# Learning Report – Embedded Linux and Kernel -PROGRAMMING

L&T Technology Services

GLOBAL ENGINEERING ACADEMY

Genesis

## Document History

| Ver. Rel. No. | Release Date | Prepared. By | Reviewed By | To be approved By | Remarks/Revision Details |
|---|---|---|---|---|---|
| 1 | | Name/PS No | Name/PS No | Module Owner Name | Comments |
| 2 | 12/04/2021 | Abhishek Mishra<br>99003738 | | | |
| | | | | | |
| | | | | | |
| | | | | | |

Table of Content

## Embedded Linux

Now days learning Embedded Systems and related technology is very essential for the engineers, who's interest involved in the sector of automation and electronic gadgets. We have Microprocessors and Microcontrollers for doing multiple task at a time as well as for doing single task at a time. Learning these technology in less time is always a big challenge for new comer in this area , and also it is quite confusing between the selection of resources and where to start learning , when there are so many developments boards like MCU(AVR, Atmel), MPU (ARM, PowerPC, MIPS) are available in market. This project is a small step in the direction of understanding the requirements and building the process to approach universally, which can apply to major development boards. Embedded Linux is also a Linux but it is different with normal Linux in terms of uses and operations. The main difference between the normal Linux and the embedded Linux are: Embedded Linux are designed for doing Real time operation or a specific set of task but normal Linux cannot do the Real time operation. Normal Linux run on the generic hardware, whereas embedded Linux run on embedded hardware. Embedded Linux required RAM and ROM (memory footprint) but normal Linux doesn't care of it. We designed embedded Linux to consume less power as compared to normal Linux. Embedded Linux also has low latency as compared to normal Linux

- Linux license which is used is constant.
- Linux license used in the embedded field offers flexibility to user
- Linux license can be used by user according to application required to observe and change the supply code
- Linux license offers exchange of original code and the changes made to the other environment.

**Why Linux**?

As a result of the few financial and specialized favorable circumstances, we're seeing strong development inside the selection of Linux for installed contraptions. This pattern has crossed really all business sectors and innovation Linux has been followed for implanted product inside the worldwide open exchanged telephone arrange, worldwide records systems, wi-fi cell handsets, and the gear that works those systems. Linux has delighted in achievement in car programs, supporter stock which incorporates games and PDAs, printers, association switches and switches, and a lot of other product. The selection cost of inserted Linux keeps on developing, and not utilizing an outcome in sight.A portion of the intentions in the blast of inserted Linux are as per the following:

• Linux has developed as a develop, over the top execution, stable option in contrast to customary exclusive inserted running structures.
• Linux helps a major kind of utilizations and systems administration conventions.
• Linux is adaptable, from little customer orientated gadgets to large, overwhelming iron, administration superbness switches and switches.
• Linux can be sent without the eminences required by utilizing customary exclusive implanted working frameworks.
• Linux has pulled in a major amount of dynamic engineers, empowering expedient guide of most recent H/W models, structures, and contraptions.
• An expanding number of H/W and programming organizations, including actually all the top-level makers and ISUs, presently help Linux.
People using those and various thought processes, we are seeing a quickened appropriation expense of Linux in bunches of ordinary family unit things, beginning from over the top definition TVs to portable handsets.

**Linux Kernel**

Linux started in 1990, while Linos Torvalds initiated composing a working contraption for Intelcorp 386and 485-essentially base PCs. He become animated by method for the Minux working contraption composed by methods for Andrew T. Taninbaum four years ahead of time. Linux differentiated in various procedures from Minux; the rule assortments that it changed into a 31-piece virtul part and the code was upper dot, later released underneath the GPT v2 license. He reported it on 24Aug, 1990, on the com.Os.Minux newsgroups in well known present that started with: Hey everyone open the use of Minux—I am doing a (detache) working machine (only a side interest, probably enormous and master GMU) 385(485) AT duplicate. This has mix because of the reality Apr, and was starting to get prepared. I would like a few input on things individuals like/loathe in Minux, as my OS takes after it fairly (same substantial configuration of the files (because of down to earth thought processes) in addition to other things).

