LINUX EMBEDDED SYSTEM

Linux Embedded Program using QEMU

Prerequisites:

- It is expected you have fair idea of Linux based system and application of its basic commands
- Knowledge of C programming language is needed incase of system component or applications
- It is expected you have Host with good hardware configuration
    - CPU based on i5 or greater
    - 4 GB RAM
    - 10 GB of Disk Space
    - The preference would be a Ubuntu based Linux system (standalone or dual boot), and incase it's not available, it is expected that you have properly configured VM

1.0 Learning Outcome

On completion of this Course participants will exhibit competencies / Skills of:

- Getting familiar with Embedded Linux architecture.
- Getting familiar with user space environment of embedded Linux, cross compilation models
- Thorough understanding on Boot loaders and Linux boot sequence
- Practical knowledge on Device Tree concepts.
- Gain practical knowledge on peripheral interfacing from Linux user space

2.0 Lab Setup

This section will guide you to create your Project workspace along with the package installation used in this module.

2.1.1 **1. Workspace Creation Guidelines**

Please adhere to the following points to have a clean workspace so that it become easy to organize the required files

- Choose a clean directory for all the work
- You may choose directory like EmbeddedLinux / E-Linux / EOS or with any sensible name under home directory
- Don’t use Desktop, Downloads, Documents, Music, Videos, Pictures etc, which are meant for other purpose.
- Avoid spaces or special symbols in path names, the build may fail, it is very important

- Under this workspace keep different sub directories for downloaded packages, extracted source to build, configuration files, examples etc.

2.1.2 **2. Project Directory Creation**

- Assuming you are in you home directory, if not you may type

```
user@Host:~] cd # Changes directory to home directory
```

- Let's create a project directory, you may give any name but let it be meaningful

```
user@Host:~] mkdir EmbeddedLinux
user@Host:~] cd EmbedddedLinux
user@Host:EmbeddedLinux]
```

- Let's create all the major top-level directories that would be used will working on Embedded Linux development

```
user@Host:EmbeddedLinux] mkdir Sources # To store all the raw source files
user@Host:EmbeddedLinux] mkdir Linux # Directory to work with Linux kernel
user@Host:EmbeddedLinux] mkdir U-Boot # Directory to work with U-Boot
user@Host:EmbeddedLinux] mkdir RootFS # Directory to store root filesystem data
user@Host:EmbeddedLinux] mkdir Images # Directory to store all the built target images
```

2.1.3 **3. Package Installation**

While configuring and building the packages we will need some system packages, you can install them using the following command

```
user@Host:EmbeddedLinux] sudo apt install libncurses-dev flex bison libssl-dev
```

- Install ifconfig command as follows

```
user@Host:EmbeddedLinux] sudo apt install net-tools
```

3.0 Introduction

3.1.1 1. Overview of Embedded Systems

An embedded system is an end product that would be used for specific function, you may see different definition being used to describe it, some commonly used ones are put below

- A System designed to preform a Specific Task
- **Combination of *Hardware* and *Software*, designed to do a *dedicated* application**
- A system with Single function, where the complexity depends on the task

So, you may describe it the way you like as it doesn't a fixed definition, I have highlighted the second one since it highlights 3 major words that has to be kept in mind designing an Embedded System

3.1.2 1. Overview of Embedded Systems

3.1.2.1 1.1. Why Embedded Systems?

Below are some use cases that can make us think to go with embedded systems

- Any function which can be automated can be done with the help of Embedded System
- Can by used in instances where human presence is not possible, like for examples deep water explorations, space explorations to name some
- Increase in efficiency of the operation performed by the system

3.1.3 1. Overview of Embedded Systems

3.1.3.1 1.2. Examples of Embedded Systems

You can observe the below image for some example of embedded system which encounter at our home or office in day to day life

LINUX EMBEDDED SYSTEM

The complexity of the system depend on the task what you are planning to perform, or in other words the number of functions you are trying to expose to the user

For example if you consider the Air conditioner for the above diagram, it has limited function to perform, like you will be able to

- Start / stop
- Set the required temperature.

On the other hand if consider the a Smart TV, it would have multiple function or features to provide, like, providing different interfaces to get the data to be displayed, some examples, you may have

- USB port
- HDMI interface
- Setup Box Interface
- Wifi Connectivity
- Audio Interface
- Remote control interface to name some.

So the complexity depends on the product requirement you are dealing with.

3.1.4 **2. Why OS in an Embedded System**

One question that should come to your mind in general will design an embedded system is, does a Operating System (OS) really needed?

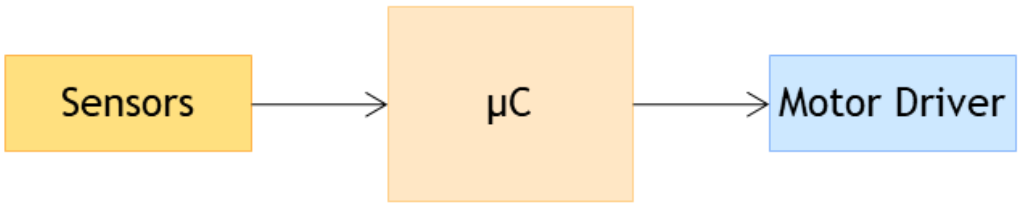
Because consider an OS just like that would a complexing to system unnecessarily.

LINUX EMBEDDED SYSTEM

So let's understand that aspect with the help of a simple use case.

Let's say you want to design an automatic door opening system as shown below

Now, if you consider the simplest possible design (without going in depth) you may majorly need the below components isn't it



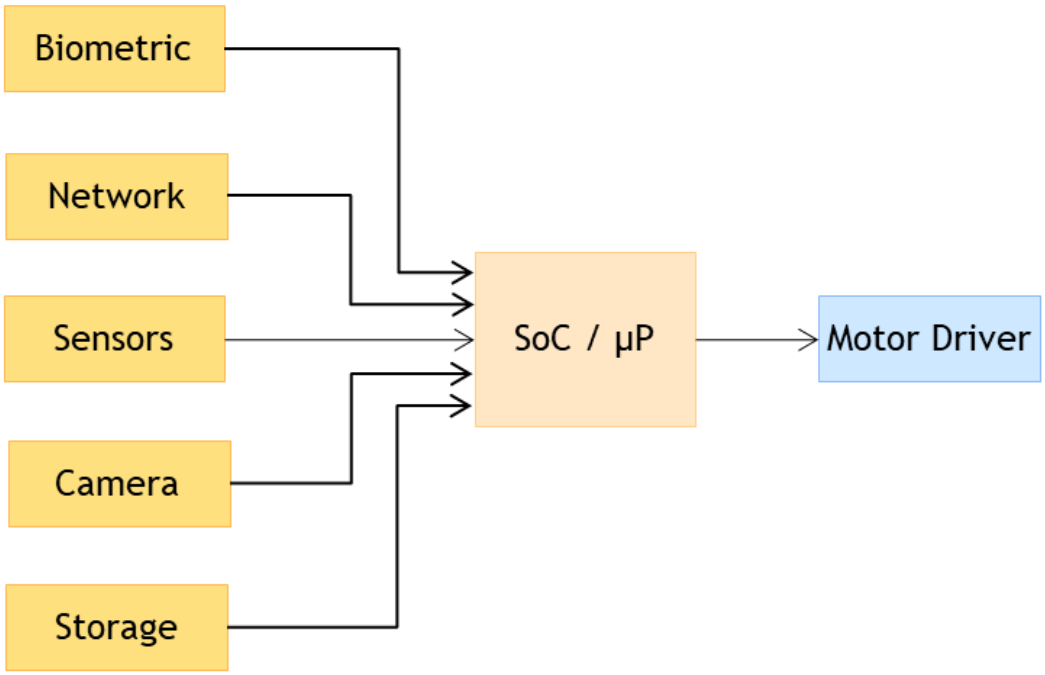
A sensor (say PIR), a Microcountroller and a Motor driver to open and close the door. The major requirement is captured. Lets not forget, the goal (specific task) is to open and close the door that's all.

Now, what if the requirement changes in such a way that you need to open the door by seeing who is at the door(assume an office environment where you will find 1000s of people)

Can the same design what we had seen above work? The answer is no right?

LINUX EMBEDDED SYSTEM

You may have to consider the following design



Why such a complex design?

- Well to see who is at the door we need a camera
- We will have to store all the images in the database that would be compared upon
- the processing power (to compare 1000s of images) should be really good hence we will have to consider a SoC or a microprocessor based design with high performance
- An employee may travel to other office location, then the data should be available in cloud, now you may wonder, why the local storage, what if the network is down, shouldn't the door open?
- Bypass system as backup with Biometric or general sensors to open the door

So these are some points to be considered isn't it, but the goal is the same to open the door, but we have lots of functions to be done

Hence this is the situation, going with bare metal programming approach will not suffice and the best pick is to go with a operating system considering the complexity of the requirement.

Hope the point is clear.

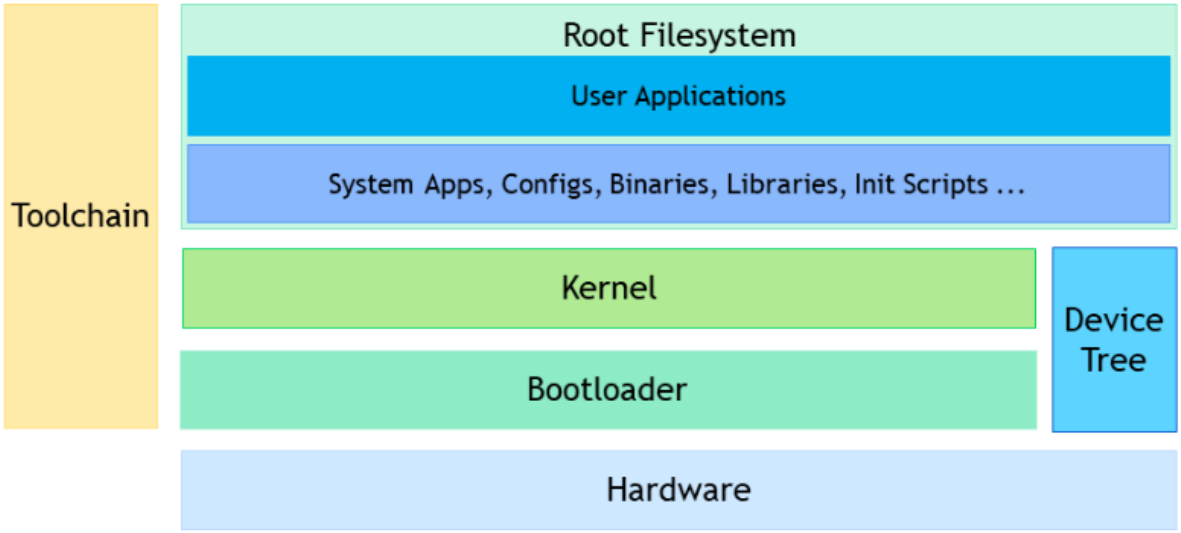
3.1.5 **3. Why Embedded Linux as OS?**

The following are some major points to pick Linux as an Embedded OS

- It is Open Source
  - Source (under GNU General Public License v2.0 : GPLv2)
  - Freedom to
    - Copy
    - Use
    - Modify
    - Redistribute
- Supports Multiple Architectures
  - x86, ARM, MIPs, PowerPC ..,
- Community Driven
  - Individuals, Academics, Hobbyists and Companies contribute in the development of the kernel
- Supported by a very large software ecosystem
  - Bootloader, System Programs, Networking Service, Advanced Graphic Services, etc., are easily available
- Stable and Reliable
- Free

3.1.6 **4. Components of Embedded Linux System**

When you start working of an Embedded Linux project, you would find lots of components to be considered to build an distribution. A simple block diagram shown below can give you some basic high level idea on possible block



So from the above diagram you can see some major components like

- Toolchain
  - A major block that is needed build and test an embedded Linux
  - Will involve different tools in different stage
  - We have multiple option to get these tools installed
  - The major point is, we would not prefer this a to be
- Bootloader
  - Responsible for load and boot the kernel with the help of Device Tree Blob
  - Multiple stages possible
- Kernel
  - The core of the system
  - Responsible load all the existing drivers
  - Allow the user to login and interact with the system
- Root File System
  - An organized data storage containing all the needed applications, libraries and configurations

4.0 Toolchain

The first and major component without which its not possible to build and embedded system (in fact we can say a system in general).

4.1.1 **1. What is a Toolchain?**

- Set of applications which help us to generate and investigate binaries as per the need
- One application's output may be used as in input to next, so that the final executable is generated, hence the name Toolchain

4.1.2 **2. Why Toolchain?**

As said this a first and major component we will start with in our product development phase. But in general we did not discuss or ponder on this topic in general while writing applications in our PC right? why now? Well,

- Generally, the tools installed in the host are native, which mean they generate the output for the same system!
- But in case of embedded systems, the binaries are specific to the Target System, hence we have tools which can generate them as needed
- This is where the cross toolchain come into existence

4.1.3 **3. Components of Toolchain**

Well, while working on system development we may have to deal with lots of tools based on the situation, but there few major components which we may have tp specifically consider as put below

- Compiler
- Binary Utilities (binutils)
- Libraries
- Kernel Headers
- Debugger

4.1.4 **4. Obtaining Toolchain**

It possible for us to get the toolchain in different possible means, like to name some

- Home Built - Creating toolchain from scratch ourselves. Required good time, patients and knowledge
- Build System - Adapting some commonly available build system to build the toolchain just by configuration.
- Pre-Built - Download some readily available package that can be installed

In our case let's consider the last one, since its very easy and suffice our need.

