ${devicetree_load_address}
```

Zynq environment variables commands – Example

```
norboot=echo Copying Linux from NOR flash to RAM... && cp.b 0xE2100000 ${kernel_load_address} ${kernel_size} && cp.b 0xE2600000 ${devicetree_load_address} ${devicetree_size} && echo Copying ramdisk... && cp.b 0xE2620000 ${ramdisk_load_address} ${ramdisk_size} && bootm ${kernel_load_address} ${ramdisk_load_address} ${devicetree_load_address}
preboot=if test $modeboot = sdboot && env run sd_uEnvtxt_existence_test; then if env run loadbootenv; then env run importbootenv; fi; fi;
qspiboot=echo Copying Linux from QSPI flash to RAM... && sf probe 0 0 0 && sf read ${kernel_load_address} 0x100000 ${kernel_size} && sf read ${devicetree_load_address} 0x600000 ${devicetree_size} && echo Copying ramdisk... && sf read ${ramdisk_load_address} 0x620000 ${ramdisk_size} && bootm ${kernel_load_address} ${ramdisk_load_address} ${devicetree_load_address}
ramdisk_image=uramdisk.image.gz
ramdisk_load_address=0x4000000
ramdisk_size=0x5E0000
rsa_jtagboot=echo TFTPing Image to RAM... && tftpboot 0x100000 ${boot_image} && zynqrsa 0x100000 && bootm ${kernel_load_address} ${ramdisk_load_address} ${devicetree_load_address}
rsa_nandboot=echo Copying Image from NAND flash to RAM... && nand read 0x100000 0x0 ${boot_size} && zynqrsa 0x100000 && bootm ${kernel_load_address} ${ramdisk_load_address} ${devicetree_load_address}
rsa_norboot=echo Copying Image from NOR flash to RAM... && cp.b 0xE2100000 0x100000 ${boot_size} && zynqrsa 0x100000 && bootm ${kernel_load_address} ${ramdisk_load_address} ${devicetree_load_address}
```

Zynq environment variables commands – Example

```
rsa_sdboot=echo Copying Image from SD to RAM... && load mmc 0 0x100000 ${boot_image} && zynqrsa 0x100000
&& bootm ${kernel_load_address} ${ramdisk_load_address} ${devicetree_load_address}
sd_uEnvtxt_existence_test=test -e mmc 0 /uEnv.txt
sdboot=if mmcinfo; then run uenvboot; echo Copying Linux from SD to RAM... && load mmc 0
${kernel_load_address} ${kernel_image} && load mmc 0 ${devicetree_load_address} ${devicetree_image} &&
load mmc 0 ${ramdisk_load_address} ${ramdisk_image} && bootm ${kernel_load_address}
${ramdisk_load_address} ${devicetree_load_address}; fi
stderr=serial@e0001000
stdin=serial@e0001000
stdout=serial@e0001000
thor_mmc=run dfu_mmc_info && thordown 0 mmc 0
thor_ram=run dfu_ram_info && thordown 0 ram 0
uenvboot=if run loadbootenv; then echo Loaded environment from ${bootenv}; run importbootenv; fi; if test -n
$uenvcmd; then echo Running uenvcmd ...; run uenvcmd; fi
uenvcmd=echo Copying Linux from SD to RAM... && ${aa} && ${bb} && fatload mmc 0 0x3000000 ${kernel_image}
&& fatload mmc 0 0x2A00000 ${devicetree_image} && bootm 0x3000000 - 0x2A00000
usbboot=if usb start; then run uenvboot; echo Copying Linux from USB to RAM... && load usb 0
${kernel_load_address} ${kernel_image} && load usb 0 ${devicetree_load_address} ${devicetree_image} && load
usb 0 ${ramdisk_load_address} ${ramdisk_image} && bootm ${kernel_load_address} ${ramdisk_load_address}
${devicetree_load_address}; fi

Environment size: 5202/131068 bytes
```