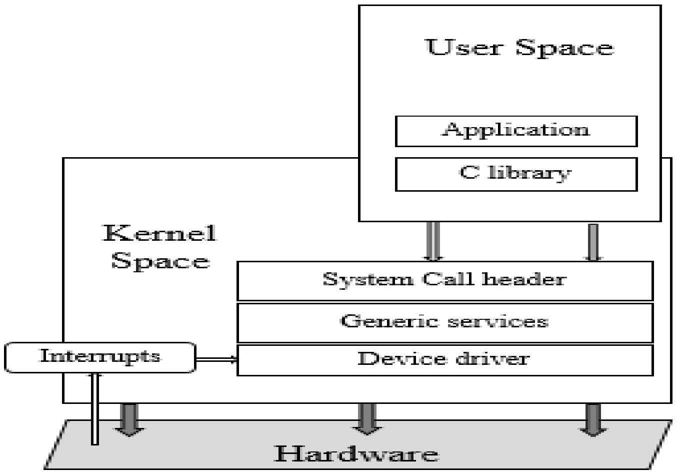
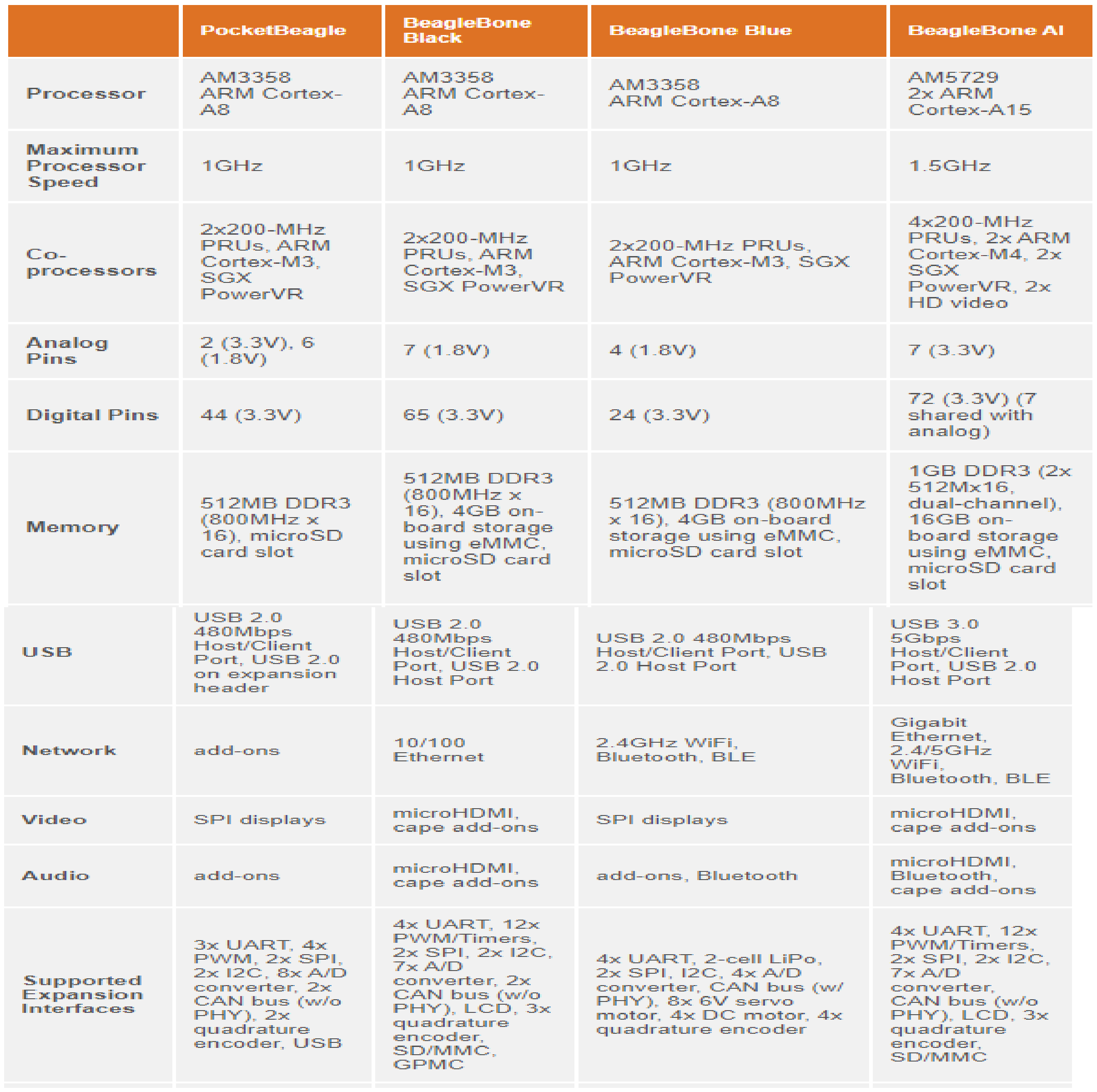


Figure *1* Architecture (Kernel space vs User space)
Apps strolling in User domain work at a less CPU benefit stage. They can do practically zero beside make lib. The main involvement among domain and the Krnel zone is the C-program lib, which interprets client level capacities, for example, the ones characterized by methods for POSIX, into piece machine calls. The framework call interface utilizes a design extraordinary method, together with a snare or a product program intrude, to supplant the CPU from low benefit client mode to over the top benefit bit mode, which grants get right of section to all memory locations and CPU reg. The device calling mechanism makes call to the fitting piece module which is part of sub-system: memory task calls go to the memory director, archives calls to the records code, and so fort .