We will be going with pre-built toolchain by Linaro, In case your are working on Ubuntu, you may use the below command to install it

```
user@Host:~] sudo apt install gcc-arm-linux-gnueabi # Incase of soft fpu
```

Incase your target supports hardware floating point unit, you may install

```
user@Host:~] sudo apt install gcc-arm-linux-gnueabihf # Incase of hard fpu
```

5.0 Bootloaders

5.1.1 **1. Introduction**

What's a Bootloader?

- A program that is responsible to find, load and execute the main application
- A system may have multiple such programs until the final applications is executed
- The number of stages depends on the complexity of the Target System

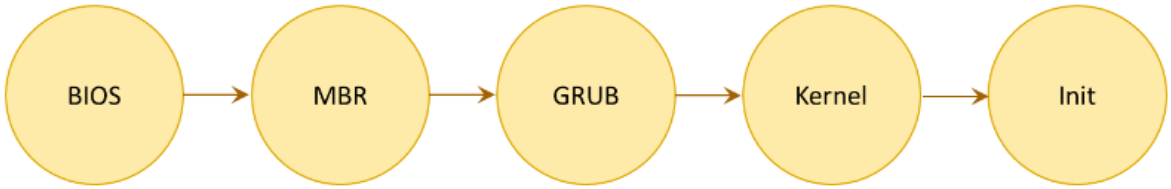
Why Bootloader?

- Helps to do a POST before next stage programs come into existence
- The SoC vendors may provide the target board designers to choose the secondary storage, the bootloader is responsible to identify those storages and execute the application
- There could arise a need of upgrading or updating the application, the bootloader will play a significant role in such instances
- We may need a console to repair the corrupted system, bootloaders help in that aspect

"A layer which helps to get, load and run the kernel"

5.1.2 **2. Boot Sequence**

5.1.2.1.1 **PC Architecture:**



In personal computers we have traditional boot sequence as put in above diagram, wherein you will see multiple stages before an user program can start, it all starts with

BIOS: Basic Input Output System. A piece of code stored in memory (Flash in modern systems) part of motherboard. Responsible for POST (Power ON Self Test). Responsible to find and load the MBR in sequence.

MBR: Master Boot Record. A region in 0^{th} sector of the drive, holding 2 information, one, the partition table and the other is the piece of code to load and run the next stage bootloader in this case its GRUB.

GRUB: GRand Unified Bootloader. Known as GRUB2 today, Multiboot boot loader responsible to load and run the OS core known as kernel

Kernel: The core of the operating system allowing the user to access all the peripherals connected to the system without ant hassle

Init: The first user process that is spawned by the kernel

5.1.2.1.2 **Embedded Architectures:**

If you observe the above diagram, it looks similar to the desktop architecture booting sequence,

ROM Code: A piece of code found with the SoC, that gets executed as soon as the board is powered on. This will have its own scanning process based on selected chip architecture. You will have to check the target board and SoC architecture to understand in detail

SB1L: Stage 1 Boot Loader. Responsible to load and execute the second stage boot loader. The necessity of the stage depends on the design.

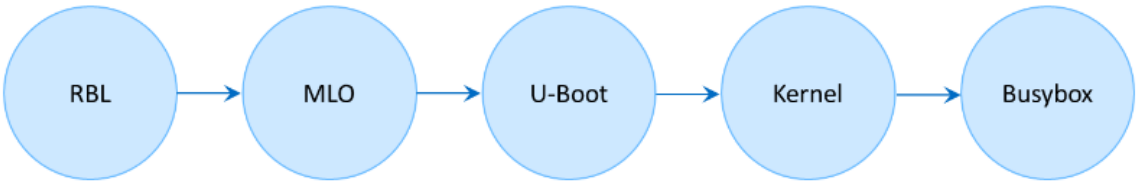
SB2L: Stage 2 Boot Loader. Responsible to load and execute the kernel. The necessity of the stage depends on the design. In absence of SB1L this would be the one that loads the kernel.

Kernel: The core of the operating system allowing the user to access all the peripherals connected to the system without ant hassle

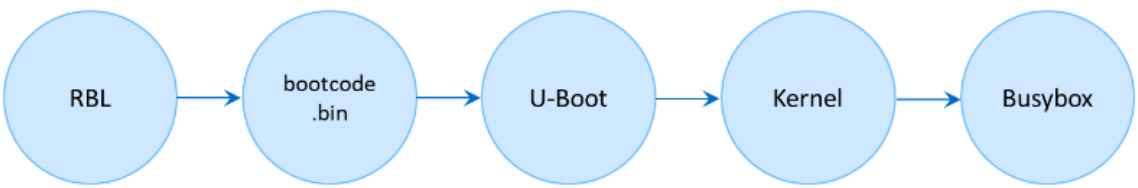
Init: The first user process that is spawned by the kernel

5.1.2.1.3 **Use Cases:**

Beaglebone Black:



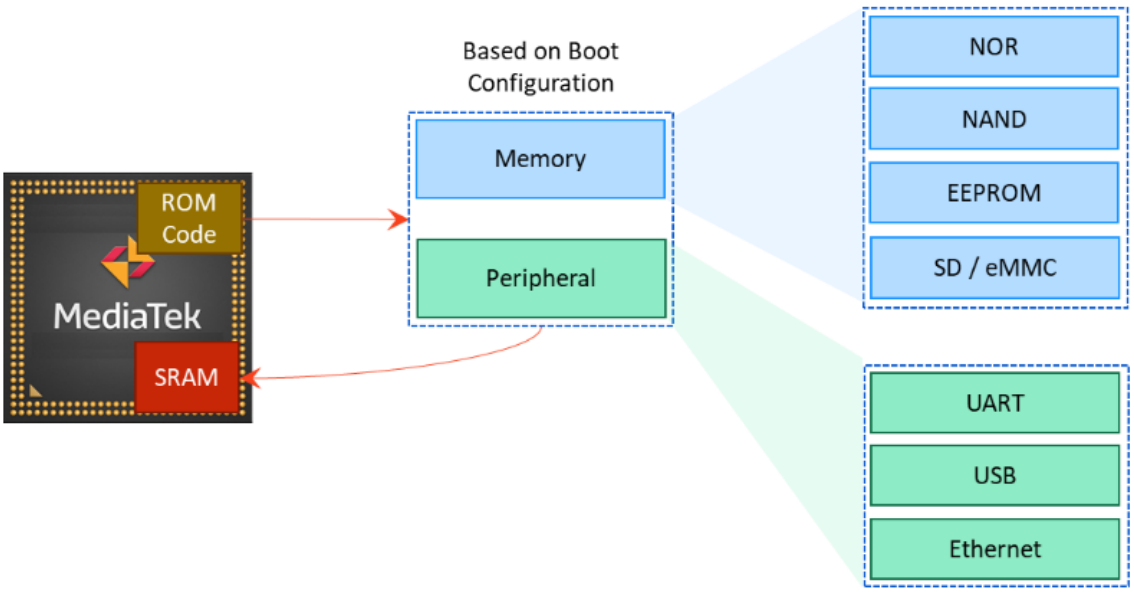
Raspberry Pi:



If you observe the above 2 diagrams the major difference is the second stage bootloader. As said, this depends on the target board and mainly depends on the behavior of the ROM code's dependence, and rest all would be similar only being differed in the features supported by the target board.

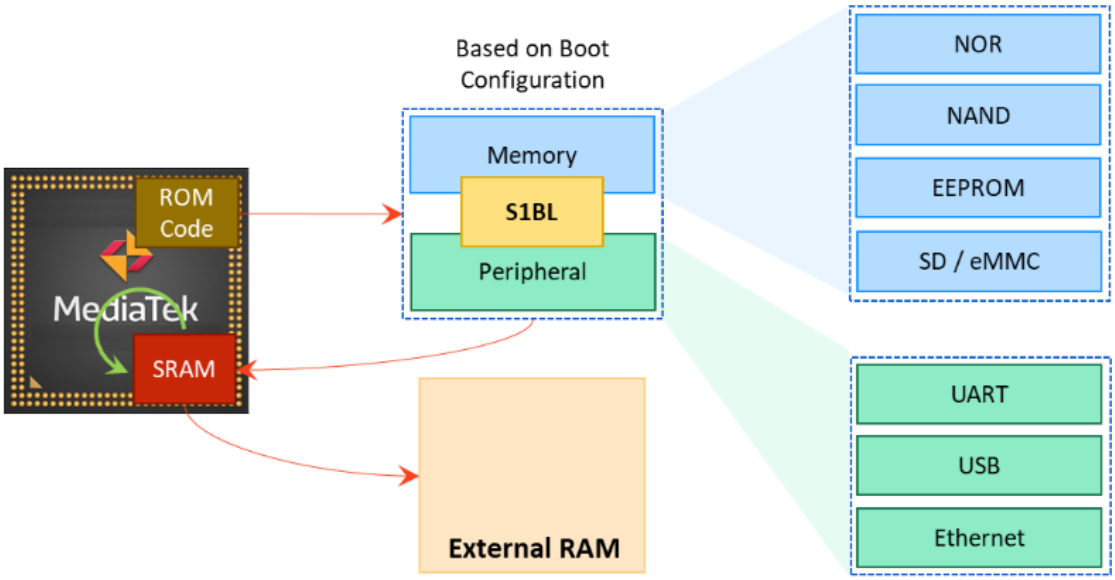
5.1.3 **3. Role of Bootloader Stages**

5.1.3.1.1 **ROM Code:**



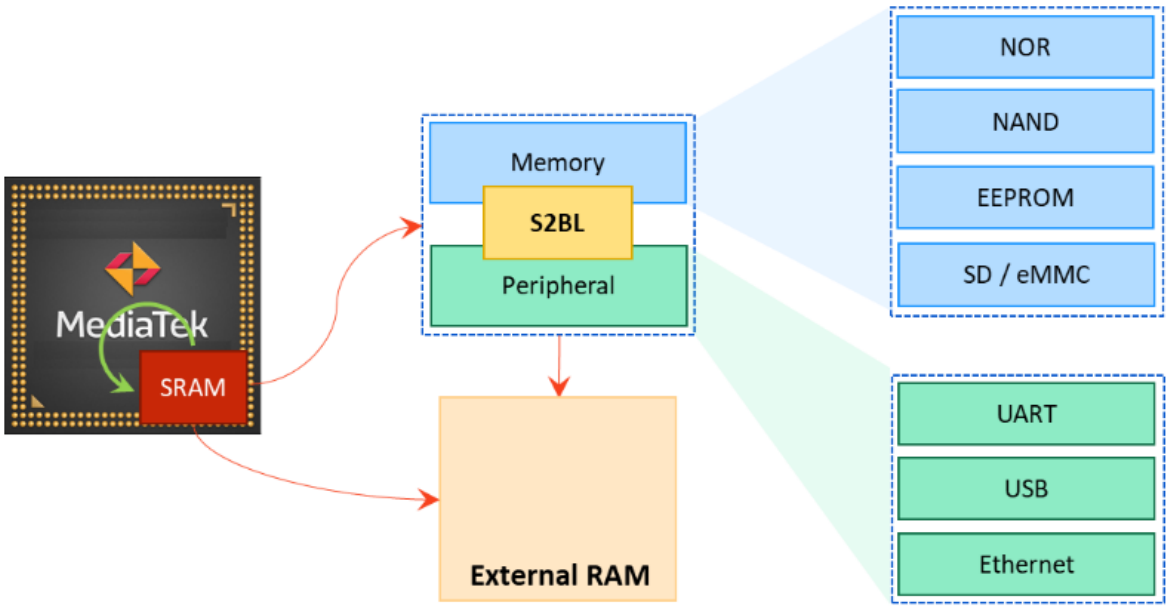
- Shipped as part of SoC as a factory code, that generally cannot be modified
- Checks for the boot configuration to identify the bootable medium
- The boot media could be memory or peripheral based
- Once identified the ROM Code is responsible to fetch and download in the internal SRAM and execute it.
- The drawback is the memory footprint of the SRAM

5.1.3.1.2 **Stage 1 Bootloader (S1BL):**



- This is Stage 1 Bootloader
- The ROM Code is responsible to get and execute this from boot media with the help of internal SRAM
- This code is primarily responsible for initializing the external memory, and use it as buffer to download the S2BL
- Once initialized, the control goes to download S2BL and execute the downloaded binary

