**Overview of arm Linux Boot sequence**

We will walk through boot up code for AT91 system-on-chip, built around the ARM926ejs ARM Thumb processor. You can also read ARM Architecture Reference Manual for better understanding the boot process.( ARM.System.Developers.Guide-Designing.and.Optimizing.System.Software.pdf).

**Components in Linux Boot Process**:  
———————————————–

Linux boot process involes the following components.

* Bootloader
* Kernel Image
* Root Filesystem

Before we see how the above components work, the following is the call flow of Linux Kernel boot process for arm architecture. This gives a big picture on whole Linux boot process. We use U-boot bootloader.

**U-boot:**

\_start (cpu/arm920t/start.S)

start\_code (cpu/arm920t/start.S)

start\_armboot (lib\_arm/board.c)

board\_init (board/atmel/at91sam9263ek/at91sam9263ek.c)

timer\_init (cpu/arm926ejs/at91/timer.c)

serial\_init (drivers/serial/atmel\_usart.c)

main\_loop (lib\_arm/board.c)

Now u-boot is up and running and is in u-boot prompt and ready to accept commands. Assume that kernel image is loaded into RAM and issued bootm command.

do\_bootm (common/cmd\_bootm.c)

bootm\_start (common/cmd\_bootm.c)

bootm\_load\_os (common/cmd\_bootm.c)

do\_bootm\_linux (lib\_arm/bootm.c)

stext (linux/arch/arm/kernel/head.S)

Control is given to linux.

**Linux Kernel:**

—————-

stext (arch/arm/kernel/head.S:78)

\_\_lookup\_processor\_type (arch/arm/kernel/head-common.S:160)

\_\_lookup\_machine\_type (arch/arm/kernel/head-common.S:211)

\_\_create\_page\_tables (arch/arm/kernel/head.S:219)

\_\_arm920\_setup (arch/arm/mm/proc-arm926.S:389)

\_\_enable\_mmu (arch/arm/kernel/head.S:160)

\_\_turn\_mmu\_on (arch/arm/kernel/head.S:205)

\_\_switch\_data (arch/arm/kernel/head-common.S:20)

start\_kernel (init/main.c:529)

start\_kernel (init/main.c:529)

tick\_init(kernel/time/tick-common.c:413)

setup\_arch (arch/arm/kernel/setup.c:666)

setup\_machine (arch/arm/kernel/setup.c:369)

lookup\_machine\_type ( )

setup\_command\_line (init/main.c:408)

build\_all\_zonelists (mm/page\_alloc.c:3031)

parse\_args (kernel/params.c:129)

mm\_init (init/main.c:516)

mem\_init (arch/arm/mm/init.c:528)

kmem\_cache\_init (mm/slab.c, mm/slob.c, mm/slub.c)

sched\_init (kernel/sched.c)

init\_IRQ (arch/arm/kernel/irq.c)

init\_timers (kernel/timer.c:1713)

hrtimers\_init (kernel/hrtimer.c:1741)

softirq\_init (kernel/softirq.c:674)

console\_init (drivers/char/tty\_io.c:3084)

vfs\_caches\_init (fs/dcache.c:2352)

mnt\_init (fs/namespace.c:2308)

init\_rootfs ()

init\_mount\_tree (fs/namespace.c:2285)

do\_kern\_mount (fs/namespace.c:1053)

set\_fs\_pwd(fs/fs\_struct.c:29)

set\_fs\_root(fs/fs\_struct.c:12)

bdev\_cache\_init (fs/block\_dev.c:465)

chrdev\_init (fs/char\_dev.c:566)

signals\_init (kernel/signal.c:2737)

rest\_init (init/main.c:425)

kernel\_thread (431, arch/arm/kernel/process.c:388)

kernel\_thread() creates a kernel thread and control is given to kernel\_init().