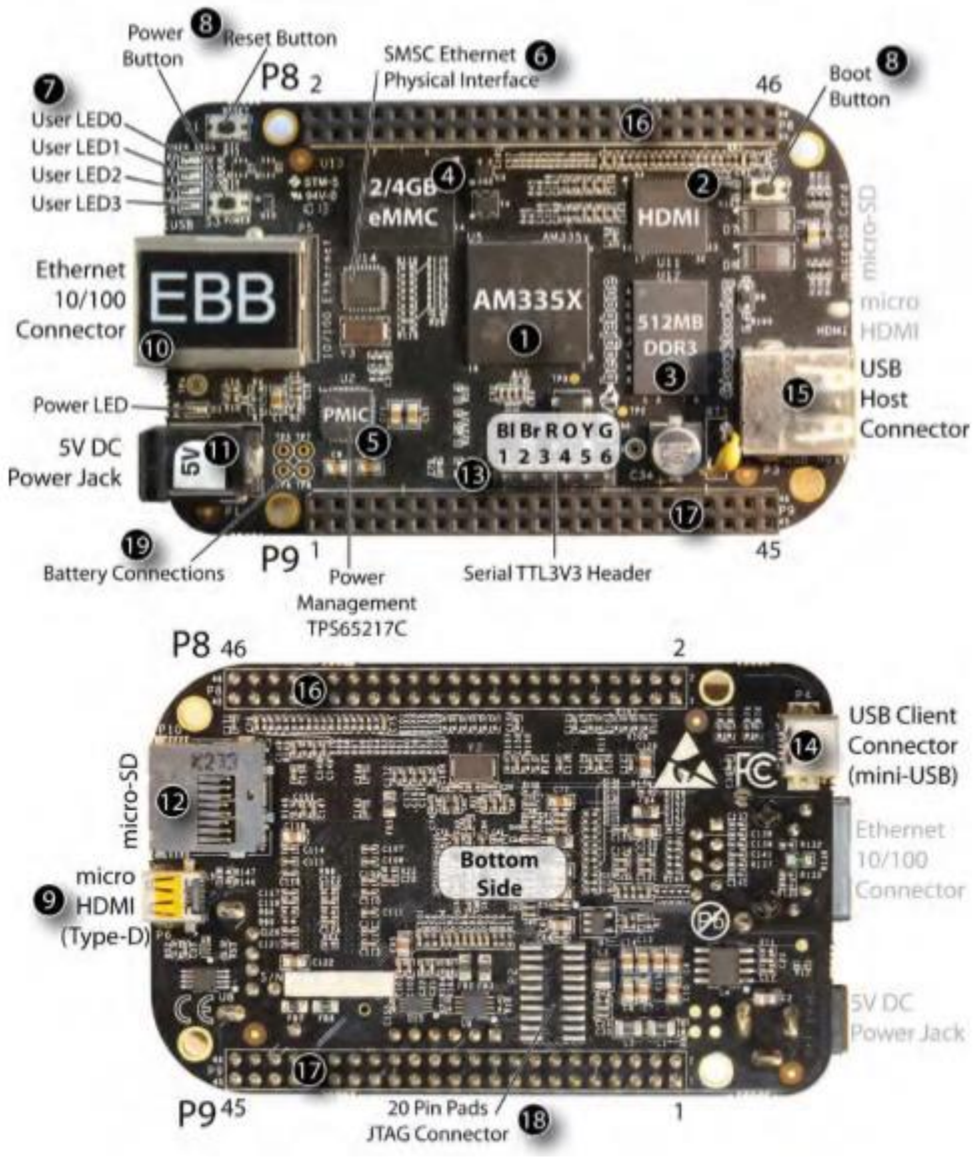
**ARM Based Board overview**

**Comparison between AM335X**



| | PocketBeagle | BeagleBone Black | BeagleBone Blue | BeagleBone AI |
|---|---|---|---|---|
| Processor | AM3358 ARM Cortex-A8 | AM3358 ARM Cortex-A8 | AM3358 ARM Cortex-A8 | AM5729 2x ARM Cortex-A15 |
| Maximum Processor Speed | 1GHz | 1GHz | 1GHz | 1.5GHz |
| Co-processors | 2x200-MHz PRUs, ARM Cortex-M3, SGX PowerVR | 2x200-MHz PRUs, ARM Cortex-M3, SGX PowerVR | 2x200-MHz PRUs, ARM Cortex-M3, SGX PowerVR | 4x200-MHz PRUs, 2x ARM Cortex-M4, 2x SGX PowerVR, 2x HD video |
| Analog Pins | 2 (3.3V), 6 (1.8V) | 7 (1.8V) | 4 (1.8V) | 7 (3.3V) |
| Digital Pins | 44 (3.3V) | 65 (3.3V) | 24 (3.3V) | 72 (3.3V) (7 shared with analog) |
| Memory | 512MB DDR3 (800MHz x 16), microSD card slot | 512MB DDR3 (800MHz x 16), 4GB on-board storage using eMMC, microSD card slot | 512MB DDR3 (800MHz x 16), 4GB on-board storage using eMMC, microSD card slot | 1GB DDR3 (2x 512Mx16, dual-channel), 16GB on-board storage using eMMC, microSD card slot |
| USB | USB 2.0 480Mbps Host/Client Port, USB 2.0 on expansion header | USB 2.0 480Mbps Host/Client Port, USB 2.0 Host Port | USB 2.0 480Mbps Host/Client Port, USB 2.0 Host Port | USB 3.0 5Gbps Host/Client Port, USB 2.0 Host Port |
| Network | add-ons | 10/100 Ethernet | 2.4GHz WiFi, Bluetooth, BLE | Gigabit Ethernet, 2.4/5GHz WiFi, Bluetooth, BLE |
| Video | SPI displays | microHDMI, cape add-ons | SPI displays | microHDMI, cape add-ons |
| Audio | add-ons | microHDMI, cape add-ons | add-ons, Bluetooth | microHDMI, Bluetooth, cape add-ons |
| Supported Expansion Interfaces | 3x UART, 4x PWM, 2x SPI, 2x I2C, 8x A/D converter, 2x CAN bus (w/o PHY), 2x quadrature encoder, USB | 4x UART, 12x PWM/Timers, 2x SPI, 2x I2C, 7x A/D converter, 2x CAN bus (w/o PHY), LCD, 3x quadrature encoder, SD/MMC, GPMC | 4x UART, 2-cell LiPo, 2x SPI, I2C, 4x A/D converter, CAN bus (w/ PHY), 8x 6V servo motor, 4x DC motor, 4x quadrature encoder | 4x UART, 12x PWM/Timers, 2x SPI, 2x I2C, 7x A/D converter, CAN bus (w/o PHY), LCD, 3x quadrature encoder, SD/MMC |

## The BeagleBone Black Hardware



It is a RISC (reduced instruction set computing) processor, so at 1,000 MHz the processor executes 2,000 million instructions per second (MIPS). The processor runs at about 1 W idle and 2.3 W for heavy processing loads.

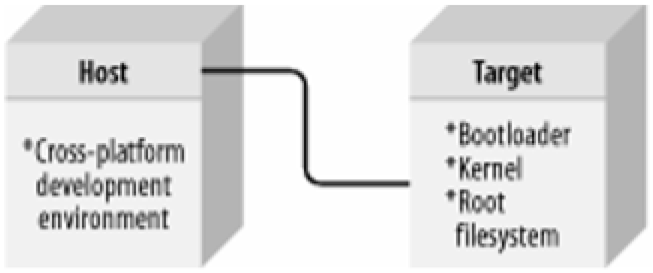
The next set of callouts, 9 to 19, identifies the various connectors on the BBB, their physical characteristics, and their function. For connector 18, the JTAG connector, there are 20 pre-tinned pads. You need to purchase a connector (such as Samtec FTR-110-03-G-D-06) for this and carefully solder it to the board. In addition, you need a JTAG interface and associated debug software. The BBW has on-board USB JTAG support.

**Types of Host and Target Development Setup**

Three different host/target architectures are available for the development of embedded Linux systems: the linked setup, the removable storage setup, and the standalone setup. Your actual setup may belong to more than one category or may even change categories over time, depending on your requirements and development methodology

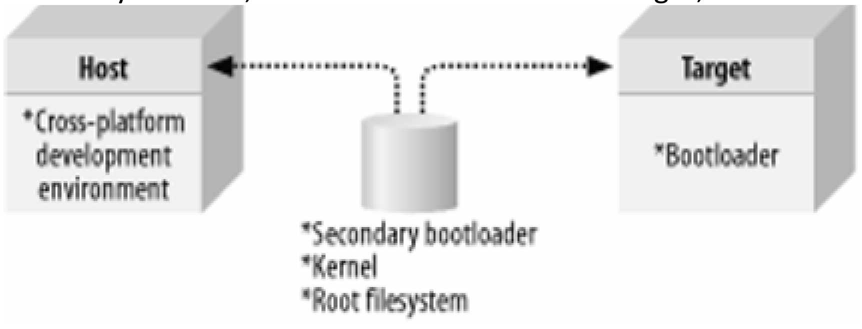
**Linked Setup**

In this setup, the target and the host are permanently linked together using a physical cable. This link is typically a serial cable or an Ethernet link. The main property of this setup is that no physical hardware storage device is being transferred between the target and the host. All transfers occur via the link.



