./

Learning Report

**Embedded Linux**



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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 |  | 99003733/Aditya Mewara |  |  |  |

**Document History**

Table of Contents

[1. Embedded Linux- 5](#_Toc68709972)

[Evolution of Linux OS 5](#_Toc68709973)

[The 5](#_Toc68709974)

[1.1 Kernel 6](#_Toc68709975)

[1.2 System Libraries 7](#_Toc68709976)

[System libraries are special programs that help in accessing the kernel's features. A kernel has to be triggered to perform a task, and this triggering is done by the applications. But applications must know how to place a system call because each kernel has a different set of system calls. Programmers have developed a standard library of procedures to communicate with the kernel. Each operating system supports these standards, and then these are transferred to system calls for that operating system 7](#_Toc68709977)

[1.3 System Tools 7](#_Toc68709978)

[1.4 Development Tools 7](#_Toc68709979)

[1.5 End User Tools 7](#_Toc68709980)

[2. Qemu 8](#_Toc68709981)

[ QEMU (short for Quick EMUlator is a free and open-source emulator and virtualizer that can perform hardware virtualization. 8](#_Toc68709982)

[ QEMU is a hosted virtual machine monitor: it emulates the machine's processor through dynamic binary translation and provides a set of different hardware and device models for the machine, enabling it to run a variety of guest operating systems. It also can be used with Kernel-based Virtual Machine (KVM) to run virtual machines at near-native speed (by taking advantage of hardware extensions such as Intel VT-x). QEMU can also do emulation for user-level processes, allowing applications compiled for one architecture to run on another. 8](#_Toc68709983)

[ User-mode emulation 8](#_Toc68709984)

[In this mode QEMU runs single Linux or Darwin/macOS programs that were compiled for a different instruction set. System calls are thunked for endianness and for 32/64 bit mismatches. Fast cross-compilation and cross-debugging are the main targets for user-mode emulation. 8](#_Toc68709985)

[ System emulation 8](#_Toc68709986)

[In this mode QEMU emulates a full computer system, including peripherals. It can be used to provide virtual hosting of several virtual computers on a single computer. QEMU can boot many guest operating systems, including Linux, Solaris, Microsoft Windows, DOS, and BSD;[6] it supports emulating several instruction sets, including x86, MIPS, 32-bit ARMv7, ARMv8, PowerPC, SPARC, ETRAX CRIS and MicroBlaze. 8](#_Toc68709987)

[ KVM Hosting 8](#_Toc68709988)

[Here QEMU deals with the setting up and migration of KVM images. It is still involved in the emulation of hardware, but the execution of the guest is done by KVM as requested by QEMU. 8](#_Toc68709989)

[ Xen Hosting 8](#_Toc68709990)

[QEMU is involved only in the emulation of hardware; the execution of the guest is done within Xen and is totally hidden from QEMU. 8](#_Toc68709991)

[2.2 Rootfs 9](#_Toc68709992)

[2.3 Toolchain 9](#_Toc68709993)

[2.4 Your First Boot (Emulation) 9](#_Toc68709994)

[3. Building Custom Kernel (Qemu) 10](#_Toc68709995)

[3.3 Obtain Configuration File 10](#_Toc68709996)

[3.5 Build the kernel 11](#_Toc68709997)

[3.6 Test the Built Outcome 11](#_Toc68709998)

[4. Cross Compiling Code 12](#_Toc68709999)

[4.1 Simple Hello Module 12](#_Toc68710000)

[4.2 Static Library 13](#_Toc68710001)

[4.3 Dynamic Linking 13](#_Toc68710002)

[5. Working with U-Boot 14](#_Toc68710003)

[5.1 Cross Building 14](#_Toc68710004)

[5.2 Simple Boot – Rootfs in SD Card 14](#_Toc68710005)

[5.3 Prepare Partitioned SD Card 15](#_Toc68710006)

[5.4 Rootfs in partitioned SD Card 15](#_Toc68710007)

[5.5 Setup TFTP on Host 15](#_Toc68710008)

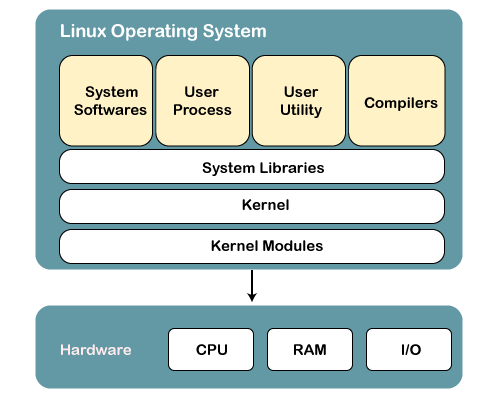
[5.6 TFTP Boot on Target 16](#_Toc68710009)

# Embedded Linux-

* Embedded Linux is a type of Linux operating system/kernel that is designed to be installed and used within embedded devices and appliances.
* It is a compact version of Linux that offers features and services in line with the operating and application requirement of the embedded system.