5.1.3.1.3 **Stage 2 Bootloader (S2BL):**

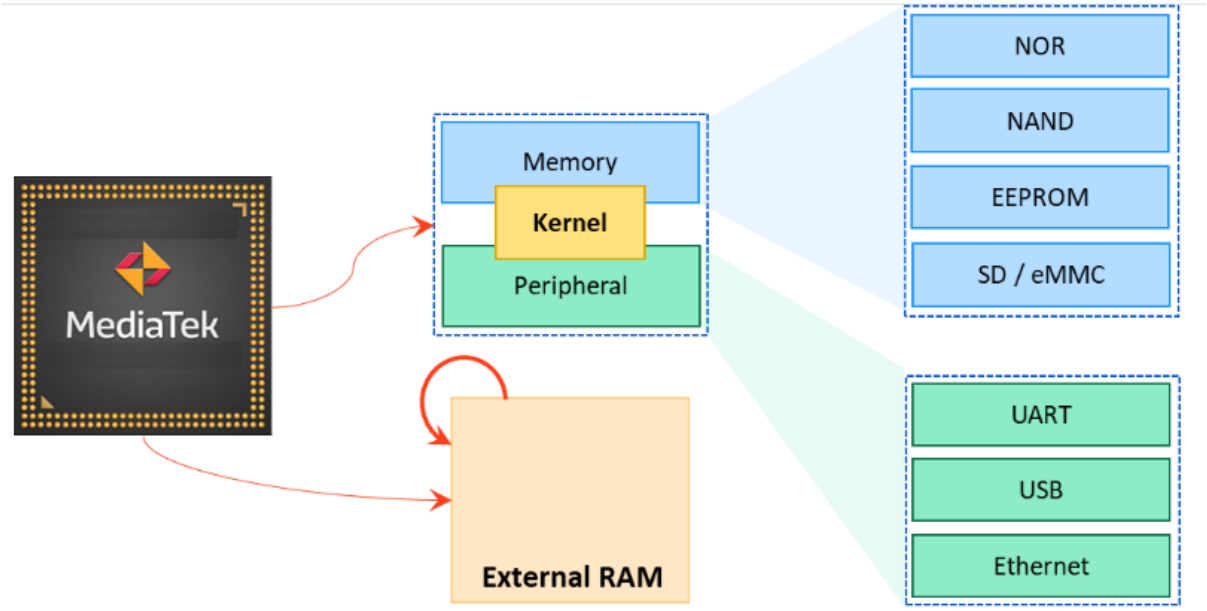


- The S1BL is responsible to get and execute this from boot media with the help of external RAM
- This code is responsible for many functions

- Execute from flash if supported and configured
- Does a POST and relocate to RAM (if XIP is supported)
- Provide console for user interaction
- Provide options to choose and download the kernel, dtb and RFS image
- Provide options to set the kernel boot arguments
- Jump to the kernel start address

5.1.3.2 Kernel:

- Number of stages depends on the target architecture
- It is possible that ROM code can point to the Second Stage Bootloader directly by making the number of stage 4 instead of 5



- Number of stages depends on the target architecture
- It is possible that ROM code can point to the Second Stage Bootloader directly by making the number of stage 4 instead of 5

5.1.4 **4. U-Boot**

- An opensource bootloader by Denx
- Named as Universal Bootloader due to its support for vast number of target boards
- License: GPLv2
- Supports Linux like configuration options
- Light weighted hence commonly used as embedded bootloader

- Support shell interface
- You can get from SourceCode | U-Boot | DENX (Note: the link may change)

5.1.5 4. U-Boot

5.1.5.1 4.1. Configuration

- Download the source code (make sure you have selected a stable version as needed) and store it in you project source directory or you may also use git clone
- Assuming you have un-compressed or cloned the source, you will have to get into the top level directory (Refer the project workspace creation section)

```
user@Host:u-boot-v2022.04] ls # Note: The contents depends on the version
Kbuild       Makefile  board   config.mk  drivers   fs       post
Kconfig      README    boot    configs    dts       include  scripts
Licenses     api       cmd     disk       env       lib      test
MAINTAINERS  arch      common  doc        examples  net      tools
user@Host:u-boot-v2022.04]
```

- The version consider here is u-boot-v2022.04
- The simplest method to configure the u-boot would be using a default configuration file found in config directory on top level

```
user@Host:u-boot-v2022.04] ls configs # Note: The contents depends on the version
am335x_evm_defconfig           Bananapi_defconfig      dragonboard820c_defconfig
Mele_M9_defconfig              nokia_rx51_defconfig    sandbox_defconfig
vexpress_ca9x4_defconfig
user@Host:u-boot-v2022.04]
```

- You will see lots of configuration files in this directory
- Select the one which matches you target board
- Assume you are working on ARM Versatile Express Board, you will have to configure your board as follows

```
user@Host:u-boot-v2022.04] make ARCH=arm vexpress_ca9x4_defconfig
HOSTCC  scripts/kconfig/conf.o
HOSTLD  scripts/kconfig/conf
#
# configuration written to .config
#
user@Host:u-boot-v2022.04]
```

- As you can see above, the configuration would be stored in a .config file, you may proceed with the build

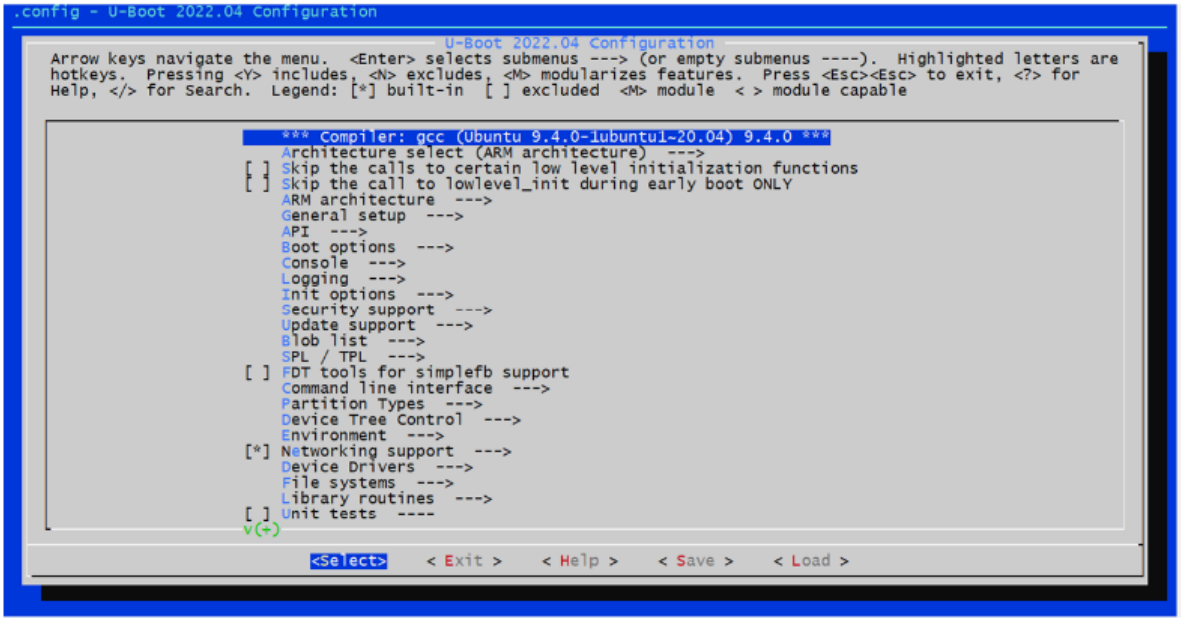
- But if you need any further configuration, you may invoke the menuconfig target

```
user@Host:u-boot-v2022.04] make ARCH=arm menuconfig
```

- It is possible that the menuconfig target may fail due to system package dependency
- So, you may try installing the below packages

```
user@Host:u-boot-v2022.04] sudo apt install libncurses-dev bison flex
Note: This may take some time to install and make sure all the packages are installed
without errors
user@Host:u-boot-v2022.04]
```

- You may re-invoke the menuconfig target again, and you will get a ncurses based configuration screen as shown



- Select the options / features as needed and you may save and exit
- The existing .config file would be updated
- **Note**: if you run defconfig target again, the manual configuration done would be overwritten and you will have to redo it again

5.1.6 **4. U-Boot**

5.1.6.1 4.2. Build

- Once u-boot is configured, it's time to build it
- You may invoke build as shown below

```
user@Host:u-boot-v2022.04] make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
```

- The build would take some time based on your system configuration
- It is possible that the menuconfig target may fail due to system package dependency
- So, you may try installing the below package

```
user@Host:u-boot-v2022.04] sudo apt install libopenssl-dev
Note: This may take some time to install and make sure all the packages are installed
without errors
user@Host:u-boot-v2022.04]
```

- Once the build is successful you will see multiple outputs files, some are as shown below
    - u-boot - An ELF file
    - u-boot.bin - Binary file
    - u-boot.srec - Motorola S-Record; binary data in text format
- You may use one of them to deploy and test on your target board
- Let's copy the executable in the Images directory (not mandatory tough) for ease of access

```
user@Host:u-boot-v2022.04] cp u-boot ../../Images
user@Host:u-boot-v2022.04] cd ../../Images
user@Host:Images]
```

5.1.7 **4. U-Boot**

5.1.7.1 4.3. Deploy

- Once the build is successful, its time to test the binary image that got generated
- It is assumed, you have the target board for which the binary is built for and is already interfaced with the host machine

- The transfer will depend on the Target SoC / Target Board you have
- To start with let's test the binary with the help of Qemu

5.1.7.2 **Qemu:**

- Open-source emulator
- Can emulate multiple processors and target machines
  - BareMetal Programs
  - Operating Systems
  - User level process built for other target processors
- For ARM based systems, it can be installed as

```
user@Host:Images] sudo apt install qemu-system-arm
```

- After installation you may verify it as

```
user@Host:Images] qemu-system-arm -version # The version may vary on your system
QEMU emulator version 4.2.1 (Debian 1:4.2-3ubuntu6.21)
Copyright (c) 2003-2019 Fabrice Bellard and the QEMU Project developers
user@Host:Images]
```

- Qemu supports varieties of machines

```
user@Host:Images] qemu-system-arm -M help # or -M ?
```

- The above command would list all supported machines
- Pick the machine for which you have configured u-boot for
- In our case it was for ARM Versatile Boards - vexpress-a9
- To deploy the binary, follow the steps mentioned

```
user@Host:Images] export QEMU_AUDIO_DRV=none
```

- The above command will disable annoying sound driver message, since we don't have the plan to use sound for time being
- The above command is needed only once for the current terminal instance.

```
user@Host:Images] qemu-system-arm -M vexpress-a9 -nographic -kernel u-boot
U-Boot 2022.04 (Jun 09 2022 - 12:50:27 +0530)
DRAM:  128 MiB
WARNING: Caches not enabled
 Core:  12 devices, 7 uclasses, devicetree: embed
Flash: 64 MiB
MMC:   mmci@5000: 0
Loading Environment from Flash... *** Warning - bad CRC, using default environment
In:    serial
Out:   serial
Err:   serial
Net:   eth0: ethernet@3,02000000
Hit any key to stop autoboot:  0
=>
```

- Observe a clean prompt!, its because I had pressed ENTER key to stop autoboot, else it would have auto booted with some errors and then you will get the prompt
- You may use CTRL+a (observe both key should be pressed simultaneously, release) and x to quit the current instance
- The first command on u-boot prompt you may type is

```
=> help # This will list all the command which is part of the configuration
```

Let's discuss see some commonly used command used in embedded Linux development

5.1.8 **4. U-Boot**

5.1.8.1 4.4. Basic Commands Set

- Boot Commands
  - bootm - Boot the loaded image from memory, usually used with uImage
  - bootp - Boot image via network using BOOTP / TFTP protocol
  - bootz - Boot the loaded zImage from the memory
  - tftpboot - Boot image via network using TFTP protocol
  - go - Execute a loaded application from memory
- Environment Commands
  - printenv - Print environmental variables
  - setenv - Set an environmental variable
  - echo - To print on variable contents, work like Linux shell echo
- List Commands
  - ls - List file and directories at the given path, / is default
  - ext2ls, ext4ls - List files from EXT formatted partition
  - fatls - List file from FAT partitions
- Load Commands
  - fatload - Load binaries from FAT partitions into memory at specified location

    - ext2load, ext4load - Load binaries from EXT partitions into memory at specified location
    - tftpboot - Load binary via network using TFTP protocol into memory at specified location
- General Commands
    - reset - Resets the target
    - mmc - Functions related to MMC subsystem
    - version - To print the running U-Boot version and the list continues
- But note, the available command list will be based on the configuration we have done
- We may add or remove command sets using menuconfig framework

5.1.9 **4. U-Boot**

5.1.9.1 4.5. Customization and Testing

- Let's do a simple customization that can be easily seen by us
- Let configure u-boot to change the command prompt and rebuild again
- Quit the running instance of Qemu
- Move to the U-Boot Source directory, and then

```
user@Host:Images] cd ../U-Boot/u-boot-v2022.04
user@Host:u-boot-v2022.04] make ARCH=arm menuconfig
```

- You get a ncurses based configuration screen and on the top level select
    - "Command line interface" under that
    - Move to "Shell prompt" and press enter, you should get a dialog box with existing prompt
    - Remove it and enter "u-boot] " and enter
    - Keep on pressing ESC key till you get the save dialog box
    - Save and exit to build

```
user@Host:u-boot-v2022.04] make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
```