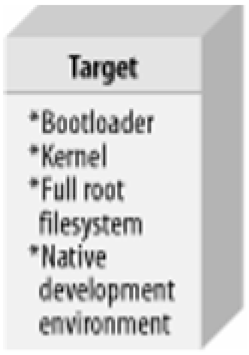
**Removable Storage Setup**

In this setup, there are no direct physical links between the host and the target. Instead, a storage device is written by the host, is then transferred into the target, and is used to boot the device.

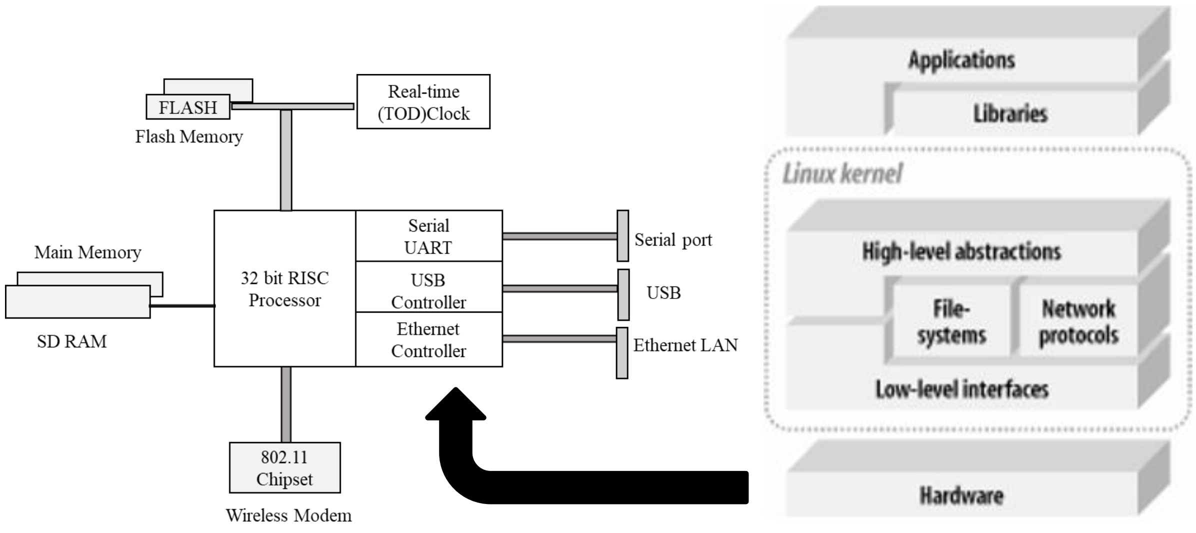


***Standalone Setup***

Here, the target is a self-contained development system and includes all the required software to boot, operate, and develop additional software. In essence, this setup is similar to an actual workstation, except the underlying hardware is not a conventional workstation but rather the embedded system itself.



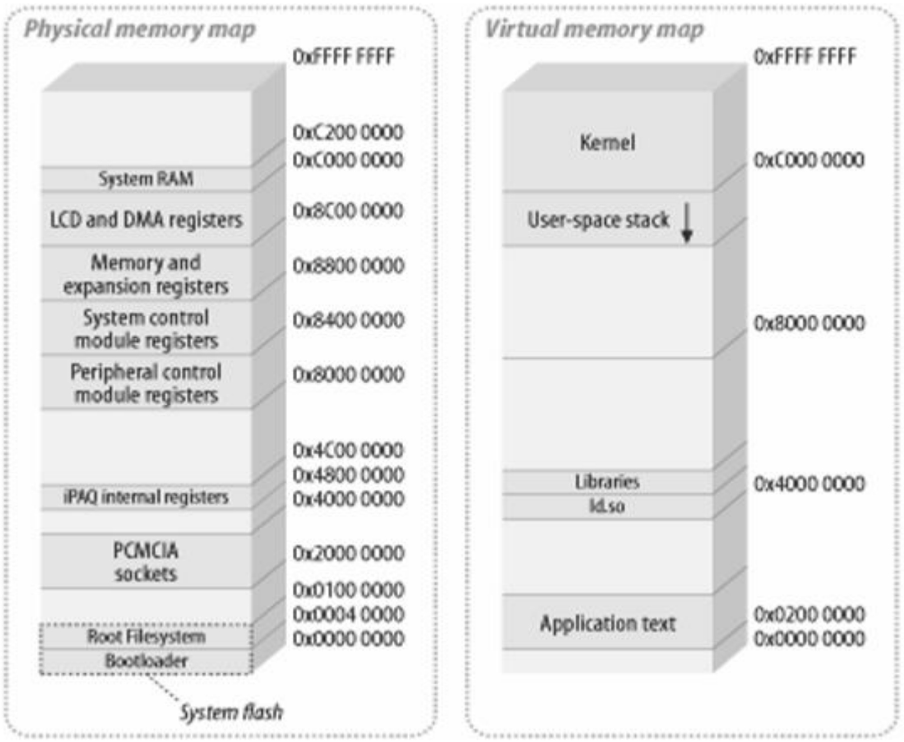
**Generic Architecture of an Embedded Linux System**



There are some broad characteristics expected from the hardware to run a Linux system. First, Linux requires at least a 32-bit CPU containing a memory management unit (MMU). Second, a sufficient amount of RAM must be available to accommodate the system. Third, minimal I/O capabilities are required if any development is to be carried out on the target with reasonable debugging facilities. This is also very important for any later troubleshooting in the field. Finally, the kernel must be able to load and/or access a root filesystem through some form of permanent or networked storage.

**System Memory Layout**

To best use the available resources, it is important to understand the system's memory layout, and the differences between the physical address space and the kernel's virtual address space. Most importantly, many hardware peripherals are accessible within the system's physical address space, but have restricted access or are completely "invisible" in the virtual address space.



<u>**The four elements of embedded Linux**</u>

Every project begins by obtaining, customizing, and deploying these four elements: the toolchain, the bootloader, the kernel, and the root filesystem.

**Toolchain:** The compiler and other tools needed to create code for your target device. Everything else depends on the toolchain.
**Bootloader:** The program that initializes the board and loads the Linux kernel. Kernel: This is the heart of the system, managing system resources and interfacing with hardware.
**Root filesystem:** Contains the libraries and programs that are run once the kernel has completed its initialization.

