PROJECT REPORT
CSE 598: EMBEDDED SYSTEMS PROGRAMMING
RUSHANG V KARIA
ASU # 1206337661

BOOTING BOOTLOADERS AND THE LINUX BOOT PROCESS

BOOTING BASICS

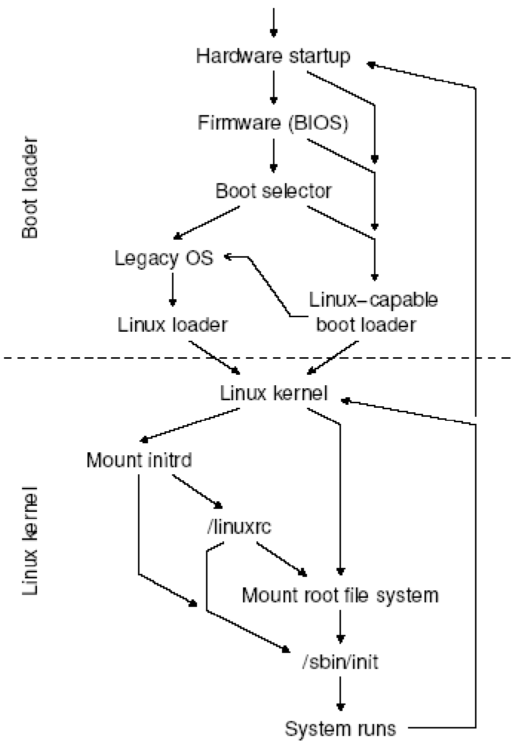
The computer world has been governed by de facto standards. This may be considered a feature since it enables all the “successful” technologies to live in harmony with each other. But there come some severe limitations by adopting such an approach.

The boot process is one that falls under such restrictions. Booting is the process of loading the operating system (kernel) into memory and passing control to it.

$$\begin{tikzcd}\boxed{\text{BIOS}} \arrow[r] & \boxed{\text{BootLoader}} \arrow[r] & \boxed{\text{Bare Kernel}} \arrow[r] & \boxed{\text{Extra functionality}}\end{tikzcd}$$

WHY IS BOOTING REQUIRED

Computers use RAM which are non-volatile data, so once powered off the memory loses its meaning. When powered on again, the kernel cannot start.



WHY A BOOT LOADER AND NOT BIOS??

There are many reasons

1. If you need to reprogram the bootloader you need to reprogram the BIOS (not easy)
2. The bootloader would need to know the OS specific features (B in BIOS=BASIC)
3. The boot process is divided into stages with each stage only having required knowledge, this makes writing new code and making portability simpler.

When the computer is powered up, the BIOS loads the first sector of ROM into memory (de facto). This standard is due to the good old days of floppy drives which were of size of mutiples of 512 bytes.

Surely an operating system (atleast a modern one) cannot fit in 512 bytes of memory. Thus a boot loader gives the operating system a hand by loading the required part of it into memory.

A special flag 0x55AA written at the last 2 bytes of a sector (typically 512 bytes of memory) characterizes it as a boot sector.

dd if=/dev/sda of=mbr.bin bs=512 count 1 <use this command to see the contents of your MBR>

It is this boot loader that loads the kernel into memory, passes control to it and then the kernel loads the rest of the functionality. Since a boot loader resides in the first sector of say your harddisk, it has to share space with the MBR (Master boot Record) which contains information about the partitions on the disk.

Typically MBR's are 70-100 bytes long and the rest of the space is given to the boot loader. Ever wondered why you cannot create many partitions on your hard drive??

BOOTING IN EMBEDDED SYSTEMS

Most Embedded systems require very fast booting. As such it is not feasible to have a boot loader load the system. Instead the embedded system is itself built to be directly bootable. Imagine waiting for a microwave to start 1 minute after you switch it on.

However large embedded systems often need an environment set and a boot loader becomes much useful at this point. Certain memory tests and other factors which do not require kernel functionality can be implemented in a boot loader.

OTHER TYPES OF BOOTING

There is a concept called chain loading which many boot-loaders like GRUB offer. Chain-loading simply means calling another boot loader from within a bootloader. Many times an operating system comes with a pre-loaded bootloader which has special environment variables, for this OS to run correctly its boot loader should run. So boot-loaders offer the functionality to call other bootloaders.

A network boot is one which typically does not require a bootloader. It is the job of the BIOS to support the protocol of doing the required loading.