- You may test and the latest binary

LINUX EMBEDDED SYSTEM

```
user@Host:u-boot-v2022.04] qemu-system-arm -M vexpress-a9 -nographic -kernel u-boot
U-Boot 2022.04 (Jun 09 2022 - 17:27:40 +0530)
DRAM:  128 MiB
WARNING: Caches not enabled
Core:  12 devices, 7 uclasses, devicetree: embed
Flash: 64 MiB
MMC:   mmci@5000: 0
Loading Environment from Flash... *** Warning - bad CRC, using default environment
In:    serial
Out:   serial
Err:   serial
Net:   eth0: ethernet@3,02000000
Hit any key to stop autoboot:  0
u-boot]
```

- Observe the command prompt
- Similarly, we can we can customize to add and remove features as needed
- Once you feel you have working version, you may move it to the Images directory for loading kernel images with help of it

```
ruchira@PETSL01166:~/EmbeddedLinux/U-Boot/u-boot-master$ ls
Kbuild  Licenses    Makefile  api   board  cmd     config.mk  disk  drivers  env      fs       lib  post     test
Kconfig  MAINTAINERS  README    arch  boot   common  configs    doc   dts      examples  include  net  scripts  tools
ruchira@PETSL01166:~/EmbeddedLinux/U-Boot/u-boot-master$ ls configs/ | grep vexp
vexpress_aemv8a_juno_defconfig
vexpress_aemv8a_semi_defconfig
vexpress_aemv8r_defconfig
vexpress_ca9x4_defconfig
ruchira@PETSL01166:~/EmbeddedLinux/U-Boot/u-boot-master$ make ARCH=arm vexpress_ca9x4_defconfig
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/kconfig/conf.o
  YACC    scripts/kconfig/zconf.tab.c
  LEX     scripts/kconfig/zconf.lex.c
  HOSTCC  scripts/kconfig/zconf.tab.o
  HOSTLD  scripts/kconfig/conf
#
# configuration written to .config
#
ruchira@PETSL01166:~/EmbeddedLinux/U-Boot/u-boot-master$ ls
Kbuild  Licenses    Makefile  api   board  cmd     config.mk  disk  drivers  env      fs       lib  post     test
Kconfig  MAINTAINERS  README    arch  boot   common  configs    doc   dts      examples  include  net  scripts  tools
ruchira@PETSL01166:~/EmbeddedLinux/U-Boot/u-boot-master$ ls arch
Kconfig  Kconfig.nxp  arc  arm  m68k  microblaze  mips  nios2  powerpc  riscv  sandbox  sh  u-boot-elf.lds  x86  xtensa
ruchira@PETSL01166:~/EmbeddedLinux/U-Boot/u-boot-master$ make ARCH=arm menuconfig
  UPD     scripts/kconfig/.mconf-cfg
  HOSTCC  scripts/kconfig/mconf.o
  HOSTCC  scripts/kconfig/lxdialog/checklist.o
  HOSTCC  scripts/kconfig/lxdialog/inputbox.o
  HOSTCC  scripts/kconfig/lxdialog/menubox.o
  HOSTCC  scripts/kconfig/lxdialog/textbox.o
  HOSTCC  scripts/kconfig/lxdialog/util.o
  HOSTCC  scripts/kconfig/lxdialog/yesno.o
  HOSTLD  scripts/kconfig/mconf
scripts/kconfig/mconf  Kconfig


*** End of the configuration.
*** Execute 'make' to start the build or try 'make help'.
```

LINUX EMBEDDED SYSTEM

5.1.10 **Introduction**

- **What's a Kernel?**
  - Core of an Operating System that is responsible manage the hardware resources across the applications efficiently and effectively

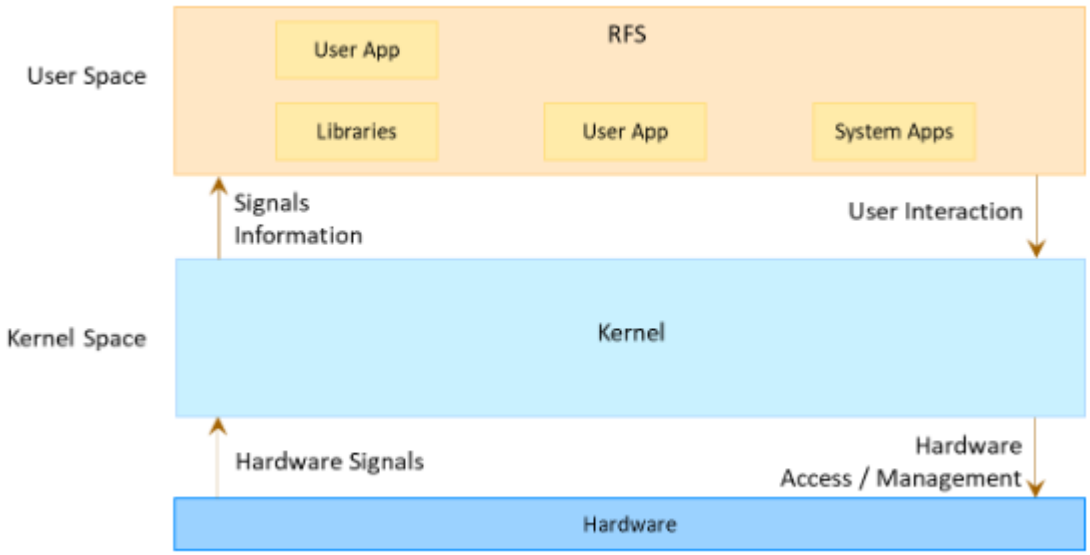
- **Why Kernel?**
  - As the complexity of the system requirement increase, the number of peripheral interfaces too increases
  - The bare metal programing approach would not be the efficient pick, we may see lag in the system performance
  - So, the scheduling approach is preferred, and the kernel is one which responsible for that

- **Why Linux Kernel?**
  - An opensource software created by Linus Torvalds, The full source code is available for learning and adaptation
  - Strong development community across the world, you will see individuals and corporates in the pool
  - Runs in most of the processor architectures and good portability support
  - Provides a mature and stable build system being an opensource initiative
  - The kernel can be configured in such a way that, it can be made to run on tiny devices as well as super computers, that's the scalability advantage we get with Linux kernel
  - It's stable, reliable and secure
  - Its modular, it allows us to load or unload the modules or drivers at runtime
  - It has good compliance to standards and is interoperable
  - Royalty free

- **Important Terms**
  - Unix - Multitasking, multi-user OS developed in 1969 by AT&T employees at Bell Labs
  - Linux - Unix like (not implementation) kernel by Linux Torvalds, officially known as GNU / Linux
  - GNU - GNU is not Unix. Collection of Free Software that can form an OS or can be part of an OS
  - GPL - General Public License, that guarantee the freedom to Use, Study, Modify and Share. In fact, the first copyleft license
  - FSF - Free Software Foundation, to promote GNU by Richard Stallman
  - FOSS - Free and Opensource Software
  - Distribution - Collections of software including Bootloaders, Kernel and RFS with applications forming an Operating System

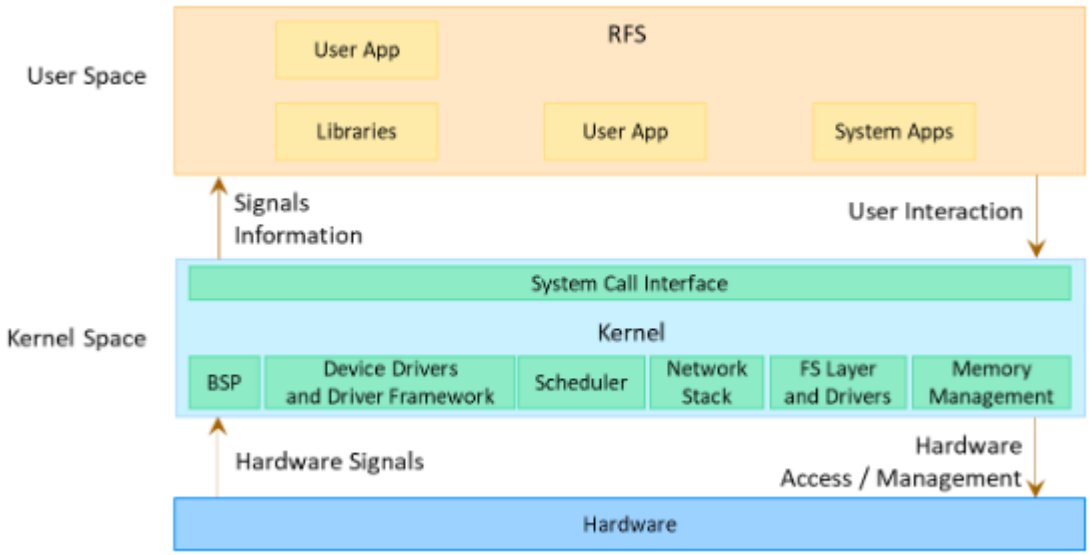
- **Interface**

LINUX EMBEDDED SYSTEM



From the above diagram you can observe that the kernel acts an interface between the user and hardware. Lots of information would pass between the user space and kernel space to interact with the hardware that could be directly or indirectly.

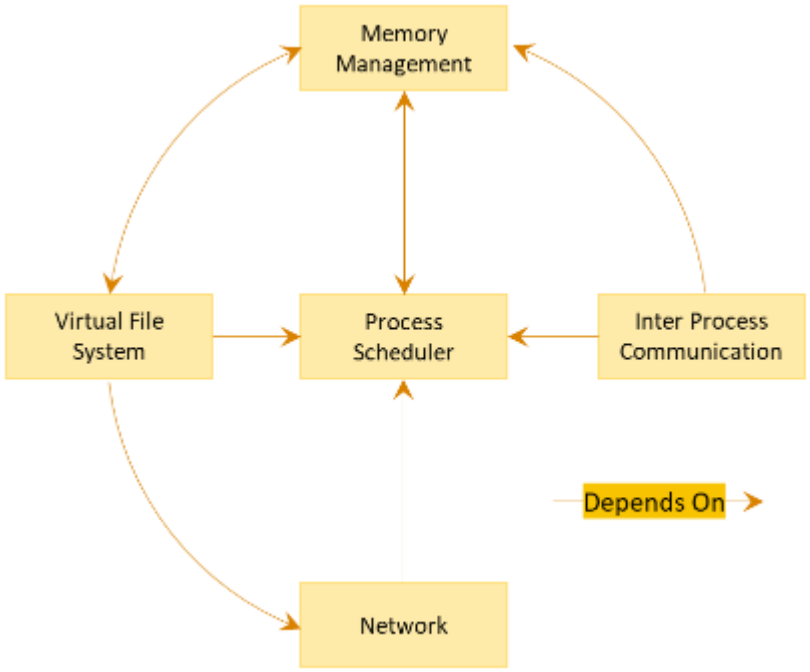
So, it is the kernel which is responsible to facilitate all the system activities by providing different features, a high-level picture is given below



- **Major role of Kernel**
  - Hardware Management
    - CPU
    - Memory and Storage
    - Network

        - I/O …
    - Interaction
        - Should provide arch and hardware independent APIs so that user applications can access the hardware
    - Concurrency
        - Allow hardware access to multiple applications

- **Kernel Subsystem**



- Process Scheduler
    - Responsible to provide fair access of CPU to the process while interacting with the hardware
- Memory Manager
    - Responsible to allocate system memory securely and efficiently by multiple process. Incase of larger memory needs, it should support Virtual Memory
- Virtual File System
    - This layer is responsible for abstracting the application from the internal hardware being used and provide a common interface for all the devices
- Network
    - Responsible to provide standards to access networking hardware
- IPC
    - Responsible to allow user process communication securely and efficiently