kernel\_init (431, init/main.c:856)

do\_basic\_setup (888, init/main.c:787)

init\_workqueues (789, kernel/workqueue.c:1204)

driver\_init (793, drivers/base/init.c:20)

do\_initcalls (796, init/main.c:769) /\* Calls all subsytems init functions \*/

prepare\_namespace (906, init/do\_mounts.c:366)

initrd\_load (399, init/do\_mounts\_initrd.c:107)

rd\_load\_image (117, init/do\_mounts\_rd.c:158) /\* if initrd is given \*/

identify\_ramdisk\_image (179, init/do\_mounts\_rd.c:53)

handle\_initrd (119, init/do\_mounts\_initrd.c:37) /\*if rd\_load\_image is success \*/

mount\_block\_root (45, init/do\_mounts.c:233)

do\_mount\_root (247, init/do\_mounts.:218)

mount\_root (417, init/do\_mounts.c:334) /\* if initrd not given \*/

mount\_block\_root (359, init/do\_mounts.c:233)

do\_mount\_root (247, init/do\_mounts.c:218)

init\_post (915, init/main.c:816)

run\_init\_process (847, init/main.c:807)

kernel\_execve (810, arch/arm/kernel/sys\_arm.c:81)

**User Space**

————-

init() /\*userspace /sbin/init \*/

**Bootloader**

A bootloader is a small program which will load the kernel image into RAM and boots up the kernel image. This is also called bootstrap as it brings(pulls) up system by loading an operating system. Bootloader starts before any other software starts and initializes the processor and makes cpu ready to execute a program like an operating system. Most processors have a default address from which the first bytes of code are fetched upon power is applied or board is reset. Hardware designers use this information to store the bootloader code at that address in ROM or flash. Since it should initialize the cpu and should run a program which is located at architecture specific address bootloaders are highly processor specific and board specific. Every embedded board comes with a bootstrap to download the kernel image or standalone application into the board and start executing the kernel image or application. Bootloader will be executed when power is applied to a processor board. Basically it will have some minimal features to load the image and boot it up.

It is also possible to control the system using a hardware debug interface such as JTAG. This interface may be used to write the boot loader program into bootable non-volatile memory (e.g. flash) by instructing the processor core to perform the necessary actions to program non-volatile memory. Generally done for first time to download the basic bootloader and for some recovery process. JTAG is a standard and popular interface provided by many board vendors. Some micro controllers provide special hardware interfaces which can’t be used to take arbitrary control of a system or directly run code, but instead they allow the insertion of boot code into bootable non-volatile memory (like flash memory) via simple protocols. Then at the manufacturing phase, such interfaces are used to inject boot code (and possibly other code) into non-volatile memory. After system reset, the micro controller begins to execute code programmed into its non-volatile memory, just like usual processors are using ROMs for booting. In many cases such interfaces are implemented by hardwired logic. In other cases such interfaces could be created by software running in integrated on-chip boot ROM from GPIO pins.

There are some other third party bootloaders available which provide rich set of features and easy user interface. You can download these third party bootloaders into board and can make them default bootloaders for your board. Generally bootloaders provided by board vendors are replaced with these third party bootloader. There are a quite few third party boolader available and some of them are open source (or free bootloaders) and some are commercial. Some of them are Das U-Boot, Red boot, GRUB (for desktops), LILO , Loadlin, , bootsect-loader, SYSLINUX, EtherBoot, ELILO.

We will take U-boot boot loader as our boot loader. U-boot is the widely used boot loader in embedded systems. We will explain code from the u-boot-2010.03 source. You can download U-boot from the following site.

<http://www.denx.de/wiki/U-Boot>

**How U-boot is built:**

Based on the configuration of U-boot, all the assembly files (.S) and C files (.c) are compiled using cross compiler which is built for a particular architecture and object files(.o) will be generated. All these object files are linked by linker and an executable file will be created. An object file or executable file is a collection of sections like .text, .data, .bss etc. Object files and executable files have a file format like elf. All the sections of the object files will be arranged in the executable file based on a script called linker script. This script tells where all the sections are to be loaded in the memory when it runs. Understanding this script is very important to know how boot loader and kernel are composed and how different sections of boot loader or kernel are loaded in the memory.