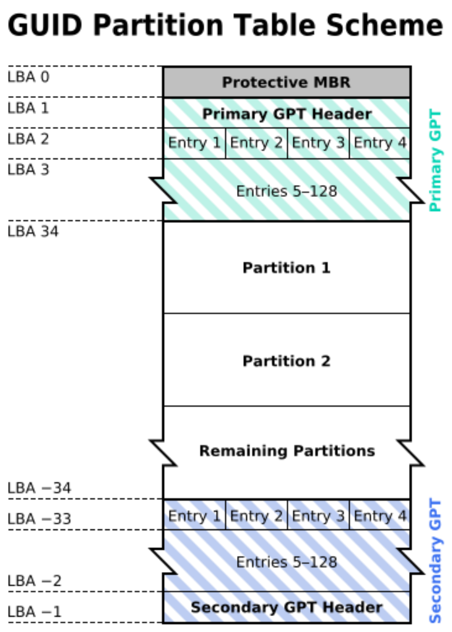
TYPES OF BOOTLOADERS

There are many types of bootloaders. Simple ones are the ones which may be used in embedded systems. These boot loaders are typically very small in size and do not have much overhead. For more complicated systems the concept of multi-stage boot loaders comes up. These are boot loaders whose functions are divided into stages. Normally the first stage executes first and loads the 2nd stage, in such a way more complicated boot loaders can be created. Multi-boot loaders also exist, these are bootloaders capable of booting more than one operating system.

UEFI – THE NEWEST BOOT SPECIFICATIONS

The Unified Extensible Firmware Interface (UEFI) is a software interface between the firmware and the Operating System. Its impetus was to provide the capability to boot harddisks which were bigger than 2TB, but it has evolved a lot since then. It now aims to remove the need for BIOS drivers for hardware but has so far been unsuccessful. It requires minimum configuration until the operating system selection phase.

UEFI replaces the MBR used to partition tables, UEFI defines a GUID (GUID Partition Table), this allows to boot hard disks as big as 9 ZetaBytes of data. UEFI pretty much changes booting in many ways.



The GUID allows for legacy MBR's to be booted but now the MBR is made to be a shadowing MBR. This essentially means that for disk sizes that are less than 2TB any external tool accessing the disk will see it as a single MBR with no data. This gives an advantage over several malware tools but severely curtails several beneficial software also.

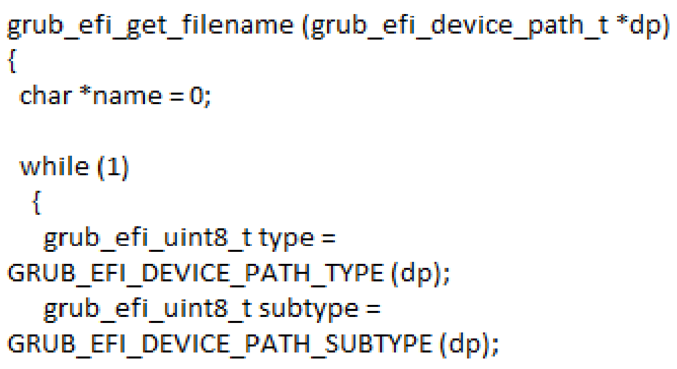
The use of UEFI will enable sectors of 4096 bytes which will significantly enhance the performance.

Entries 5-128 → the disk can contain upto 128 partitions.

UEFI does not require a boot sector. It has the capability to automatically search the drive for a boot loader. Typically a UEFI based boot loader often has a fixed path.

UEFI requires a lot of variables to store the boot time configuration and typically stores them in ROM for faster access.

Here is a snippet of code that shows that GRUB probes for the efi filename. It runs in an infinite loop until the file has been determined to be found or not found.



```
grub_efi_get_filename (grub_efi_device_path_t *dp)
{
 char *name = 0;

 while (1)
  {
   grub_efi_uint8_t type =
GRUB_EFI_DEVICE_PATH_TYPE (dp);
   grub_efi_uint8_t subtype =
GRUB_EFI_DEVICE_PATH_SUBTYPE (dp);
```

THE WINDOWS UEFI DEBATE

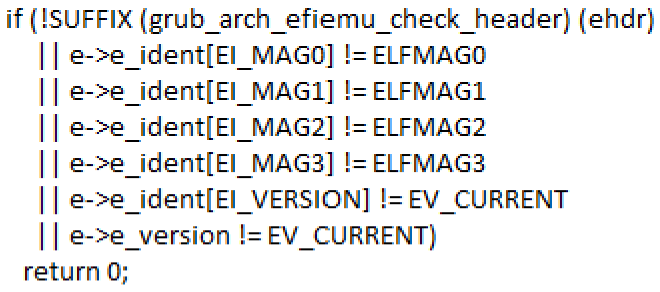
UEFI enables secure boot so that malware cannot infect the boot sector. But this has created several complications. One of them being Microsoft requiring all Windows machines be enabled with secure boot via a Microsoft private key. This makes the installation of Linux difficult and may even render a system useless.