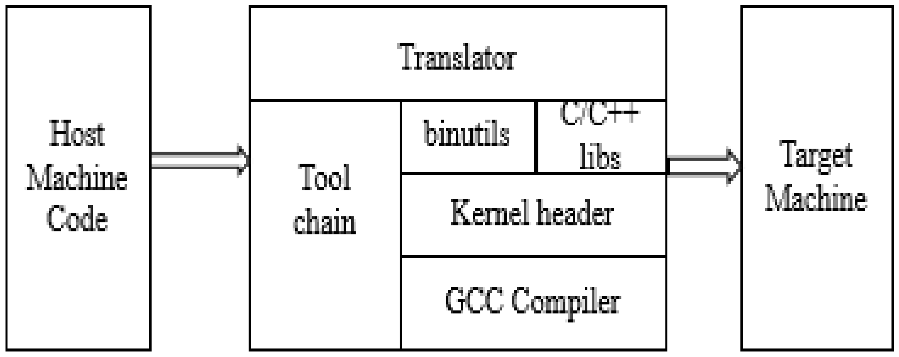
Of course, there is also a fifth element, not mentioned here. That is the collection of programs specific to your embedded application which make the device do whatever it is supposed to do, be it weigh groceries, display movies, control a robot, or fly a drone.

## Toolchain

A toolchain is the set of tools that compiles source code into executables that can run on your target device, and includes a compiler, a linker, and run-time libraries. Initially you need one to build the other three elements of an embedded Linux system: the bootloader, the kernel, and the root filesystem. It has to be able to compile code written in assembly, C, and C++ since these are the languages used in the base open source packages.

***<u>A standard GNU toolchain consists of three main components:</u>***

- *Binutils:* A set of binary utilities including the assembler and the linker. It is available at http://www.gnu.org/software/binutils .
- *GNU Compiler Collection (GCC):* These are the compilers for C and other languages which, depending on the version of GCC, include C++, Objective-C, Objective-C++, Java, Fortran, Ada, and Go. They all use a common backend which produces assembler code, which is fed to the GNU assembler. It is available at http://gcc.gnu.org/.
- *C library:* A standardized application program interface (API) based on the POSIX specification, which is the main interface to the operating system kernel for applications. There are several C libraries to consider, as we shall see later on in this



## Types of toolchains

For our purposes, there are two types of toolchain:

**Native:**

This toolchain runs on the same type of system (sometimes the same actual system) as the programs it generates. This is the usual case for desktops and servers, and it is becoming popular on certain classes of embedded devices. The Raspberry Pi running Debian for ARM, for example, has self-hosted native compilers.

**Cross:**

This toolchain runs on a different type of system than the target, allowing the development to be done on a fast desktop PC and then loaded onto the embedded target for testing.

**Bootloader**

The bootloader is the second element of embedded Linux. It is the part that starts the system up and loads the operating system kernel. In this chapter, I will look at the role of the bootloader and, in particular, how it passes control from itself to the kernel using a data structure called a device tree, also known as a flattened device tree or FDT. I will cover the basics of device trees, so that you will be able to follow the connections described in a device tree and relate it to real hardware.

**Function of Bootloader**

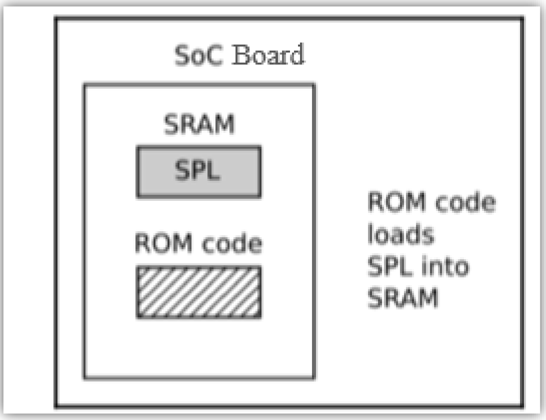
In an embedded Linux system, the bootloader has two main jobs: to initialize the system to a basic level and to load the kernel. In fact, the first job is somewhat subsidiary to the second, in that it is only necessary to get as much of the system working as is needed to load the kernel.

When the first lines of the bootloader code are executed, following a power-on or a reset, the system is in a very minimal state. The DRAM controller would not have been set up, and so the main memory would not be accessible. Likewise, other interfaces would not have been configured, so storage accessed via NAND flash controllers, MMC controllers, and so on, would also not be usable. Typically, the only resources operational at the beginning are a single CPU core and some on-chip static memory. As a result, system bootstrap consists of several phases of code, each bringing more of the system into operation. The final act of the bootloader is to load the kernel into RAM and create an execution environment for it. The details of the interface between the bootloader and the kernel are architecture-specific, but in each case it has to do two things. First, bootloader has to pass a pointer to a structure containing information about the hardware configuration, and second it has to pass a pointer to the kernel command line. The kernel command line is a text string that controls the behavior of Linux. Once the kernel has begun executing, the bootloader is no longer needed and all the memory it was using can be reclaimed.

**The boot sequence**

**Part 1 – ROM code**

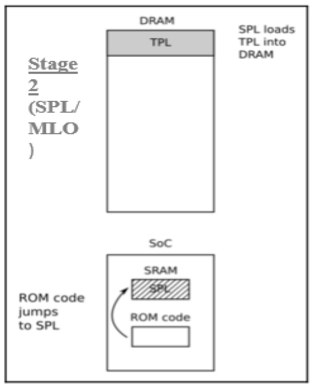
In the absence of reliable external memory, the code that runs immediately after a reset or power-on has to be stored on-chip in the SoC; this is known as ROM code. It is loaded into the chip when it is manufactured, and hence the ROM code is proprietary and cannot be replaced by an open source equivalent. Usually, it does not include code to initialize the memory controller, since DRAM configurations are highly device-specific, and so it can only use Static Random Access Memory (SRAM), which does not require a memory controller.