### Evolution of Linux OS

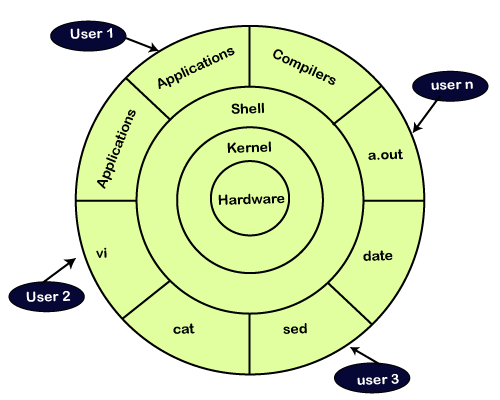
The [Linux OS](https://www.javatpoint.com/linux-tutorial) was developed by Linus Torvalds in 1991, which sprouted as an idea to improve the UNIX OS. He suggested improvements but was rejected by UNIX designers. Therefore, he thought of launching an OS, designed in a way that could be modified by its users.



**Figure: Linux Embedded**

## **1.1 Kernel**

Linux kernel is the core part of the operating system. It establishes communication between devices and software. Moreover, it manages system resources. It has four responsibilities:



**Figure : Kernel**

**Device management:** A system has many devices connected to it like CPU, a memory device, sound cards, graphic cards, etc. A kernel stores all the data related to all the devices in the device driver (without this kernel won't be able to control the devices). Thus kernel knows what a device can do and how to manipulate it to bring out the best performance. It also manages communication between all the devices. The kernel has certain rules that have to be followed by all the devices.

**Memory management:** Another function that kernel has to manage is the memory management. The kernel keeps track of used and unused memory and makes sure that processes shouldn't manipulate data of each other using virtual memory addresses.

**Process management:** In the process, management kernel assigns enough time and gives priorities to processes before handling CPU to other processes. It also deals with security and ownership information.

**Handling system calls:** Handling system calls means a programmer can write a query or ask the kernel to perform a task.

## **1.2 System Libraries**

System libraries are special programs that help in accessing the kernel's features. A kernel has to be triggered to perform a task, and this triggering is done by the applications. But applications must know how to place a system call because each kernel has a different set of system calls. Programmers have developed a standard library of procedures to communicate with the kernel. Each operating system supports these standards, and then these are transferred to system calls for that operating system.

The most well-known system library for Linux is Glibc (GNU C library).

## **1.3 System Tools**

* Linux OS has a set of utility tools, which are usually simple commands. It is a software which GNU project has written and publish under their open source license so that software is freely available to everyone.
* With the help of commands, you can access your files, edit and manipulate data in your directories or files, change the location of files, or anything.

## **1.4 Development Tools**

With the above three components, your OS is running and working. But to update your system, you have additional tools and libraries. These additional tools and libraries are written by the programmers and are called toolchain. A toolchain is a vital development tool used by the developers to produce a working application.

## **1.5 End User Tools**

* These end tools make a system unique for a user. End tools are not required for the operating system but are necessary for a user.
* Some examples of end tools are graphic design tools, office suites, browsers, multimedia players, etc.

# Qemu

## QEMU (short for Quick EMUlator is a [free and open-source](https://en.wikipedia.org/wiki/Free_and_open-source) [emulator](https://en.wikipedia.org/wiki/Emulator) and [virtualizer](https://en.wikipedia.org/wiki/Hypervisor) that can perform [hardware virtualization](https://en.wikipedia.org/wiki/Hardware_virtualization).

## QEMU is a [hosted virtual machine monitor](https://en.wikipedia.org/wiki/Virtual_machine_monitor" \l "Classification): it emulates the machine's [processor](https://en.wikipedia.org/wiki/Central_processing_unit) through dynamic [binary translation](https://en.wikipedia.org/wiki/Binary_translation) and provides a set of different hardware and device models for the machine, enabling it to run a variety of [guest operating systems](https://en.wikipedia.org/wiki/Guest_operating_system). It also can be used with [Kernel-based Virtual Machine](https://en.wikipedia.org/wiki/Kernel-based_Virtual_Machine) (KVM) to run virtual machines at near-native speed (by taking advantage of hardware extensions such as [Intel VT-x](https://en.wikipedia.org/wiki/X86_virtualization)). QEMU can also do emulation for user-level processes, allowing applications compiled for one architecture to run on another.