GRUB [x86]

How does it load a sector into memory?

```
        movb    $0x42, %ah
        int     $0x13
```

Grub uses the BIOS interrupt 0x13 to load a sector into memory, AH contains the flag 0x42 to signify the load operation. DI contains the start of the sector, SI typically contains the length.

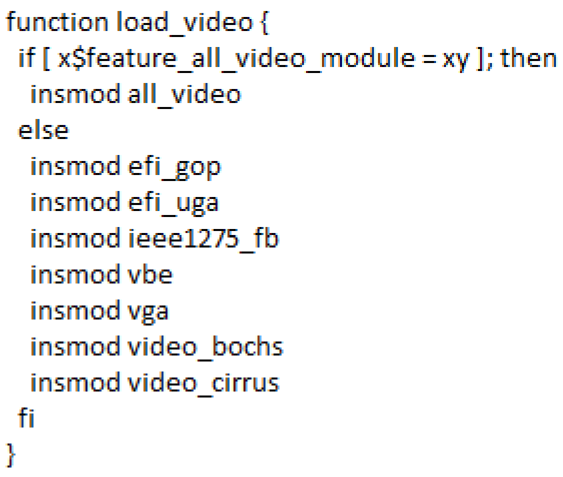


```
if (!SUFFIX (grub_arch_efiemu_check_header) (ehdr)
    || e->e_ident[EI_MAG0] != ELFMAG0
    || e->e_ident[EI_MAG1] != ELFMAG1
    || e->e_ident[EI_MAG2] != ELFMAG2
    || e->e_ident[EI_MAG3] != ELFMAG3
    || e->e_ident[EI_VERSION] != EV_CURRENT
    || e->e_version != EV_CURRENT)
  return 0;
```

GRUB stores the kernel address and other parameters into variables.

GRUB_BOOT_MACHINE_KERNEL_ADDR → stores the directory where the compressed kernel is found

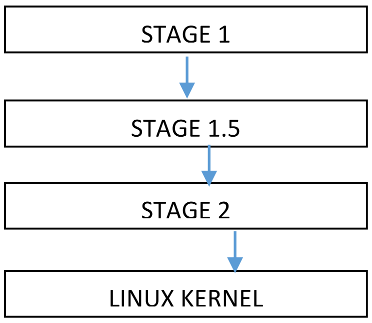
GRUB provides a rich environment to program. It employs a modular structure. Most of the functions as the one shown in the figure as similar to their UNIX variants. GRUB defines its own C library which is intermixed with assembly code to execute the functions. GRUB also supports plug and play and with GRUB 2 being rolled in, it can be used to perform special recovery through the command line interface.



```
function load_video {
 if [ x$feature_all_video_module = xy ]; then
  insmod all_video
 else
  insmod efi_gop
  insmod efi_uga
  insmod ieee1275_fb
  insmod vbe
  insmod vga
  insmod video_bochs
  insmod video_cirrus
 fi
}
```

BOOTING WITH GRUB

Grub typically requires 2 stages but there might be specific hardware where an additional stage is required. Stage 1.5 is often provided to allow file system like access to read the grub.conf file.



The first stage of GRUB is the minimalistic stage. It resides in the MBR and its

Only job is to load stage 1.5. Stage 1.5 sets up the necessary file systems and loads the configuration files. Finally stage 2 loads the linux kernel into memory.