Generally, when a program is run (executed) a loader reads executable file and loads different sections of the executable file in the specified memory location and starts executing the start function(entry point) specified in the linker script. But, if you want to run(load) a boot loader there will not be any loader to load(basically to understand the file format) different sections of executable file into the memory. Then you need to use a tool called objcopy which will take all sections from the executable file and create a binary file which doesn’t have any file format. This binary file can be loaded into the memory and executed or can be written in to the ROM at a particular address (specific to the architecture) which will be executed by cpu when power is applied to the board. You can find good tutorial on linker script in the following location.

<http://www.redhat.com/docs/manuals/enterprise/RHEL-4-Manual/gnu-linker/scripts.html>

Assume that based on the U-boot configuration all files are compiled and object files are created. U-boot makefile uses the following linker script (specific to architecture) to build an executable file.  
File: cpu/arm926ejs/u-boot.lds

32 OUTPUT\_FORMAT(“elf32-littlearm”, “elf32-littlearm”, “elf32-littlearm”)

33 OUTPUT\_ARCH(arm)

34 ENTRY(\_start)

35 SECTIONS

36 {

37 . = 0×00000000;

38

39 . = ALIGN(4);

40 .text :

41 {

42 cpu/arm920t/start.o (.text)

43 \*(.text)

44 }

4546 . = ALIGN(4);

47 .rodata : { \*(SORT\_BY\_ALIGNMENT(SORT\_BY\_NAME(.rodata\*))) }

48

49 . = ALIGN(4);

50 .data : { \*(.data) }

51

52 . = ALIGN(4);

53 .got : { \*(.got) }

54

55 . = .;

56 \_\_u\_boot\_cmd\_start = .;

57 .u\_boot\_cmd : { \*(.u\_boot\_cmd) }

58 \_\_u\_boot\_cmd\_end = .;

59

60 . = ALIGN(4);

61 \_\_bss\_start = .;

62 .bss (NOLOAD) : { \*(.bss) . = ALIGN(4); }

63 \_end = .;

64 }

OUTPUT\_FORMAT in line #32 specify the file format of the executable file. Here the executable file format is elf32 and endianness is little endian. OUTPUT\_ARCH in line # 33 specify the architecture on which this code runs. ENTRY in line #34 specifies the start function(entry point) of u-boot program. Here the entry point is *\_start*.

SECTIONS in line #35 define how different sections are mapped in the executable file. Loader uses the addresses specified in this section to load different section of the program into the memory.

‘.’ in the line #37 specifies the start address where the following sections should be loaded. In this case start address is 0×00000000. After this in line #39 the memory is aligned by 4 bytes and the .text section follows in the line #40.

40 .text :

41 {

42 cpu/arm920t/start.o (.text)

43 \*(.text)

44 }

At the ‘.’ position (0×00000000) the code in the cpu/arm926ejs/start.o is mapped and follows the code that is there in .text sections of all other object (.o) files. cpu/arm920t/start.o contains the \_start() function(in assembly language) which is entry point of this program.

Now the ‘.’ will be at 0×00000000 + sizeof (.text). Again memory is aligned by 4 bytes and .rodata section follows in line #47.

. = ALIGN(4);

47 .rodata : { \*(SORT\_BY\_ALIGNMENT(SORT\_BY\_NAME(.rodata\*))) }

.rodata sections from all objects files are mapped at this address. Follows the .data and .git sections.

49 . = ALIGN(4);

50 .data : { \*(.data) }

51

52 . = ALIGN(4);

53 .got : { \*(.got) }

Each U-boot command is an object of type ‘cmd\_tbl\_t’ which contains command name, help string and function pointer to be executed when this command is run. All these command objects are placed in the memory sequentially. Each of this command object is built into an U-boot defined section called .u\_boot\_cmd in the object file. These all .u\_boot\_cmd sections are placed in the memory after the above sections(.data and .git).

. = .;

56 \_\_u\_boot\_cmd\_start = .;

57 .u\_boot\_cmd : { \*(.u\_boot\_cmd) }

58 \_\_u\_boot\_cmd\_end = .;

\_\_u\_boot\_cmd\_start contains the start of the commands objects and \_\_u\_boot\_cmd\_end contains the end of the command objects.

And next follows the .bss (uninitialized global variables) sections.