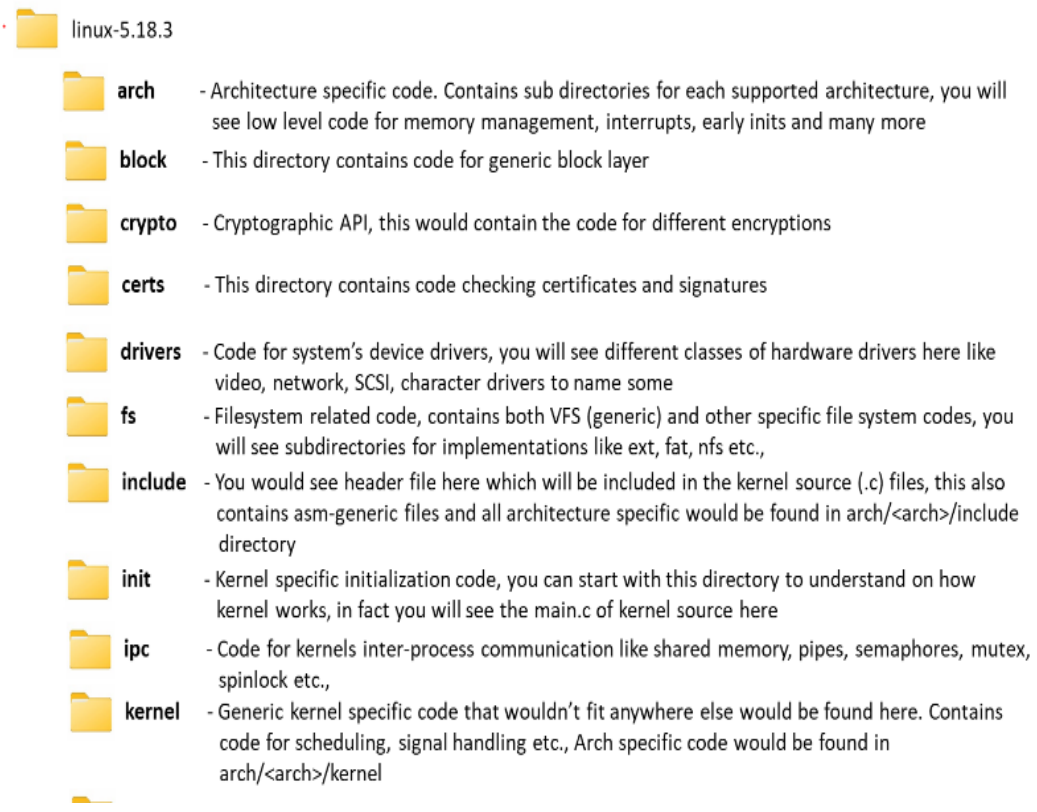
- **Where to get the source?**

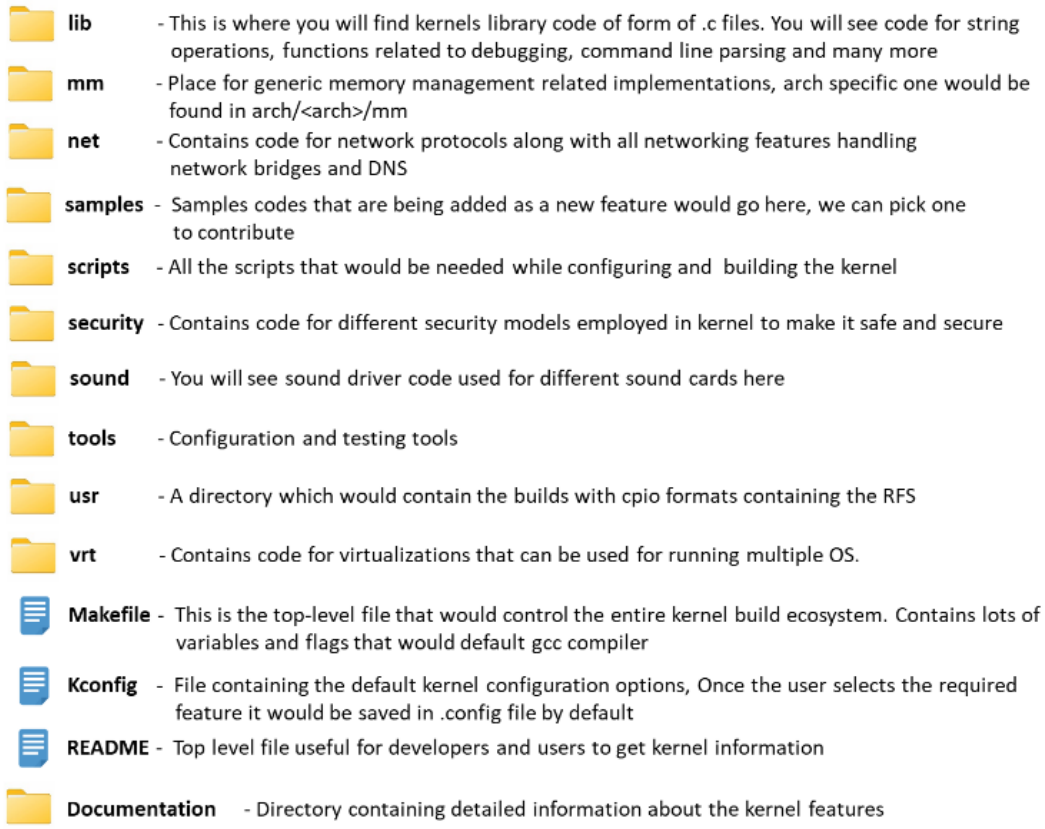
- www.kernel.org

5.1.11 **2. Kernel Source Tree**

- Download the source code (make sure you have selected a stable version as needed) and store it in you project source directory or you may also use git clone
- Uncompress the downloaded source (Refer the project work creation section)

```
user@Host:linux-5.18.3] ls # Note: The contents depends on the version
COPYING         Kconfig      README  crypto   init    mm        security  virt
CREDITS         LICENSES     arch    drivers  ipc     net       sound
Documentation   MAINTAINERS  block   fs       kernel  samples   tools
Kbuild          Makefile     certs   include  lib     scripts   usr
user@Host:linux-5.18.3]
```





5.1.12 **3. Build Targets**

- Once the kernel source is downloaded, its time to configure and build it
- The kernel build system provides us lots of functions / targets to do that
- It's all based on multiple Makefiles, and the top level Makefile would invoke the inner levels as needed
- To know the available targets, you may type the below command

```
user@Host:linux-5.18.3] make ARCH=arm help | less
```

- In the above command if you don't specify the ARCH, it will pick the host architecture implicitly
- The list would depend on the architecture selected
- Let's see some important targets provided by the build system

Cleaning targets:

- clean - Remove most generated files but keep the config and enough build support to build external modules
- mrproper - Remove all generated files + config + various backup files
- distclean - mrproper + remove editor backup and patch files

- The above targets are used to clean the kernel source in case you want to build the kernel source fresh
- Depending on the depth of cleaning you may pick any one of them

Configuration targets:
The Linux kernel build system provides us lots of configuration options that can be used as needed
Some commonly used configurations targets are

- config
  - Based on line-oriented program
  - Though this not that user friendly, could be used in situations like limited host installation
  - Forces you to follow a sequence of questions while configuring the kernel
- menuconfig
  - The most common make target used to configure the Linux kernel
  - Simple to use based on ncurses library
  - Navigation options are bases on keyboard arrow keys and hot keys
- xconfig
  - GUI based approach, needs libqt-dev package installation
  - Easy to use with the help of mouse and key board

Build targets:

- The build framework provides us some architecture specific targets
- The default is zImage, which is a compressed kernel image
- In case you want U-Boot wrapped zImage you may explicitly provide uImage while building the kernel

5.1.13 **4. Configuration**

- The downloaded kernel source contains code for variety of architectures and hardwares
- We will not be able to build unless and until we specify the architecture we intend to build for, and that must be done by kernel configuration
- As seen in the previous slide there are few configuration targets available that could be used, **menuconfig** being the common pick
- But the issue is, we will have to configure all from scratch which would be time consuming and knowledge curve needed would be high
- To make life easy, the board vendors provide default configuration files
- These configurations file would make sure the right architecture is set along with the peripherals interfaced in the target system

- These configuration can be found in **"Architecture specific targets:"** section of **make ARCH=help** output
- The other way to get this info is to see the configs directory as shown below

```
user@Host:linux-5.18.3] ls arch/arm/configs # Note: you will see lots of file here.
at91_dt_defconfig              bcm2835_defconfig      exynos_defconfig
multi_v7_defconfig             omap1_defconfig        qcom_defconfig
vexpress_defconfig
user@Host:linux-5.18.3]
```

- Proceeding with these configuration file in general save lots of time
- Once you have the kernel running, you can reconfigure it as needed later using menuconfig target later
- Assume, you are working on ARM Versatile Express Board, you will have to configure your board as follows

```
user@Host:linux-5.18.3] make ARCH=arm vexpress_defconfig # The output print may vary!
HOSTCC  scripts/kconfig/conf.o
HOSTLD  scripts/kconfig/conf
#
# configuration written to .config
#
user@Host:linux-5.18.3]
```

- As you can see above, the configuration would be stored in a .config file, you may proceed with the build
- But if you need any further configuration, you may invoke the menuconfig target

```
user@Host:linux-5.18.3] make ARCH=arm menuconfig
```

- The configuration would be saved in .config file found in top level of kernel source
- If you feel the above configuration is fine and need to preserve it, you may follow as mentioned below

```
user@Host:linux-5.18.3] make ARCH=arm savedefconfig
user@Host:linux-5.18.3] ls
COPYING        Kconfig      README  crypto     include  lib      scripts   usr
CREDITS        LICENSES     arch    defconfig  init     mm       security  virt
Documentation  MAINTAINERS  block   drivers    ipc      net      sound
Kbuild         Makefile     certs   fs         kernel   samples  tools
user@Host:linux-5.18.3] mv defconfig arch/arm/configs/custom_defconfig
```

- You may give a meaningful name based on you project and use it later

```
user@Host:linux-5.18.3] mv defconfig arch/arm/configs/custom_defconfig
```

5.1.14 **5. Build**

- Once the Linux kernel is configured, it's time to build it
- You may invoke build as shown below

```
user@Host: linux-5.18.3] make -j$(nproc) ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
```

- The build could take some good time based on your system configuration
- It is assumed that the dependency packages are installed as mentioned Host Setup
- Observe -j option that could help you speedup the build
- Since we have not provided any explicit targets, it will build all the default targets which will be marked with * in make ARCH=help output
- Once the build is successful you will see multiple outputs files, some are as shown below
    - **vmlinux** – The bare kernel in ELF format. This is bulky file containing the symbol information that is generally used of debugging purpose. This file acts as source to generate other target images. This will be found in the top level of kernel source directory
    - **Image** - Uncompressed bootable kernel image found in arch/<arch>/boot directory
    - **zImage** - Compressed bootable kernel image for ARM based targets. Even this would be found in found in arch/<arch>/boot directory
    - **vexpress-dtb** - Device tree blob for the selected target board. This will be available in arch/<arch>/boot/dts directory
    - **.ko files** - The modules if any selected while configuration
- It's time to test the built image on the target

5.1.15 **6. Deploy - Qemu (vexpress-a9)**

- Once the build is successful, its time to test the binary image that got generated
- It is assumed, that u-boot is already running on your target board

LINUX EMBEDDED SYSTEM

- The next step is to think on how to get the kernel image in the memory!
- Well, there are multiple possibilities and that depends on you target machine peripheral and secondary storage support
- But to keep it simple let's test the binary with the help of **Qemu**
- Now, if its Qemu we really don't have the bootloader dependency, that cool isn't it
- Yeah, it save lots of time, once the we have the final image, we can boot it with the bootloader
- Let's disable the sound driver messages

```
user@Host:linux-5.18.3] export QEMU_AUDIO_DRV=none
```

- Let's copy the needed files to the Images directory (Note: Assuming you have followed the directory structure as mentioned in Host Setup chapter)

```
user@Host:linux-5.18.3] cp arch/arm/boot/zImage ../../Images
user@Host:linux-5.18.3] cp arch/arm/boot/dts/vexpress-v2p-ca9.dtb ../../Images
user@Host:linux-5.18.3] cd ../../Images
user@Host:Images]
```

- Let's run the qemu command to boot the kernel image, as

```
user@Host:Images] qemu-system-arm -M vexpress-a9 -nographic -kernel zImage -dtb vexpress-
v2p-ca9.dtb
Booting Linux on physical CPU 0x0
Linux version 5.18.3 (adil@MYTSL01490) (arm-linux-gnueabi-gcc (Ubuntu 9.4.0-
1ubuntu1~20.04.1) 9.4.0, GNU ld (GNU Binutils for Ubuntu) 2.34) #2 SMP Wed Jun 15 16:02:46
IST 2022
CPU: ARMv7 Processor  410fc090  revision 0 (ARMv7), cr=10c5    d
CPU: PIPT / VIPT
```

```
panic from mount_block_root+0x1ac/0x248
 mount_block_root from mount_root+0x22c/0x254
 mount_root from prepare_namespace+0x154/0x190
 prepare_namespace from kernel_init+0x18/0x12c
 kernel_init from ret_from_fork+0x14/0x2c
Exception stack(0x88825fb0 to 0x88825ff8)
5fa0:                                     00000000 00000000 00000000 00000000
5fc0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
•  5fe0: 00000000 00000000 00000000 00000000 00000013 00000000
---[ end Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0) ]-
```

- You should see the kernel booting, but once it reaches the point to run init, it fails with a panic

- And the reason behind this is missing Linux Root Filesystem (RootFS) data
- Now, it becomes very important for us to provide the Root Filesystem Data while booting the kernel, else we will not be able to interact with our system
- But what is the RootFS data?
    - how do we get it?
    - how do we know what should be part of it?
    - How would we let our kernel know about it while booting?
- Well, many questions to be answered right?
- Not to worry, you will get them answered as you proceed
- But to start with let's take the simplest approach, and that is, get the ready one for the internet

5.1.16 **6. Deploy - Qemu (vexpress-a9)**

5.1.16.1 6.1. Prebuilt Root Files System (RootFS) Data

- As discussed, lets download a ready made RootFS data from the Yocto Project site, so that, at least we have a running kernel to start with, But note don't expect this work in all the situations
- You may also use wget command to download that too! as shown

```
user@Host:Images] wget -P ../Sources
http://downloads.yoctoproject.org/releases/yocto/yocto-2.5/machines/qemu/qemuarm/core-image-minimal-qemuarm.ext4
```