Stage 1.5 is also responsible for displaying the GUI and enabling the command line interface. The command line interface acts as a medium to provide additional run time parameters. One example where they are useful are when GRUB is used to fix a system. Suppose a driver does not work, you can simply disable the module using GRUB command lines.

stage 1 = boot.img
stage 1.5= diskboot.img+kernel.img+pc.mod+ext2.mod (the core image)
stage 2 = normal.mod+_chain.mod

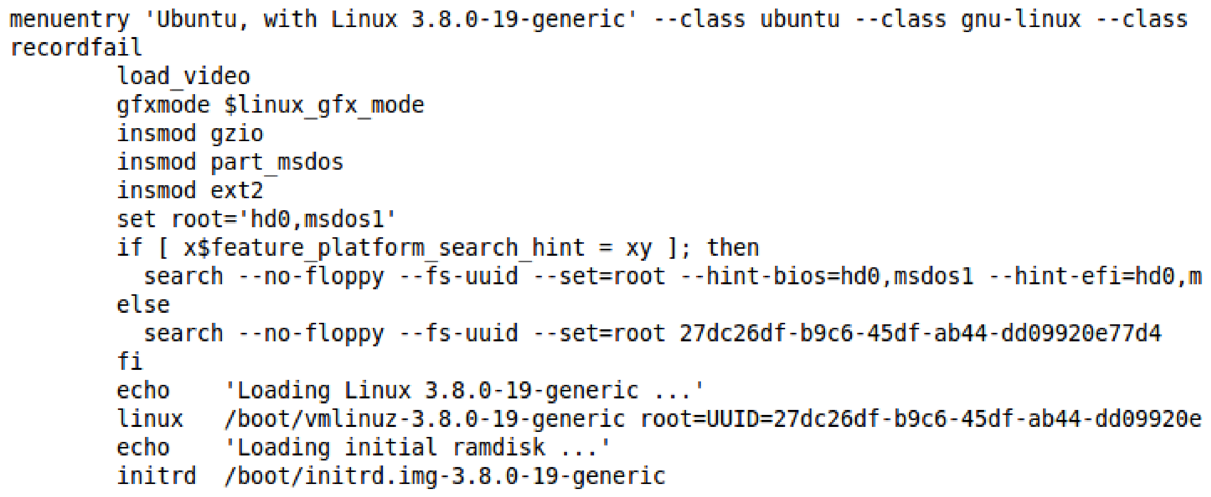
STAGE 2

Stage 2 contains important functionality of GRUB. One question that comes to mind is how does GRUB know which Operating System to boot. Since GRUB is a multi-boot loader, it is reasonable to ask this question. GRUB has defined configuration files named grub.conf, these files are read and the parameters are extracted. It is this stage which provides the user with the GUI.

A grub.conf file is like a script in GRUB language. It contains several parameters that a programmer can change thus giving it powerful flexibility. This file is located in /boot/grub/.

GRUB also contains a file called /boot/grub/gfxblacklist.txt. This file is mainly used to recover a system, GRUB disables the functioning of these modules during system boot. So typically modules which crash the system are typically recorded here till the problem is solved.

/boot/grub/fonts and /boot/grub/<MACHINE>-PC are font and module directories.



```
menuentry 'Ubuntu, with Linux 3.8.0-19-generic' --class ubuntu --class gnu-linux --class
recordfail
        load_video
        gfxmode $linux_gfx_mode
        insmod gzio
        insmod part_msdos
        insmod ext2
        set root='hd0,msdos1'
        if [ x$feature_platform_search_hint = xy ]; then
          search --no-floppy --fs-uuid --set=root --hint-bios=hd0,msdos1 --hint-efi=hd0,m
        else
          search --no-floppy --fs-uuid --set=root 27dc26df-b9c6-45df-ab44-dd09920e77d4
        fi
        echo    'Loading Linux 3.8.0-19-generic ...'
        linux   /boot/vmlinuz-3.8.0-19-generic root=UUID=27dc26df-b9c6-45df-ab44-dd09920e
        echo    'Loading initial ramdisk ...'
        initrd  /boot/initrd.img-3.8.0-19-generic
```

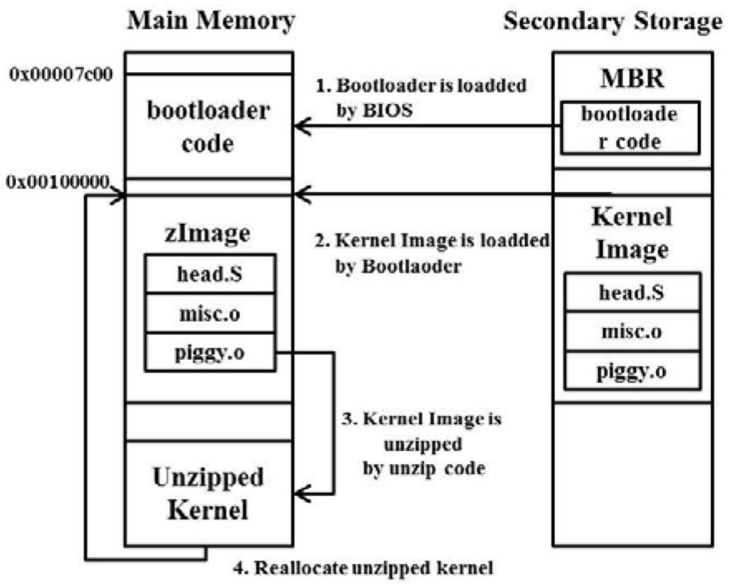
This is what a typical GRUB configuration file looks like.