60 . = ALIGN(4);

61 \_\_bss\_start = .;

62 .bss (NOLOAD) : { \*(.bss) . = ALIGN(4); }

63 \_end = .;

\_\_bss\_start points to the .bss start address and \_end contains the end of the all sections.

Using this linker script linker will generate an executable file called u-boot. Objcopy tool is used to generate a binary file from the u-boot executable file.

u-boot.bin: u-boot

$(OBJCOPY) ${OBJCFLAGS} -O binary $< $@

U-boot binary file will be copied to the board RAM or written in the flash disk. At91 board comes with a boot programmer, A tiny program called atmel bootstrap that can be used to download the image into flash or RAM and start execute or to reset a corrupted board. U-boot will be copied to flash disk or internal RAM( if it is less size) and will be downloaded to RAM and will be executed when the board power is applied to the board. For this board the code is always downloaded from device address 0×0000\_0000 to the address 0×0000\_0000 of the SRAM after remap. That ’s why we have given the start address of the .text section as 0×00000000. If you want to load the code any where in the RAM and want to execute U-boot you need to build you code as position independent code(PIC). Then the instructions addresses will be offset into to PC(cpu register) value. So the downloaded code must be position-independent or linked at address 0×0000\_0000. For our explanation purpose assume that U-boot code is linked at 0×00000000 and the Boot program downloaded U-boot from the data flash(Check AT91 spec for downloading process)) and call entry point into the U-boot.

**Kernel startup**

The kernel start-up can be split into the following phases.

***CPU/Platform-Specific Initialization***

If you are porting Linux to your platform this section is very important as it marks the important milestones in BSP porting. The platform-specific initialization consists of the following steps.

*Setting up the environment for the first C routine:*

The kernel entry point is an assembly language routine; the name of this entry point varies (stext on ARM, kernel\_entry on MIPS, etc.). Look at the linker script to know the entry point for your platform. This function normally resides in the arch/<name>/kernel/head.S file. This function does the following.

* On machines that do not have the MMU turned on, this turns on the MMU. Most of the boot loaders do not work with the MMU so the virtual address equals the physical address. However, the kernel is compiled with the virtual address. This stub needs to turn on the MMU so that the kernel can start using the virtual address normally. This is not required on platforms such as MIPS where the MMU is turned on at power-on.
* Do cache initialization. This is again platform-dependent.
* Set up the BSS by zeroing it out (normally you cannot rely on the boot loader to do this).
* Set up the stack so that the first C routine can be invoked. The first C routine is the start\_kernel() function in init/main.c. This function is a jumbo function that does a lot of things until it terminates in an idle task (the first task in the system having a process id of 0). This function invokes the rest of the platform initialization functions that are discussed below.

*The setup\_arch() function*

This function does the platform- and CPU specific initialization so that the rest of the initialization can be invoked safely. Again this is highly platform-specific; only the common functionalities are explained:

* Recognizing the processor. Because a CPU architecture can come in various flavors, this function recognizes the processor (such as, if you have selected the ARM processor this finds out the ARM flavor) using hardware or information that may be passed at the time of building. Again any processor-specific fixups can be done in this code.
* Recognizing the board. Again because the kernel supports a variety of boards this option recognizes the board and does the board-specific fixups.
* Analysis of command-line parameters passed to the kernel.
* Identifying the ram disk if it has been set up by the boot loader so that the kernel later can mount it as the root file system. Normally the boot loader passes the starting of the ram disk area in memory and size.
* Calling the *bootmem* functions. Bootmem is a misnomer; it refers to the initial memory that the kernel can reserve for various purposes before the paging code grabs all the memory. For example, you can reserve a portion of a contiguous large memory that can be used for DMA by your device by calling the bootmem allocator.
* Calling the paging initialization function, which takes the rest of the memory for setting up pages for the system.

*Initialization of exceptions — the* trap\_init() *function:*

This function sets the kernel-specified exception handlers. Prior to this if an exception happens, the outcome is platform-specific. (For example, on some platforms the boot loader-specified exception handlers get invoked.)