The ROM code is capable of loading a small chunk of code from one of several preprogrammed locations into the SRAM. As an example, TI OMAP and Sitara chips try to load code from the first few pages of NAND flash memory, or from flash memory connected through a Serial Peripheral Interface (SPI), or from the first sectors of an MMC device (which could be an eMMC chip or an SD card), or from a file named MLO on the first partition of an MMC device. If reading from all of these memory devices fails, then it tries reading a byte stream from Ethernet, USB, or UART; the latter is provided mainly as a means of loading code into flash memory during production, rather than for use in normal operation. Most embedded SoCs have a ROM code that works in a similar way. In SoCs where the SRAM is not large enough to load a full bootloader like U-Boot, there has to be an intermediate loader called the secondary program loader, or SPL

**Part 2 – Secondary program loader**

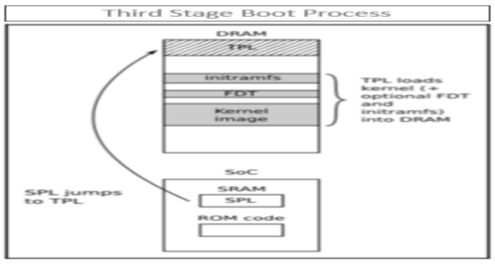
The SPL must set up the memory controller and other essential parts of the system preparatory to loading the Tertiary Program Loader (TPL) into DRAM. The functionality of the SPL is limited by the size of the SRAM. It can read a program from a list of storage devices, as can the ROM code, once again using pre-programmed offsets from the start of a flash device. If the SPL has file system drivers built in, it can read well known file names, such as u-boot.img, from a disk partition. The SPL usually doesn't allow for any user interaction, but it may print version information and progress messages, which you can see on the console. The following diagram explains the phase 2 architecture:



The SPL may be open source, as is the case with the TI x-loader and Atmel AT91Bootstrap, but it is quite common for it to contain proprietary code that is supplied by the manufacturer as a binary blob.

**Part 3- TPL**

Now, at last, we are running a full bootloader, such as U-Boot. Usually, there is a simple command-line user interface that lets you perform maintenance tasks, such as loading new boot and kernel images into flash storage, and loading and booting a kernel, and there is a way to load the kernel automatically without user intervention.
The following diagram explains the phase 3 architecture:



At the end of the third phase, there is a kernel in memory, waiting to be started. Embedded bootloaders usually disappear from memory once the kernel is running, and perform no further part in the operation of the system.

**Moving from bootloader to Linux kernel**

When the bootloader passes control to the kernel it has to pass some basic information, which may include some of the following:

- *The machine number*, which is used on PowerPC, and ARM platforms without support for a device tree, to identify the type of the SoC .
- *Basic details of the hardware detected so far*, including at least the size and location of the physical RAM, and the CPU clock speed
- The kernel command line .

- Optionally, the location and size of a device tree binary.
- Optionally, the location and size of an initial RAM disk, called the initial RAM file system (initramfs).

**Introducing device trees**

A device tree is a flexible way to define the hardware components of a computer system. Usually, the device tree is loaded by the bootloader and passed to the kernel, although it is possible to bundle the device tree with the kernel image itself to cater for bootloaders that are not capable of loading them separately.

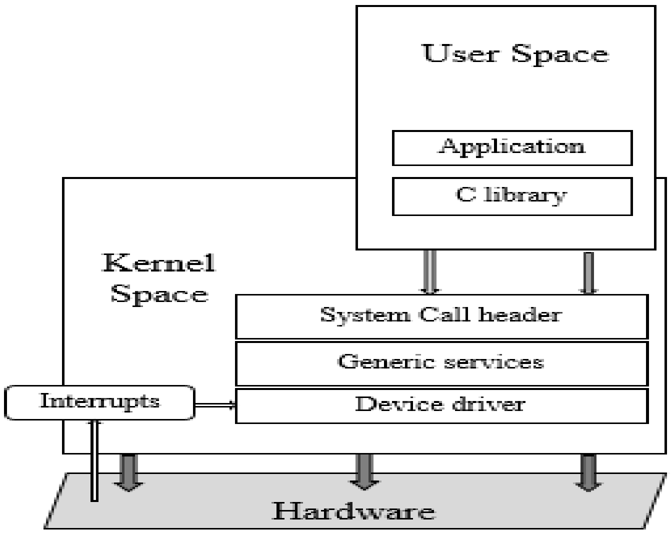
**Device tree basics**

The Linux kernel contains a large number of device tree source files in arch/$ARCH/boot/dts, and this is a good starting point for learning about device trees. There are also a smaller number of sources in the U-boot source code in arch/$ARCH/dts. If you acquired your hardware from a third party, the dts file forms part of the board support package and you should expect to receive one along with the other source files. The device tree represents a computer system as a collection of components joined together in a hierarchy, like a tree. The device tree begins with a root node, represented by a forward slash, /, which contains subsequent nodes representing the hardware of the system. Each node has a name and contains a number of properties in the form name = "value". Here is a simple example:

```
/dts-v1/; /{    model = "TI AM335x  BeagleBone";
compatible = "ti,am33xx";    #address-cells = <1>;
 #size-cells = <1>;
cpus {       #address-cells = <1>;
  #size-cells = <0>;
cpu@0
{          compatible = "arm,cortex-a8";
 device_type = "cpu";
 reg = <0>;        };
  };
 memory@0x80000000  {       device_type = "memory";
       reg = <0x80000000   0x20000000>;
/* 512 MB */    };
};
```

Linux Kernel
The kernel is the third element of embedded Linux. It is the component that is responsible for managing resources and interfacing with hardware, and so affects almost every aspect of your final software build. It is usually tailored to your particular hardware configuration. All About Bootloaders, device trees allow you to create a generic kernel that is tailored to particular hardware by the contents of the device tree.

The kernel has three main jobs: to manage resources, to interface with hardware, and to provide an API that offers a useful level of abstraction to user space programs, as summarized in the following diagram: Applications running in User space run at a low CPU privilege level. They can do very little other than make library calls. The primary interface between the User space and the Kernel space is the C library, which translates user level functions, such as those defined by POSIX, into kernel system calls. The system call interface uses an architecture-specific method, such as a trap or a software interrupt, to switch the CPU from low privilege user mode to high privilege kernel mode, which allows access to all memory addresses and CPU registers.



The System call handler dispatches the call to the appropriate kernel subsystem: memory allocation calls go to the memory manager, filesystem calls to the filesystem code, and so on. Some of those calls require input from the underlying hardware and will be passed down to a device driver. In some cases, the hardware itself invokes a kernel function by raising an interrupt.
*Note : The preceding diagram shows that there is a second entry point into kernel code: hardware interrupts. Interrupts can only be handled in a device driver, never by a user space application.*

**Kernel command line**

The kernel command line is a string that is passed to the kernel by the bootloader, via the bootargs variable in the case of U-Boot; it can also be defined in the device tree, or set as part of the kernel configuration in CONFIG_CMDLINE.

- Debug- Sets the console log level to the highest level, 8, to ensure that you see all the kernel messages on the console.
- init= The init program to run from a mounted root filesystem, which defaults to /sbin/init.
- Lpj = Sets loops_per_jiffy to a given constant. There is a description of the significance of this in the paragraph following this table.