Menuentry	→ what title the user will be shown
-class	→ hints to GRUB about the Operating System (this help GRUB perform some optimizations)
Recordfail	→ to record a failure to be used for debugging purposes.
Insmod	→ insert modules
Set root	→ set it as root harddisk. Operating System resides here.
Search	→ search the file /boot for the operating system image
Initrd	→ initialize a root file system in RAM so that the kernel has simplified tasks. (since it need not bother with low level RAM anymore)

GRUB	**LILO**
Grub has an interactive shell	Lilo does not have a shell
Grub does not need to be reconfigured if its configuration file changes	Lilo needs to be reconfigure if its configuration file is changed
GRUB provides a lot of flexibility and support	Lilo does not offer much flexibility
GRUB is 3-stage	Lilo is 2 stage

THE LINUX BOOT PROCESS

The Linux boot process is the next step after the GRUB stage 2. GRUB stage 2 decompresses the linux kernel. The Linux kernel is usually kept in compressed form so that it does not occupy contiguous sectors in memory. Decompressing usually provides low overhead on the slow hard drives. A compressed kernel has minimum functionality so the first thing that it does once loaded is decompress itself.



The kernel resides in high memory. It typically sets up the hardware before moving on to more specific things. When uncompressed ./arch/i386/boot/head.S is the assembly language routing that is executed.

This code resides at absolute address 0x90200 and initializes video.S Also setup.S is run. This routing requests the BIOS for important parameters. At this point it switches to protected mode. Linux does not reply on BIOS interrupts and this is the last point where it will request the BIOS for information.

0x9000-0x901FF → critical BIOS environment.

Setup.S does many things. It initializes the hardware, sets the keyboard mouse and many other paramters.

First JMP 0x9200

MOV DS,0x9020 →this is to find the setup code, If found the next step is executed

MOV DS,INITSEG → whose value is 0x9000. This is the area in memory where Linux is loaded.

FOR ARM PROCESSOR

```
        /* Jump to relocated kernel */
        mov lr,r1
        mov r0,#0
        ldr r1,kexec_mach_type
        ldr r2,kexec_boot_atags
ARM(    mov pc, lr        )
THUMB(  bx lr             )
```

/arch/i386/boot/compressed/head.S is the next routine to be executed and this causes the kernel to get decompressed. Finally startup_32() is the function that is called. There are two startup_32 routines but these do not cause any conflicts since they are absolute address programs.

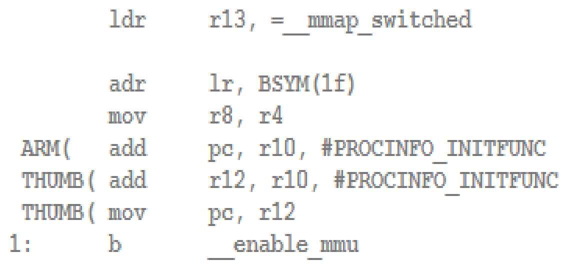
For ARM processors, the kernel changes to ARM mode. It then stores the PIC address and jumps to that address. Start_up 32() is called.

```
          __HEAD
ENTRY(stext)

 THUMB( adr      r9, BSYM(1f)     )
 THUMB( bx       r9               )
 THUMB( .thumb                    )
 THUMB(1:                         )
```

This process is called process 0 and it sets up memory tables and other important utilities. Once set up the kernel will never refer to physical level addresses and a virtual memory addressing will be employed.