*Initialization of interrupt handling procedure — the* init\_IRQ() *function:*

This function initializes the interrupt controller and the interrupt descriptors (these are data structures that are used by the BSP to route interrupts; more of this in the next chapter). Note that interrupts are not

enabled at this point; this is the responsibility of the individual, drivers owning the interrupt lines to enable them during their initialization which is called later. (For example, the timer initialization would make sure that the timer interrupt line is enabled.)

*Initialization of timers — the* time\_init() *function*

This function initializes the timer tick hardware so that the system starts producing the periodic tick, which is the system heartbeat.

*Initialization of the console—the* console\_init() *function*

This function does the initialization of the serial device as a console. Once the console is up, all the start-up messages appear on the screen. To print a message from the kernel, the printk() function has to be used. (printk() is a very powerful function as it can be called from anywhere, even from interrupt handlers.)

*Calculating the delay loops for the platform — the* calibrate\_delay() *function*

This function is used to implement microdelays within the kernel using the udelay() function. The udelay() function spins for a few cycles for the microseconds specified as the argument. For udelay to

work, the number of clock cycles per microsecond needs to be known by the kernel. This is exactly done by this function; it calibrates the number of delay loops. This makes sure that the delay loops work uniformly across all platforms. Note that the working of this depends on the timer interrupt.

***Subsystem Initialization***

This includes

* Scheduler initialization
* Memory manager initialization
* VFS initialization

Note that most of the subsystem initialization is done in the start\_kernel() function. At the end of this function, the kernel creates another process, the init process, to do the rest of the initialization (driver initialization, initcalls, mounting the root file system, and jumping to user space) and the current process becomes the idle process with process id of 0.

***Driver Initialization***

The driver initialization is done after the process and memory management is up. It gets done in the context of the init process.

***Mounting Root File System***

Recall that the root file system is the master file system using which other file systems can be mounted. Its mounting marks an important process in the booting stage as the kernel can start its transition to user space. The block device holding the root file system can be hard-coded in the kernel (while building the kernel) or it can be passed as a command line argument from the boot loader using the boot loader tag “root=”.

There are three kinds of root file systems that are normally used on embedded systems:

* The initial ram disk
* Network-based file system using NFS
* Flash-based file system

Note that the NFS-based root file system is mainly used for debugging builds; the other two are used for production builds. The ram disk simulates a block device using the system memory; hence it can be used to mount file systems provided a file system image is copied onto it. The ram disk can be used as a root file system; this usage of the ram disk is known as *initrd* (short form for initial ram disk). Initrd is a very powerful concept and has wide uses especially in the initial parts of embedded Linux development when you do not have a flash driver ready but your applications are ready for testing (often this is the case when you have a driver and a separate application team working in parallel). So how do you proceed without a flash-based root file system? You can use a network-based file system provided your network driver

is ready; if not, the best alternative is the initrd.

**How kernel mounts initrd**

If you want the kernel to load an initrd, you should configure the kernel during the build process with the

CONFIG\_BVLK\_DEV\_INITRD option. As previously explained, the initrd image is loaded along with the kernel image and the kernel needs to be passed the starting address and ending address of the initrd using command line arguments. Once it is known, the kernel will mount a root file system loaded on

initrd. The file systems normally used are romfs and ext2 file systems.

There is more magic to initrd. Initrd is a use-and-throw root file system. It can be used to mount another root file system. Why is this necessary? Assume that your root file system is mounted on a storage device whose driver is a kernel module. So it needs to be present on a file system. This presents a chicken-and-egg problem; the module needs to be on a file system, which in turn requires that the module be loaded first. To circumvent this, the initrd can be used. The driver can be made as a module in the initrd; once the initrd is mounted then the driver module can be loaded and hence the storage device can be accessed. Then the file system on that storage device can be mounted as the actual root file system and finally the initrd can be discarded. The Linux kernel provides a way for this use-and-throw facility; it detects a file linuxrc in the root of the initrd and executes it. If this binary returns, then the kernel assumes that initrd is no longer necessary and it switches to the actual root file system (the file linuxrc can be used to load the driver modules).