- The above file is the ext4 format
- Let's also download the raw directory content so that it can be used whenever needed

```
user@Host:Images] wget -P ../Sources
http://downloads.yoctoproject.org/releases/yocto/yocto-2.5/machines/qemu/qemuarm/core-image-minimal-qemuarm.tar.bz2
```

- Now with these ready data let's try to boot the kernel with different methods

5.1.17 **6. Deploy - Qemu (vexpress-a9)**

5.1.17.1 6.2. initramfs

- One of the simplest method to pass the RFS data to kernel
- In this case we need the RFS data in raw format, so let's extract it into the RootFS directory

LINUX EMBEDDED SYSTEM

```
user@Host:Images] cd ../RootFS
user@Host:RootFS] tar xvf ../Sources/core-image-minimal-qemuarm.tar.bz2
```

- We have major consideration here, one of the path the kernel looks for init is the / directory itself, and in case of initramfs this true, if it doesn't find init in / it will panic
- So, let's do that first

```
user@Host:RootFS] cp sbin/init.sysvinit  init # Verify the init path once yourself
```

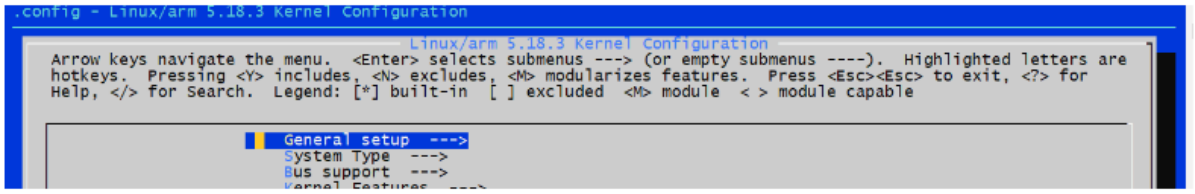
- The next step is to compile the kernel by providing the RootFS directory path, and this must be done by configuring the kernel
- Let's get the path of the RootFS directory first

```
user@Host:RootFS] pwd # The output depends on Project Directory structure, copy the output
/home/user/EmbeddedLinux/RootFS
user@Host:RootFS]
```

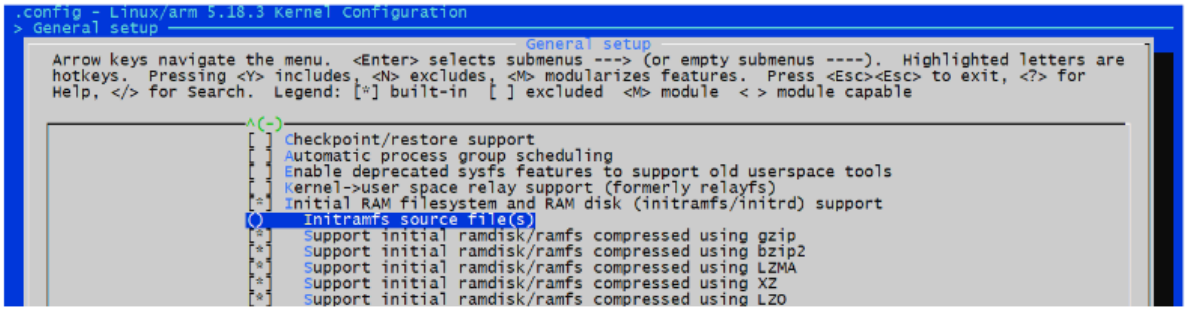
- Move to the kernel Source and start the configuration

```
user@Host:RootFS] cd ../Linux/linux-5.18.3
user@Host:linux-5.18.3] make ARCH=arm menuconfig
```

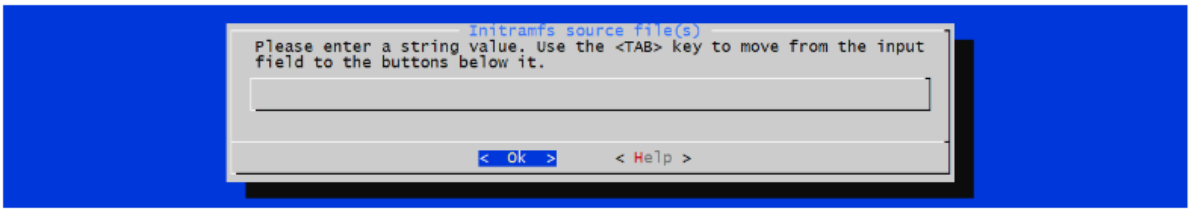
- You should get the configuration screen, Now go to "General Setup"



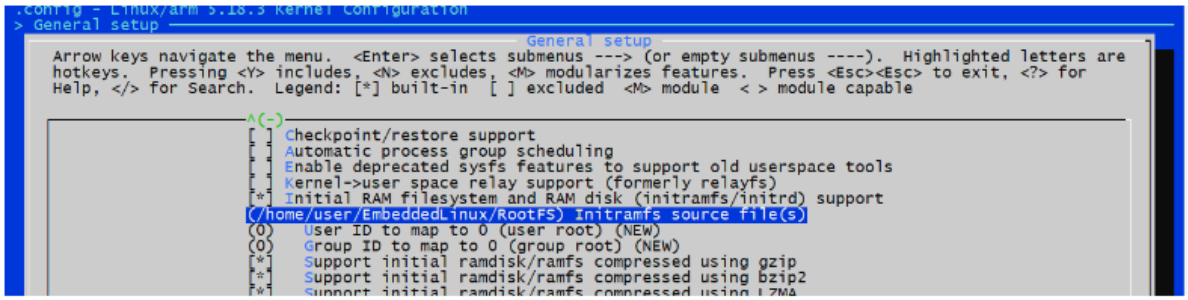
- under that, press Enter Key on the below selection



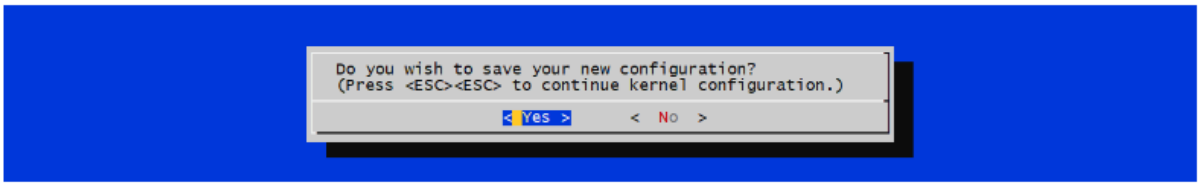
- You will have a window popped up as shown below



Paste the copied path, it should look like



- Keep on hitting ESC key until you get a dialog box on confirming to save,



- Select save and exit
- Since we have changed the configuration, we need to build the kernel again

```
user@Host:linux-5.18.3] make -j$(nproc) ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
user@Host:linux-5.18.3] cp arch/arm/boot/zImage ../../Images/zImage.initramfs # No need to
copy .dtb file
user@Host:linux-5.18.3] cd ../../Images
user@Host:Images]
```

- Let's run the qemu command to boot the kernel image, as

```
user@Host:Images] qemu-system-arm -M vexpress-a9 -nographic -kernel zImage.initramfs -dtb
vexpress-v2p-ca9.dtb
Booting Linux on physical CPU 0x0
Linux version 5.18.3 (user@Host) (arm-linux-gnueabi-gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1)
9.4.0, GNU ld (GNU Binutils for Ubuntu) 2.34) #11 SMP Thu Jun 16 13:01:01 IST 2022
```

```
EXT4-fs (mmcblk0): mounted filesystem with ordered data mode. Quota mode: disabled.
VFS: Mounted root (ext4 filesystem) readonly on device 179:0.
Freeing unused kernel image (initmem) memory: 1024K
Run /sbin/init as init process
random: crng init done
udevd[79]: starting version 3.2.5
udevd[80]: starting eudev-3.2.5
EXT4-fs (mmcblk0): re-mounted. Quota mode: disabled.
ext4 filesystem being remounted at / supports timestamps until 2038 (0x7fffffff)
Poky (Yocto Project Reference Distro) 2.5 qemuarm /dev/ttyAMA0
qemuarm login:
```

- You should see the kernel boots perfectly fine and login prompt is available
- The default user is root, you may type that and ENTER as shown

```
Poky (Yocto Project Reference Distro) 2.5 qemuarm /dev/ttyAMA0

qemuarm login:root
root@qemuarm:~# ls / # Initial RAM Filesystem (initramfs) Data
bin    dev    home   lib    mnt    run    sys    usr
boot   etc    init   media  proc   sbin   tmp    var
root@qemuarm:~# uname -a # System information
Linux qemuarm 5.18.3 #2 SMP Wed Jun 15 16:02:46 IST 2022 armv7l GNU/Linux
root@qemuarm:~#
```

- So, with this you have your Embedded Linux system running on you target board, you may try all those thing you generally do with a Linux system, But note there could be certain limitations, since its embedded Linux and not general-purpose Linux isn't it
- Some points to be remembered here is
    - Any change we do in the filesystem, to reflect that, we need compile the kernel again and redeploy it
    - The data modification on the running system is volatile since it's a RAM filesystem

5.1.18 **6. Deploy - Qemu (vexpress-a9)**

5.1.18.1 6.3. SD / eMMC

- We can use the downloaded ext4 image form Yocto site and can be emulated as a SD Card with -sd option
- But there is catch, we cannot use this image as is, since the geometry (block count) doesn't match as expected by the default configuration done by us!!, hence we need to align the block size
- So, let's do that
- First let's copy the rootfs data in Images directory

```
user@Host:Images] cp ../Sources/core-image-minimal-qemuarm.ext4 .
user@Host:Images]
```

- Let's use the e2fsck command for block alignment

```
user@Host:Images] e2fsck core-image-minimal-qemuarm.ext4 # Even this might suffice if the
downloaded file size is <= 8MB, else run the resize2fs command too!
e2fsck 1.45.5 (07-Jan-2020)
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
rootfs.ext4: 718/1112 files (0.4% non-contiguous), 6807/8192 blocks
user@Host:Images] resize2fs core-image-minimal-qemuarm.ext4 16M # Changing the block size
to 16MB
resize2fs 1.45.5 (07-Jan-2020)
Resizing the filesystem on core-image-minimal-qemuarm.ext4 to 16384 (1k) blocks.
The filesystem on core-image-minimal-qemuarm.ext4 is now 16384 (1k) blocks long.

user@Host:Images]
```

- Assuming intramfs path is **not set** in the **General setup -> Initramfs source file(s) path**
- Now its time to run Qemu. But one question to be answered here is how will the kernel come to know about filesystem?
- Well, this is where the kernel boot arguments come in picture
- The downloaded RootFS data will be treated as an SD card image, and we will tell the kernel to mount that as a multimedia block (mmc) device
- But as soon as we do this via bootargs, the default bootargs with console information will be lost, hence we need to set even that
- So, now we will be using 2 extra options in qemu
    - -sd - for SD card emulation
    - -append - for boot arguments
- Let's run the kernel now with above considerations

LINUX EMBEDDED SYSTEM

```
user@Host:Images] qemu-system-arm -M vexpress-a9 -nographic -kernel zImage -dtb vexpress-
v2p-ca9.dtb -sd core-image-minimal-qemuarm.ext4 -append "console=ttyAMA0
root=/dev/mmcblk0"
Booting Linux on physical CPU 0x0
Linux version 5.18.3 (user@Host) (arm-linux-gnueabi-gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1)
9.4.0, GNU ld (GNU          s for     tu) 2.34) #11 SMP Thu          13:01:01

drm-clcd-pl111 10020000.clcd: DVI muxed to daughterboard 1 (core tile) CLCD
drm-clcd-pl111 10020000.clcd: initializing Versatile Express PL111
/dev/root: Can't open blockdev
/dev/root: Can't open blockdev
udevd[76]: starting version 3.2.5
random: crng init done
udevd[77]: starting eudev-3.2.5

Poky (Yocto Project Reference Distro) 2.5 qemuarm /dev/ttyAMA0

qemuarm login:
```

- You should see the kernel boots perfectly fine and login prompt is available
- The default user is root, you may type that and ENTER as shown

```
Poky (Yocto Project Reference Distro) 2.5 qemuarm /dev/ttyAMA0
qemuarm login:root
root@qemuarm:~# ls / # Passed RootFS Data
bin         dev         home        lost+found  mnt         run         sys         usr
boot        etc         lib         media       proc        sbin        tmp         var
root@qemuarm:~# uname -a # System information
Linux qemuarm 5.18.3 #2 SMP Wed Jun 15 16:02:46 IST 2022 armv7l GNU/Linux
root@qemuarm:~#
```