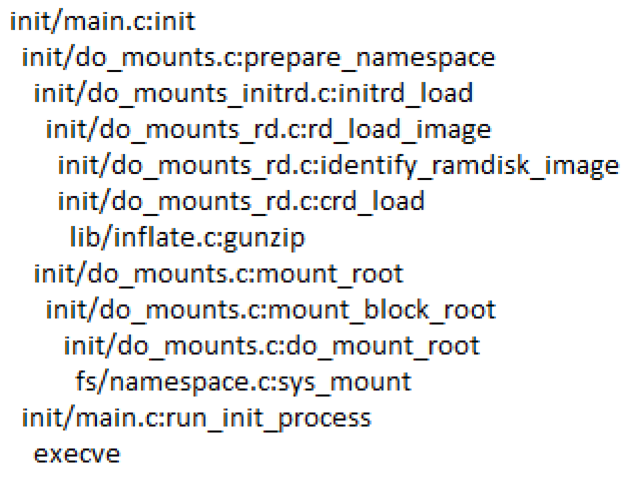
At this point there are two images of the kernel in memory. One is the compressed one which is specified by initrd in the GRUB configuration file and the decompressed kernel. The decompressed kernel saves the state of the virtual file system and then deletes the compressed image.



```
        ldr     r13, =__mmap_switched

        adr     lr, BSYM(1f)
        mov     r8, r4
 ARM(   add     pc, r10, #PROCINFO_INITFUNC
 THUMB( add     r12, r10, #PROCINFO_INITFUNC
 THUMB( mov     pc, r12
1:      b       __enable_mmu
```

All the previous initialization was relatively low level. Now the first code executing in C language is encountered. The first user space process with PID=1 resides in /sbin/init. Main.c contains the startk_kernel function which essentially is the main() function for Linux. It sets up interrupts and other context level activities and finally makes a call to kernel_thread residing in process.c to set up user space processes. At this point, the kernel call idle_task which makes the kernel go in a dormant state and provides services only when requested. Populating /etc/fstab and /etc/mtab (linux partition information) is also done here.

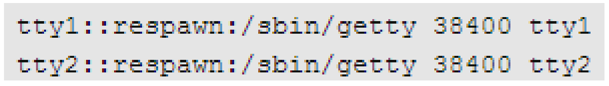


```
init/main.c:init
 init/do_mounts.c:prepare_namespace
  init/do_mounts_initrd.c:initrd_load
   init/do_mounts_rd.c:rd_load_image
    init/do_mounts_rd.c:identify_ramdisk_image
    init/do_mounts_rd.c:crd_load
     lib/inflate.c:gunzip
  init/do_mounts.c:mount_root
   init/do_mounts.c:mount_block_root
    init/do_mounts.c:do_mount_root
     fs/namespace.c:sys_mount
 init/main.c:run_init_process
  execve
```

But yet the work is not done, there are many more activities to be done. /etc/inittab gives an idea of the run levels of the process. All these run level programs are stored in /etc/rc.d/rc<level>.d. Depending on the run-level these scripts are executed. Each of these files contain script files beginning with S<number> or K<number>, S is used when system starts and vice versa. The number is used so that the scripts are executed in a chronological sequence.

Next configuration files kept in /etc/inittab and init.d are executed as startup applications. Finally Linux is available for log in.

id:rstate:action:process



```
tty1::respawn:/sbin/getty 38400 tty1
tty2::respawn:/sbin/getty 38400 tty2
```

Main.c further calls sched.c and other functions to ready the user space environment. Linux is deemed ready to boot from user processes when do_basic_setup() is called. One of the most important functions that this function performs is to load the driver model. It does this by making a call to the function devices_init().

AKNOWLEDGEMENTS

I would like thank Dr. Yann-Hang Lee for providing me with this opportunity. I would also like to thank the Teaching Assistants Young Song and Duo Lo for all their help.

CONCLUSION

Bootloaders continue to play a vital role in operating systems. The increasing of role of embedded systems will cause a change in the architecture of the boot loaders. Conventional boot loaders are considered too slow for embedded systems. Current boot loaders for embedded systems need massive improvements. Meanwhile it is not surprising if new methods into the Linux boot process are introduced. The ever increasing size and complexity of the Linux kernel will place new demands on the Linux boot process. As Linux becomes more ubiquitous appearing in mobile devices, embedded systems which require a very small booting time, this will give a further impetus to research in booting Linux.

References

[1] http://en.wikipedia.org/wiki/Booting

[2]http://www.intel.com/content/dam/doc/reference-guide/efi-compatibility-support-module-specification-v097.pdf

[3] BOOTING LINUX FASTER Dokeun Lee, Youjip Won Proceedings ofIC-NIDC2012

[4] http://unixbhaskar.wordpress.com/2010/03/19/insight-into-gnulinux-boot-process/

[5] http://www.linuxfromscratch.org/lfs/view/development/chapter06/grub.html

[6] http://docs.oracle.com/cd/E19683-01/817-3814/6mjcp0qgh/index.html