If the root file system is not mounted, the kernel will stall execution and enter the panic mode after logging the complaint on the console:

Unable to mount root fs on device

***Doing Initcall and Freeing Initial Memory***

If you open the linker script for any architecture, it will have an init section. The start of this section is marked using \_\_init\_begin and the end is marked using \_\_init\_end. The idea of this section is that it contains text and data that can be thrown away after they are used once during the system start-up.

Driver initialization functions are an example of the use-and-throw function. Once a driver that is statically linked to the kernel does its registration and initialization, that function will not be invoked again and hence it can be thrown away. The idea behind putting all such functions together is that the

entire memory occupied by all such functions can be freed as a big chunk and hence will be available for the memory manager as free pages. Considering that memory is a scarce resource on the embedded systems, the reader is advised to use this concept effectively. A use-and-throw function or variable is declared using the \_\_init directive. Once all the driver and subsystem initialization is done, the start-up code frees all the memory. This is done just before moving to user space.

Linux also provides a way of grouping functions that should be called at system start-up time. This can be done by declaring the function with the \_\_initcall directive. These functions are automatically called during kernel start-up, so you need not insert them into system start-up code.

***Moving to User Space***

The kernel that is executing in the context of the init process jumps to the user space by overlaying itself (using execve) with the executable image of a special program also referred to as init. This executable normally resides in the root file system in the directory /sbin. Note that the user can specify the init program using a command line argument to the kernel. However, if the kernel is unable to load either the user-specified init program or the default one, it enters the panic state after logging the complaint:

No init found. Try passing init= option to the kernel.

***User Space Initialization***

User space initialization is distribution dependent. The responsibility of then kernel ends with the transition to the init process. What the init process does and how it starts the services is dependent on the distribution. We now study the generic model on Linux (which assumes that the init process is

/sbin/init); the generic model is pretty similar to the initialization sequence of a UNIX variant, System V UNIX.

*The* /sbin/init *Process and* /etc/inittab .The init process is a very special process to the kernel; it has the following capabilities.It can never be killed. Linux offers a signal called SIGKILL that can terminate execution of any process but it cannot kill the init process.

When a process starts another process, the latter becomes the child of the former. This parent–child relationship is important. In case the parent dies before the child then init adopts the orphaned processes.

The kernel informs the init of special events using signals. For example: if you press the Ctrl-Alt-Del on your system keyboard, this makes the kernel send a signal to the init process, which typically does a

system shutdown.

The init process can be configured on any system using the inittab file, which typically resides in the /etc directory. init reads the inittab file and does the actions accordingly in a sequential manner. init also decides the system state known as *run level*. A run level is a number that is passed as an argument to init. In case none is passed the default run level can be picked up by init from the inittab file. The following are the run levels that are used.

* 0 – Halt the system
* 1 – Single-user mode (used for administrative purposes)
* 2 – Multi-user mode with restricted networking capabilities
* 3 – Full multi-user mode
* 4 – Unused
* 5 – Graphics mode (X11™)
* 6 – Reboot state

The inittab file has a special format. It generally has the following details. (Please refer to the main page of inittab on your system for more information.)

* The default run level.
* The actions to be taken when init is moved to a run level. Typically a script /etc/rc.d/rc is invoked with the run level as the argument.
* The process that needs to be executed during system start-up. This is typically the file /etc/rc.d/rc.sysinit file.
* init can respawn a process if it is so configured in the inittab file.
* This feature is used for respawning the log-in process after a user has logged out from his previous log-in.
* Actions to trap special events such as Ctrl-Alt\_Del or power failure.

*The rc.sysinit File*

This file does the system initialization before the services are started. Typically this file does the following on an embedded system.

* Mount special file systems such as proc, ramfs
* Create directories and links if necessary
* Set the hostname for the system
* Set up networking configuration on the system

*Starting Services*

As mentioned above, the script /etc/rc.d/rc is responsible for starting the services. A service is defined as a facility to control a system process. Using services, a process can be stopped, restarted, and its status can be queried. The services are normally organized into directories based on the run levels; depending on what run level is chosen the services are stopped or started. After performing the above steps, the init starts a log-in program on a TTY or runs a window manager on the graphics display (depending on the run level).

Note:For clear explanation please refer notes