- So, with this you have your Embedded Linux system running on you target board, you may try all those thing you generally do with a Linux system, But note there could be certain limitations, since this is an embedded Linux and not general-purpose one isn't it
- Some points to be remembered here is
- Since it's a SD storage, the data is persistent and change we do on the running system the
- In case you want to add the data from host to the card you will have to mount the image and copy as shown in the example below.

```
user@Host:Images] sudo mount core-image-minimal-qemuarm.ext4 /media # Make sure no other
media is mounted on this dircetory, if you don't have media dircetory, you may create one
user@Host:Images] echo 'while true; do echo "Hello"; sleep 1; done' | sudo tee
/media/home/root/script.sh # You can do anything you like copying, modifying as needed
user@Host:Images] sudo umount /media
user@Host:Images] qemu-system-arm -M vexpress-a9 -nographic -kernel zImage -dtb vexpress-
v2p-ca9.dtb -sd core-image-minimal-qemuarm.ext4 -append "console=ttyAMA0 root=/dev/mmcblk0"
Booting Linux on physical CPU 0x0
Linux version 5.18.3

root@qemuarm:~# ls # You should the script file here; you add +x permission and run it
script.sh
root@qemuarm:~#
```

5.1.19 6. Deploy - Qemu (vexpress-a9)

5.1.19.1 6.4. initrd

- The Initial RAM Disk (initrd) method needs "RAM Block Device Support" feature enabled in kernel to work
- So, lets first enable that by configuring the Linux kernel

```
user@Host:Images] make ARCH=arm menuconfig -C ../Linux/linux-5.18.3/
```

- menuconfig screen will be opened, now, enable the RAM Block support by navigating to

```
Device Drivers  --->
  [*] Block devices  --->
        <*>  RAM block device support                -> Make sure this * not M
        (16)  Default number of RAM disks
        (16384)Default RAM disk size (kbytes)        -> Provide the Size based on RootFS image size
General setup  --->
    [*] Initial RAM filesystem and RAM disk (initramfs/initrd) support      -> Make sure this *
    ( ) Initramfs source file(s)                                            -> Make sure this is empty
```

- Keep on hitting ESC key until you get a dialog box on confirming to save, select save and exit
- Since we have changed the configuration, we need to build the kernel again

```
user@Host:Images] make -j$(nproc) ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- -C
../Linux/linux-5.18.3/
```

- Once the build is done, lets copy the zImage into the images directory

```
user@Host:Images] cp ../Linux/linux-5.18.3/arch/arm/boot/zImage zImage.initrd
user@Host:Images]
```

- In this method the ramdisk image must be copied in RAM and the address where it is copied must be specified in boot arguments
- So, in order to do this, we need a bootloader that would help us to do the above mentioned
- But how will the ramdisk image be fetched by the Bootloader?
- Well, we have multiple possibilities, let's use the SD card approach to start with

```
user@Host:Images] qemu-img create boot.img 16M # Create a card image of 16M size
user@Host:Images] mkfs.vfat boot.img # Formatting it a FAT, generally boot files use this
type
user@Host:Images] sudo mount -o loop,rw,sync boot.img /media
user@Host:Images] sudo cp zImage.initrd vexpress-v2p-ca9.dtb rootfs.ext4 /media/
user@Host:Images] sudo umount /media
user@Host:Images]
```

- Let's run the u-boot and load all these contents

```
user@Host:Image] qemu-system-arm -M vexpress-a9 -nographic -kernel u-boot -drive
file=boot.img,if=sd,format=raw
U-Boot 2022.04 (Jun 09 2022 - 17:27:40 +0530)
DRAM:  128 MiB
WARNING: Caches not enabled
Core:  12 devices, 7 uclasses, devicetree: embed
Flash: 64 MiB
MMC:   mmci@5000: 0
Loading Environment from Flash... *** Warning - bad CRC, using default environment
In:    serial
Out:   serial
Err:   serial
Net:   eth0: ethernet@3,02000000
Hit any key to stop autoboot:  0
u-boot] # Note this prompt is because of custom image we built
```

- Observe the qemu options, we have used -drive instead of -sd, this is just to avoid the annoying warning which pops up using -sd option related to raw image format

```
u-boot] mmc info # Note depends on the interface
Device: mmci@5000
Manufacturer ID: aa
OEM: 5859
Name: QEMU!
Bus Speed: 12000000
Mode: MMC legacy
Rd Block Len: 512
SD version 2.0
High Capacity: No
Capacity: 16 MiB
Bus Width: 1-bit
Erase Group Size: 512 Bytes
u-boot] ls mmc 0 # List the file contents
  5115032   zImage.initrd
    14081   vexpress-v2p-ca9.dtb
  8388608   rootfs.ext4
3 file(s), 0 dir(s)
u-boot]
```

```
u-boot] fatload mmc 0:0 0x60200000 zImage.initrd # Type fatload and ENTER to know more on
options passed
5115032 bytes read in 1223 ms (4 MiB/s)
u-boot] fatload mmc 0:0 0x60100000 vexpress-v2p-ca9.dtb
14081 bytes read in 17 ms (808.6 KiB/s)
u-boot] fatload mmc 0:0 0x62000000 rootfs.ext4
8388608 bytes read in 1964 ms (4.1 MiB/s)
u-boot] setenv bootargs 'console=ttyAMA0 root=/dev/ram0 rw rootfstype=ext4
initrd=0x62000000,8388608'
u-boot] bootz 0x60200000 - 0x60100000
Kernel image @ 0x60200000 [ 0x000000 - 0x4e1d38 ]
## Flattened Device Tree blob at 60100000
   Booting using the fdt blob at 0x60100000
   Loading Device Tree to 67b1f000, end 67b25700 ... OK

Starting kernel ...
Booting Linux on physical CPU 0x0
Linux version 5.18.3
```

```
RAMDISK: ext2 filesystem found at block 0
RAMDISK: Loading 8192KiB [1 disk] into ram disk... |
done.
EXT4-fs (ram0): mounted filesystem with ordered data mode. Quota mode: disabled.
ext4 filesystem being mounted at /root supports timestamps until 2038 (0x7fffffff)
VFS: Mounted root (ext4 filesystem) on device 1:0.
Freeing unused kernel image (initmem) memory: 1024K
Run /sbin/init as init process
udevd[78]: starting version 3.2.5
random: crng init done
udevd[79]: starting eudev-3.2.5
EXT4-fs (ram0): re-mounted. Quota mode: disabled.

Poky (Yocto Project Reference Distro) 2.5 qemuarm /dev/ttyAMA0

qemuarm login:
```

5.1.20 **6. Deploy - Qemu (vexpress-a9)**

5.1.20.1 6.5. tftp

- Assuming you have all the images ready, lets try to get all those images on the target board via network with the help of TFTP protocol
- But to achieve this, we need to install some packages in our host system

```
user@Host:Images] sudo apt install tftpd-hpa tftp-hpa
```

- Once installed check whether the service in active and running

```
user@Host:Images] sudo service tftpd-hpa status
```

- It should be active and running, the directory path, where the files to be kept would be mentioned in the information, copy all the needed files into that path

```
user@Host:Images] sudo cp zImage.initrd vexpress-v2p-ca9.dtb rootfs.ext4 /srv/tftp
```

- Now let's proceed to the qemu target and run the needed commands to boot the board

```
user@Host:Image] sudo
user@Host:Image] sudo qemu-system-arm -M vexpress-a9 -nographic -kernel u-boot -net nic -
net tap,ifname=tap0
W: /etc/qemu-ifup: no bridge for guest interface found
U-Boot 2022.04 (Jun 09 2022 - 17:27:40 +0530)
DRAM:  128 MiB
WARNING: Caches not enabled
Core:  12 devices, 7 uclasses, devicetree: embed
Flash: 64 MiB
MMC:   mmci@5000: 0
Loading Environment from Flash... *** Warning - bad CRC, using default environment
In:    serial
Out:   serial
Err:   serial
Net:   eth0: ethernet@3,02000000
Hit any key to stop autoboot:  0
u-boot] # Note this prompt is because of custom image we built
```

- Observe the warning message in the previous slide, it's looking for a bridge interface
- So, on host system, open one more terminal / tab, and run the following command

```
user@Host:Images] sudo ifconfig tap0 192.167.7.1
```

- If ifconfig in not installed, you may install it as said in the "Host Setup" slides
- Come back to the target board and run the following commands

```
u-boot] setenv serverip 192.168.7.1
u-boot] setenv ipaddr 192.168.7.100
u-boot] ping 192.168.7.1
smc911x: detected LAN9118 controller
smc911x: phy initialized
smc911x: MAC 52:54:00:12:34:56
Using ethernet@3,02000000 device
smc911x: MAC 52:54:00:12:34:56
host 192.168.7.1 is alive
u-boot]
```

```
u-boot] tftpboot 0x60200000 zImage.initrd
smc911x: detected LAN9118 controller
smc911x: phy initialized
smc911x: MAC 52:54:00:12:34:56
Using ethernet@3,02000000 device
TFTP from server 192.168.7.1; our IP address is 192.168.7.100
Filename 'zImage'.
Load address: 0x60200000
Loading: ##################################################################
         ##################################################################
         ##################################################################
         ##################################################################
         ##################################################################
         ########################
         8.3 MiB/s
done
Bytes transferred = 5115032 (4e0c98 hex)
smc911x: MAC 5 2:54:00:12:34:56
u-boot]
```

```
u-boot] tftpboot 0x60100000 vexpress-v2p-ca9.dtb
smc911x: detected LAN9118 controller
smc911x: phy initialized
smc911x: MAC 52:54:00:12:34:56
Using ethernet@3,02000000 device
TFTP from server 192.168.7.1; our IP address is 192.168.7.100
Filename 'vexpress-v2p-ca9.dtb'.
Load address: 0x60200000
Loading: #
         2.7 MiB/s
done
Bytes transferred = 14081 (3701 hex)
smc911x: MAC 52:54:00:12:34:56
u-boot]
```

```
u-boot] tftpboot 0x62000000 rootfs.ext4
smc911x: detected LAN9118 controller
smc911x: phy initialized
smc911x: MAC 52:54:00:12:34:56
Using ethernet@3,02000000 device
TFTP from server 192.168.7.1; our IP address is 192.168.7.100
Filename 'rootfs.ext4'.
Load address: 0x60200000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         ##################################################
         8.4 MiB/s
done
Bytes transferred = 8388608 (800000 hex)
smc911x: MAC 52:54:00:12:34:56
u-boot]
```

```
u-boot] setenv bootargs 'console=ttyAMA0 root=/dev/ram0 rw rootfstype=ext4
initrd=0x62000000,8388608'
u-boot] bootz 0x60200000 - 0x60100000
Kernel image @ 0x60200000 [ 0x000000 - 0x4e1d38 ]
## Flattened Device Tree blob at 60100000
   Booting using the fdt blob at 0x60100000
   Loading Device Tree to 67b1f000, end 67b25700 ... OK

Starting kernel ...
Booting Linux on physical CPU 0x0
Linux version 5.18.3 (user@Host) (arm-linux-gnueabi-gcc (Ubuntu 9.4.0-
1ubuntu1~20.04.1) 9.4.0, GNU ld (GNU Binutils for Ubuntu) 2.34) #13 SMP Fri Jun 17
22:35:27 IST 2022
CPU: ARMv7 Processor [410fc090] revision 0 (ARMv7), cr=10c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT nonaliasing instruction cache
OF: fdt: Machine model: V2P-CA9
OF: fdt: Ignoring memory block 0x80000000 - 0x80000004
Memory policy: Dat
Res
```

```
RAMDISK: ext2 filesystem found at block 0
RAMDISK: Loading 8192KiB [1 disk] into ram disk... |
done.
EXT4-fs (ram0): mounted filesystem with ordered data mode. Quota mode: disabled.
ext4 filesystem being mounted at /root supports timestamps until 2038 (0x7fffffff)
VFS: Mounted root (ext4 filesystem) on device 1:0.
Freeing unused kernel image (initmem) memory: 1024K
Run /sbin/init as init process
udevd[78]: starting version 3.2.5
random: crng init done
udevd[79]: starting eudev-3.2.5
EXT4-fs (ram0): re-mounted. Quota mode: disabled.

Poky (Yocto Project Reference Distro) 2.5 qemuarm /dev/ttyAMA0

qemuarm login:
```

5.1.21 6. Deploy - Qemu (vexpress-a9)

5.1.21.1 6.6. NFS

- The best choice of booting the kernel for Application or Driver Module development is NFS Boot
- The necessary kernel configuration for this to work is enable by default
- But in the host side we must do some installation and configurations

```
user@Host:Images] sudo apt install nfs-kernel-server
```

- Add the following line in /etc/export file (add it as the last line)

```
/home/user/EmbeddedLinux/RootFS        *(rw,sync,no_root_squash,no_subtree_check)
```

- Restart the NFS server

```
user@Host:Images] sudo service restart nfs-kernel-server
```

- Now, lets take the advantage of booting the kernel with qemu directly to run the system

```
user@Host:Images] sudo qemu-system-arm -M vexpress-a9 -nographic -kernel zImage -dtb
vexpress-v2p-ca9.dtb -append "console=ttyAMA0 root=/dev/nfs rw ip=192.168.7.100
nfsroot=192.168.7.1:/home/adil/EmbeddedLinux/RootFS,nfsvers=3 rootdelay=10" -net nic -net
tap,ifname=tap0
W: /etc/qemu-ifup: no bridge for guest interface found
Booting Linux on physical CPU 0x0
Linux version 5.18.3 (user@Host) (arm-linux-gnueabi-gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1)
9.4.0, GNU ld (GNU Binutils for Ubuntu) 2.34) #11 SMP Thu Jun 16 13:01:01 IST 2022
```