- Panic = Behavior when the kernel panics: if it is greater than zero, it gives the number of seconds before rebooting; if it is zero, it waits forever (this is the default); or if it is less than zero, it reboots without any delay.
- quiet - Sets the console log level to , suppressing all but emergency messages. Since most devices have a serial console, it takes time to output all those strings. Consequently, reducing the number of messages using this option reduces boot time. rdinit= The init program to run from a ramdisk. It defaults to /init.
- Ro- Mounts the root device as read-only. Has no effect on a ramdisk, which is always read/write.
- Root = Device to mount the root filesystem.
- Rootdelay = The number of seconds to wait before trying to mount the root device; defaults to zero. Useful if the device takes time to probe the hardware, but also see rootwait.
- Rootfstype = The filesystem type for the root device. In many cases, it is auto-detected during mount, but it is required for jffs2 filesystems.
- Rootwait - Waits indefinitely for the root device to be detected. Usually necessary with mmc devices.
- rw - Mounts the root device as read-write (default).

**Root File System**

The root filesystem is the fourth and the final element of embedded Linux. Once you have read this chapter, you will be able build, boot, and run a simple embedded Linux system.
The kernel will get a root filesystem, either an initramfs, passed as a pointer from the bootloader, or by mounting the block device given on the kernel command line by the root= parameter. Once it has a root filesystem, the kernel will execute the first program, by default named init, as described in the section Early user space above, Configuring and Building the Kernel. Then, as far as the kernel is concerned, its job is complete. It is up to the init program to begin starting other programs and so bring the system to life.

To make a minimal root filesystem, you need these components:

- init: This is the program that starts everything off, usually by running a series of scripts.
- Shell: You need a shell to give you a command prompt but, more importantly, also to run the shell scripts called by init and other programs.
- Daemons: A daemon is a background program that provides a service to others. Good examples are the system log daemon (syslogd) and the secure shell daemon (sshd). The init program must

start the initial population of daemons to support the main system applications. In fact, init is itself a daemon: it is the daemon that provides the service of launching other daemons.
- Shared libraries: Most programs are linked with shared libraries, and so they must be present in the root filesystem.
- Configuration files: The configuration for init and other daemons is stored in a series of text files, usually in the /etc directory. Device nodes: These are the special files that give access to various device drivers.
- /proc and /sys: These two pseudo filesystems represent kernel data structures as a hierarchy of directories and files. Many programs and library functions depend on proc and sys.
- Kernel modules: If you have configured some parts of your kernel to be modules, they need to be installed in the root filesystem, usually in /lib/modules/[kernel version].

**The directory layout**

Embedded devices tend to use a subset based on their needs, but it usually includes the following:
- /bin: Programs essential for all users.
- /dev: Device nodes and other special files.
- /etc: System configuration files.
- /lib: Essential shared libraries, for example, those that make up the C-library.
- /proc: The proc filesystem.
- /sbin: Programs essential to the system administrator .
- /sys: The sysfs filesystem .
- /tmp: A place to put temporary or volatile files .
- /usr: Additional programs, libraries, and system administrator utilities, in the directories /usr/bin, /usr/lib and /usr/sbin, respectively .
- /var: A hierarchy of files and directories that may be modified at runtime, for example, log messages, some of which must be retained after boot.

**The staging directory**

We should begin by creating a staging directory on your host computer where you can assemble the files that will eventually be transferred to the target. In the following examples.I have used ~/rootfs. We need to create a skeleton directory structure in it, for example, take a look here:

```
$ mkdir ~/rootfs
$ cd ~/rootfs
$ mkdir bin dev etc home lib proc sbin sys tmp usr var
$ mkdir usr/bin usr/lib usr/sbin
```

$ mkdir -p var/log
To see the directory hierarchy more clearly, you can use the handy tree command used in the following example with the -d option to show only the directories:
$ tree –d

```
├── bin
├── dev
├── etc
├── home
├── lib
├── proc
├── sbin
├── sys
├── tmp
├── usr
|    ├── bin
|    ├── lib
|    └── sbin
├── va
└── var
 └── log
```

**Compiling stages and generating useful software components:**

**QEMU INSTALLATION**

Command to install Qemu on Ubuntu is:-
->sudo apt install qemu-system-arm
(sudo keyword gives root user access to install any software on Ubuntu).
Tool-Chain required for Cross-Compilation ARM Architecture based Target Board.
Run Command
-> sudo apt install gcc-arm-linux-gnueabi (for soft float)
-> sudo apt install gcc-arm-linux-gnueabihf (for hard float)

## RUNNING QEMU

Pre-built vexpress-v2p-ca9.dtb and zImage(Kernel Image) given by faculty.

**Now building our own SD card.**

Steps to Copy Output files on the SD Card

- At beginning for the first time create a directory in root by using following commands.

->sudo mkdir /mnt/rootfs
->sudo mkdir /mnt/boot (copy vexpress-v2p-ca9.dtb and zImage here)

- Now everytime to copy new file on SD Card use following commands

->sudo mount -o loop,rw,sync rootfs.img /mnt/rootfs (To mount SD Card)
-> sudo cp a.out /mnt/rootfs/home/root (Run this command where outfile is generated)
->sudo umount rootfs.img /mnt/rootfs (To un-mount SD Card)

Download core image minimal qemuarm.ext4 from
http://downloads.yoctoproject.org/releases/yocto/yocto_2.5/machines/qemu/qemuarm/

Rename core image minimal qemuarm.ext4 as rootfs.img

**Now run these two commands**:-

- e2fsck -f rootfs.img (This command structures the rootfs.img file if it is disrupted).
- resize2fs rootfs.img 16M (This Command is used to resize rootfs.img file) .
- Now Keep rootfs.img , zImage, vexpress-v2p-ca9.dtb in the same folder run following command:-
- Emulation using Qemu- SD card approach
- ->qemu-system- arm -M vexpress-a9 -m 1024 serial stdio \
- -kernel zImage -dtb vexpress-v2p-ca9.dtb \
- -sd rootfs.img append "console=ttyAMA0 root=/dev/mmcblk0 rw"
- Emulation using Qemu-initrd approach
- ->qemu-system-arm -M vexpress-a9 -m 1024 -serial stdio \
- -kernel zImage -dtbvexpress-v2p-ca9.dtb \
- -initrd rootfs.img -append "console=ttyAMA0 root=/dev/ram0 rw"
- Now let’s understand this command :-
- qemu-system-arm is our emulator for ARM Architecture based machines.
- -M is specifies development board
- -vexpress-a9 (Virtual Express A9, ARM Cortex-A9 CPU)
- -m (memory of 1024 is allocated)

-

-serial stdio it redirects virtual qemu serial stdio
-Next we load Kernel as zImage and device tree as vexpress-v2p-ca9.dtb)
- now we load our sd card rootfs.img which contains output files to execute on qemu emulator
- append is there to specify current terminal which is of ttyAMA0 architecture.