- Now in the other terminal, make you configure the tap interface before the set timeout. In this case it's 10 secs

```
user@Host:Images] sudo ifconfig tap0 192.167.7.1
```

```
         Guessing netmask 255.255.255.0
IP-Config: Complete:
     device=eth0, hwaddr=52:54:00:12:34:56, ipaddr=192.168.7.100, mask=255.255.255.0,
gw=255.255.255.255
     host=192.168.7.100, domain=, nis-domain=(none)
     bootserver=255.255.255.255, rootserver=192.168.7.1, rootpath=
ALSA device list:
  #0: ARM AC'97 Interface PL041 rev0 at 0x10004000, irq 37
Waiting 3 sec before mounting root device...
VFS: Mounted root (nfs filesystem) on device 0:13.
Freeing unused kernel image (initmem) memory: 1024K
Run /sbin/init as init process
udevd[77]: starting version 3.2.5
random: crng init done
udevd[78]: starting eudev-3.2.5

Poky (Yocto Project Reference Distro) 2.5 qemuarm /dev/ttyAMA0

qemuarm login:
```

5.1.22 **6. Deploy - Qemu (vexpress-a9)**

5.1.22.1 6.7. Multi Partition using SD / eMMC

- In general, we may need multiple partition on on memory to store different types of data like, bootable images and RFS data
- We may have to store them in different formats as per the requirements of ROM bootloader dependency
- Hence, let's see how we can emulate multiple partition using Qemu.
- In our case let's assume that we need 2 partitions, one to store bootable images and the other to store RFS data
- First, lets create a card image using qemu-img command as shown below

```
user@Host:Images] qemu-img create sd.img 128M # Create a card image of 128M size
Formatting 'sd.img', fmt=raw size=134217728
user@Host:Images]
```

- We need to create 2 partition on the created sd image, You may use fdisk command to do that
- **Note: You must be cautious in these steps, any misstep could lead to corruption your system**

```
user@Host:Images] fdisk sd.img
Welcome to fdisk (util-linux 2.34).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.

Device does not contain a recognized partition table.
Created a new DOS disklabel with disk identifier 0x3528211e.

Command (m for help):
```

```
Command (m for help): p
Disk sd.img: 128 MiB, 134217728 bytes, 262144 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x3528211e

Command (m for help):
```

- The commands and the inputs used here are highlighted in RED color, in data part you may press ENTER to key default
- Let's keep the first partition size as 16M for bootable images and the remaining space for RFS data, but this all depends on the requirements

```
Command (m for help): n
Partition type
   p   primary (0 primary, 0 extended, 4 free)
   e   extended (container for logical partitions)
Select (default p): p
Partition number (1-4, default 1): 1
First sector (2048-262143, default 2048): 2048
Last sector, +/-sectors or +/-size{K,M,G,T,P} (2048-262143, default 262143): +16M
Created a new partition 1 of type 'Linux' and of size 16 MiB.
Command (m for help): n
Partition type
   p   primary (1 primary, 0 extended, 3 free)
   e   extended (container for logical partitions)
Select (default p): p
Partition number (2-4, default 2): 2
First sector (34816-262143, default 34816): 34816
Last sector, +/-sectors or +/-size{K,M,G,T,P} (34816-262143, default 262143): 262143
Created a new partition 2 of type 'Linux' and of size 111 MiB.
```

```
Command (m for help): p
Disk sd.img: 128 MiB, 134217728 bytes, 262144 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x3528211e

Device     Boot Start    End Sectors  Size Id Type
sd.img1          2048  34815   32768   16M 83 Linux
sd.img2         34816 262143  227328  111M 83 Linux

Command (m for help): t
Partition number (1,2, default 2): 1
Hex code (type L to list all codes): c
Changed type of partition 'Linux' to 'W95 FAT32 (LBA)'.
Command (m for help): a
Partition number (1,2, default 2): 1
The bootable flag on partition 1 is enabled now.
```

```
Command (m for help): p
Disk sd.img: 128 MiB, 134217728 bytes, 262144 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x3528211e

Device     Boot Start    End Sectors  Size Id Type
sd.img1    *     2048  34815   32768   16M  c W95 FAT32 (LBA)
sd.img2         34816 262143  227328  111M 83 Linux

Command (m for help): w
The partition table has been altered.
Syncing disks.

user@Host:Images]
```

- Now we a SD image with 2 partitions, we to need format and mount them to copy the needed files
- But the questions how would we mount 2 partitions of a single image?, well let's use losetup command to achieve it

```
user@Host:Images] sudo losetup -Pf sd.img
user@Host:Images] lsblk /dev/loop0
NAME      MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
loop0       7:0    0  128M  0 loop
├─loop0p1 259:0    0   16M  0 part
└─loop0p2 259:1    0  111M  0 part
user@Host:Images]
```

- First let's format the partition 1 as FAT

```
user@Host:Images] sudo mkfs.vfat /dev/loop0p1
mkfs.fat 4.1 (2017-01-24)
user@Host:Images]
```

- Mount the partition and copy all the boot related files

```
user@Host:Images] sudo mount /dev/loop0p1 /media/
user@Host:Images] sudo cp zImage vexpress-v2p-ca9.dtb /media/
user@Host:Images] sudo umount /media
```

- Now let's format the second partitions as ext4 and un-compress the RFS data into it

```
user@Host:Images] sudo mkfs.ext4 /dev/loop0p2
mke2fs 1.45.5 (07-Jan-2020)
Discarding device blocks: done
Creating filesystem with 28416 4k blocks and 28416 inodes

Allocating group tables: done
Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done
user@Host:Images] sudo mount /dev/loop0p2 /media/
user@Host:Images] sudo tar xvf ../Sources/core-image-minimal-qemuarm.tar.bz2 -C /media/
user@Host:Images] sudo umount /media
```

- Let's delete the loop device created

```
user@Host:Images] sudo sudo losetup -d /dev/loop0
user@Host:Images] lsblk # Observe the loop devices are gone
NAME MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda    8:0    0  256G  0 disk
sdb    8:16   0  256G  0 disk /
user@Host:Images]
```

- Now let's boot our system with the help of bootloader, Please follow the steps as put in the next slide

LINUX EMBEDDED SYSTEM

```
user@Host:Image] qemu-system-arm -M vexpress-a9 -nographic -kernel u-boot -drive
file=sd.img,if=sd,format=raw
U-Boot 2022.04 (Jun 09 2022 - 17:27:40 +0530)
DRAM:  128 MiB
WARNING: Caches not enabled
Core:  12 devices, 7 uclasses, devicetree: embed
Flash: 64 MiB
MMC:   mmci@5000: 0
Loading Environment from Flash... *** Warning - bad CRC, using default environment
In:    serial
Out:   serial
Err:   serial
Net:   eth0: ethernet@3,02000000
Hit any key to stop autoboot:  0
u-boot] # Note this prompt is because of custom image we built
```

```
u-boot] fatload mmc 0:1 0x60200000 zImage
5119288 bytes read in 1850 ms (2.6 MiB/s)
u-boot] fatload mmc 0:1 0x60100000 vexpress-v2p-ca9.dtb
14081 bytes read in 17 ms (808.6 KiB/s)
u-boot] setenv bootargs 'console=ttyAMA0 root=/dev/mmcblk0p2 rw rootfstype=ext4'
u-boot] bootz 0x60200000 - 0x60100000
Kernel image @ 0x60200000 [ 0x000000 - 0x4e1d38 ]
## Flattened Device Tree blob at 60100000
   Booting using the fdt blob at 0x60100000
   Loading Device Tree to 67b1f000, end 67b25700 ... OK

Starting kernel ...
Booting Linux on physical CPU 0x0
Linux version 5.18.3 (adil@MYTSL01490) (arm-linux-gnueabi-gcc (Ubuntu 9.4.0-
1ubuntu1~20.04.1) 9.4.0, GNU ld (GNU Binutils for Ubuntu) 2.34) #13 SMP Fri Jun 17
22:35:27 IST 2022
CPU: ARMv7 Processor [410fc090] revision 0 (ARMv7), cr=10c5387d
CPU: PIPT / VIPT    liasin    ta cache, VIPT nona        instructi
OF:
```

LINUX EMBEDDED SYSTEM

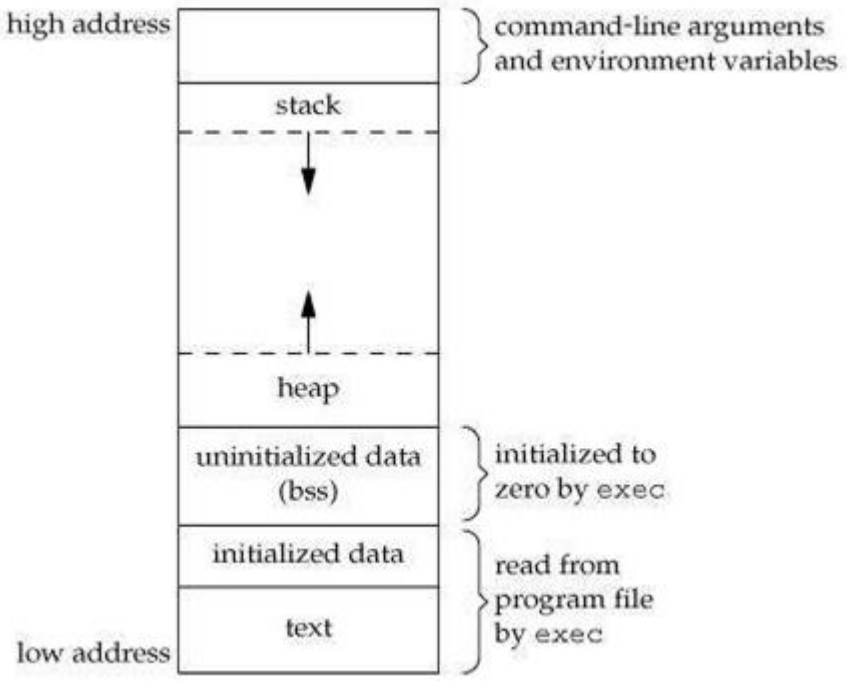
```
bus@40000000:iofpga@7,00000000/10007000.kmi/serio1/input/input2
drm-clcd-pl111 10020000.clcd: DVI muxed to daughterboard 1 (core tile) CLCD
drm-clcd-pl111 10020000.clcd: initializing Versatile Express PL111
EXT4-fs (mmcblk0p2): mounted filesystem with ordered data mode. Quota mode: disabled.
ext4 filesystem being mounted at /root supports timestamps until 2038 (0x7fffffff)
VFS: Mounted root (ext4 filesystem) on device 179:2.
Freeing unused kernel image (initmem) memory: 1024K
Run /sbin/init as init process
random: crng init done
udevd[77]: starting version 3.2.5
udevd[78]: starting eudev-3.2.5
EXT4-fs (mmcblk0p2): re-mounted. Quota mode: disabled.

Poky (Yocto Project Reference Distro) 2.5 qemuarm /dev/ttyAMA0

qemuarm login:
```

In practical words, when we run any C-program, its executable image is loaded into RAM of computer in an organized manner.

This memory layout is organized in following fashion :-



1>Text or Code Segment :-

Text segment contains machine code of the compiled program. Usually, the text segment is **sharable** so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. The text segment of an

executable object file is often **read-only segment** that prevents a program from being accidentally modified.

2>Initialized Data Segment :-

Initialized data stores all `global, static, constant`, and external variables **(** declared with `extern` keyword **)** that are initialized beforehand. Data segment is **not read-only**, since the values of the variables can be altered at run time.

This segment can be further classified into initialized **read-only area** and initialized **read-write area**.

```
#include <stdio.h>

char c[]="rishabh tripathi";     /*  global variable stored in Initialized
Data Segment in read-write area*/
const char s[]="HackerEarth";    /* global variable stored in Initialized
Data Segment in read-only area*/

int main()
{
    static int i=11;          /* static variable stored in Initialized Data
Segment*/
    return 0;
}
```

3>Uninitialized Data Segment (bss) :-

Data in this segment is initialized to arithmetic `0` before the program starts executing. Uninitialized data starts at the end of the data segment and contains all `global` variables and `static` variables that are initialized to `0` or do not have explicit initialization in source code.

```
#include <stdio.h>

char c;               /* Uninitialized variable stored in bss*/

int main()
{
    static int i;     /* Uninitialized static variable stored in bss */
    return 0;
}
```

4>Heap :-

Heap is the segment where ***dynamic memory allocation*** usually takes place. When some more memory need to be allocated using `malloc` and `calloc` function, heap grows upward.

LINUX EMBEDDED SYSTEM

The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

```
#include <stdio.h>
int main()
{
    char *p=(char*)malloc(sizeof(char));     /* memory allocating in heap
segment */
    return 0;
```

```
}
```

5>Stack :-

Stack segment is used to store all **local variables** and is used for **passing arguments** to the functions along with the return address of the instruction which is to be executed after the function call is over. Local variables have a scope to the block which they are defined in, they are created when control enters into the block. All recursive function calls are added to stack.

The stack and heap are traditionally located at opposite ends of the process's virtual address space.