**BUILDING CUSTOM UBOOT**

Dowload uboot version (u-boot-2020.10.tar.bz2) from ftp://ftp.denx.de/pub/u-boot/and extract, choose any stable.
Extract the file.

Run following command to build u-boot:-
->Build it as partition sd card as sdcard.img and copy uboot file from extracted file.
Now boot using sd card using as:-
-> qemu-system-arm -M vexpress-a9 -m 1024 -serial stdio -kernel u-boot -sd sdcard.img
Stop auto boot by hitting any key on u-boot shell –
Run following commands on u-boot shell->
mmcinfo fatls mmc 0:1
fatload mmc 0:1 0x60200000 zImage (loading zImage in fat file)
fatload mmc 0:1 0x60100000 vexpress-v2p-ca9.dtb (loading device tree in fat file)
setenv bootargs 'console=ttyAMA0 root=/dev/mmcblk0p2 rw rootfstype=ext4' (setting environment for booting)
bootz 0x60200000 - 0x60100000 (boot zImage and vexpress-v2p-ca9.dtb)

**CUSTOM KERNEL DEVELOPMENT**

- https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.14.202.tar.xz (Download this file)
- Now Extract this file manually or use following command
  -> tar –xvf linux-4.14.202.tar.xz
- Now put the extract file in KSRC.
- Run following commands:-

-> make ARCH=arm mrproper (this will clean the make file it will work as make clean)
-> make ARCH=arm vexpress_defconfig (It will deconfigure Virtual Express file)
-> make ARCH=arm menuconfig (It is to configure kernel)

Inside this a new window will be shown where perform some minimal changes as follows

1. General Setup -> Local Version -> "-custom“
2. Device Drivers -> Block Devices ->

**Enable RAM Block device support**

Increase default RAM disk size to suitable limit, say 65536

3. Enable the block layer

Support for large (2TB+)

(Tip – To Enable press-y , to disable press-n and to make it user controlled press-m)

- Now after customizing kernel run following command to build zImage

-> make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-zImage -j <no of cores based on system such as 2, 4,6> (You can get generated zImage in /arch/arm/boot/)

Now build device tree binaries

-> make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-dtbs firmware (You can get generated vexpress-v2p-ca9.dtb in /arch/arm/boot/dts/)

- Here kernel is generated in two forms compressed and un-compressed. Compressed version is zImage and uncompressed version is vmlinux file which is located in KSRC.
- Now we have our zImage and vexpress-v2p-ca9.dtb.Now we load our Qemu on our custom kernel successfully.

**BUILDING KERNEL MODULES**

- Create a folder in KSRC/ linux4.14.xxx /drivers/Char/htest add a simple hello source code file.
- Add makefile in same folder with entry obj += hello.o (only) (Note-> obj += means adding .o file with
- the existing o files).
- Modify makefile in KSRC/linux4.14.xxx/drivers/char as obj-m += htest/
- Modifying zImage adding our module through htest file.
- Now build new zImage and install new modules using following command
  - ➔ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- zImage modules
- Now mount SD card
- sudo make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- modules_install
- INSTALL_MOD_PATH=/mnt/rootfs (for install new modules in KSRC/linux4.14.xxx)
- Now unmount SD Card.
- Now run Qemu an test output.
- ->dmesg –c (clearing kernel log)

- ->insmod hello.ko (loading out-tree modules)
- ->cat /proc/modules(gives list of existing modules)

->dmesg
->rmmod hello (unloading module)
module_init (), module_exit defines function exclusively for initialization and exit.
GPL is a certification or License.
Module_param() function used to create multiple device having same module specification.
->insmod hello.ko ndevices=10 (module runs for 10 devices )
Module dependency can be created using EXPORT_SYMBOL_GPL() and extern. Loading parent module is must before loading it's dependent module otherwise program encounters a run-time error.

**CREATING PARTITIONED SD CARD**

• Run following commands :-
-> qemu-img create sdcard.img 128M
->cfdisk sdcard.img (this will now create partition in SD card) dos write-yes
->sudo fdisk -l sdcard.img (this will display the partition) 2048x512=1048576, 34816x512=17825792 ->sudo losetup -o 1048576 /dev/loop31 sdcard.img (attach partition 1)
->sudo losetup -o 17825792 /dev/loop32 sdcard.img(attach partition 2)
->sudo mkfs.vfat /dev/loop31 (kernel space partition)
->sudo mkfs.ext4 /dev/loop32 (user space partition)
->sudo mount -o loop,rw,sync /dev/loop31 /mnt/boot
->sudo mount -o loop,rw,sync /dev/loop32 /mnt/rootfs • Now copy zImage, vexpress-v2p-ca9.dtb to /mnt/boot
• Then extract core-image-minimal-qemuarm.tar.bz2 to /mnt/rootfs using command
->tar -jxvf core-image-minimal-qemuarm.tar.bz2 -C /mnt/rootfs
• ->sudo umount /mnt/boot
->sudo umount /mnt/rootfs
->sudo losetup -d /dev/loop31(detach partition)
->sudo losetup -d /dev/loop32(detach partition) (Note- check for mount no. of mounts, if there are more than 2 than unmount the device loop from outside mount (use command -> mount)).

**In-tree Modules (STATIC AND DYNAMIC)**

These modules can be loaded and unloaded anywhere while running.
To build this module create Kconfig in mtest folder(in drivers/char/mtest) as

```
config SIMPLE
tristate "Simple module"
default n
help A simple module
config SAMPLE
tristate "Sample module"
depends on SIMPLE
default n
endmenu
```

Now add following in char/Kconfig
->source “drivers/char/mtest/Kconfig”
Add following in char/Makefile
->obj-y += mtest/
Now check changes in ->make ARCH=arm menuconfig
Just enable any custom module and check the changes in .config file.
(Trick -> If you want have run a certain command and can’t able to access it using up-down arrow use following command ->history | grep <command name>)