**QEMU has multiple operating modes** :-

## **User-mode emulation**

## In this mode QEMU runs single [Linux](https://en.wikipedia.org/wiki/Linux) or [Darwin](https://en.wikipedia.org/wiki/Darwin_(operating_system))/[macOS](https://en.wikipedia.org/wiki/MacOS) programs that were compiled for a different [instruction set](https://en.wikipedia.org/wiki/Instruction_set). System calls are [thunked](https://en.wikipedia.org/wiki/Thunk) for [endianness](https://en.wikipedia.org/wiki/Endianness) and for 32/64 bit mismatches. Fast cross-compilation and cross-debugging are the main targets for user-mode emulation.

## **System emulation**

## In this mode QEMU emulates a full computer system, including [peripherals](https://en.wikipedia.org/wiki/Peripheral). It can be used to provide virtual hosting of several virtual computers on a single computer. QEMU can boot many guest [operating systems](https://en.wikipedia.org/wiki/Operating_system), including [Linux](https://en.wikipedia.org/wiki/Linux), [Solaris](https://en.wikipedia.org/wiki/Solaris_(operating_system)), [Microsoft Windows](https://en.wikipedia.org/wiki/Microsoft_Windows), [DOS](https://en.wikipedia.org/wiki/DOS), and [BSD](https://en.wikipedia.org/wiki/BSD);[[6]](https://en.wikipedia.org/wiki/QEMU" \l "cite_note-yfwuu-6) it supports emulating several instruction sets, including [x86](https://en.wikipedia.org/wiki/X86), [MIPS](https://en.wikipedia.org/wiki/MIPS_architecture), 32-bit [ARMv7](https://en.wikipedia.org/wiki/ARMv7), [ARMv8](https://en.wikipedia.org/wiki/ARMv8), [PowerPC](https://en.wikipedia.org/wiki/PowerPC), [SPARC](https://en.wikipedia.org/wiki/SPARC), [ETRAX CRIS](https://en.wikipedia.org/wiki/ETRAX_CRIS) and [MicroBlaze](https://en.wikipedia.org/wiki/MicroBlaze).

## **KVM Hosting**

## Here QEMU deals with the setting up and migration of [KVM](https://en.wikipedia.org/wiki/Kernel-based_Virtual_Machine) images. It is still involved in the emulation of hardware, but the execution of the guest is done by KVM as requested by QEMU.

## **Xen Hosting**

## QEMU is involved only in the emulation of hardware; the execution of the guest is done within [Xen](https://en.wikipedia.org/wiki/Xen) and is totally hidden from QEMU.

* 1. **Setup Qemu**

**Install Qemu, a full system emulator for ARM target architecture**

* sudo apt install qemu-system-arm
* qemu-system-arm –v
* qemu-system-arm –M ?
* qemu-system-aarch64 -v

## **2.2 Rootfs**

* 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
* Align the size of rootfs

e2fsck -f rootfs.img resize2fs

rootfs.img 16M

## **2.3 Toolchain**

**Install linaro toolchain from ubuntu package manager**

* sudo apt install gcc-arm-linux-gnueabi # soft float
* sudo apt install gcc-arm-linux-gnueabihf # hard float

## **2.4 Your First Boot (Emulation)**

* Collect prebuilt zImage, vexpress-v2p-ca9.dtb from faculty
* Ensure rootfs.img is also in same location
* Emulate using Qemu – sdcard 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"

**--------------------------------------------------------------------------------------------------------------------------------------------**

**2.5 First Steps on Target**

* uname –r
* uname –v
* uname –a
* cat /proc/cpuinfo
* free –m
* df –kh
* mount
* dmesg

# Building Custom Kernel (Qemu)

A **kernel** is an important program of every device out there. **Android** is a famous operating system that features a lot of **custom kernel** out there for almost every phone nowadays. **Custom Kernels** not only offer security updates, but also various improvements over the Stock **Kernel**.

**3.2 Download Kernel Source**

* Download any recent LTS version of kernel source
* Let’s go with 4.14.x for now, for better compatibility with Qemu

wget <https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.14.202.tar.xz tar –xvf linux-4.14.202.tar.xz>

## **3.3 Obtain Configuration File**

* Locate default config available in KSRC/arch/arm/configs, we’ll refer vexpress\_defconfig for Versatil Express target being used for Qemu emulation
* Or collect any well tested configuration file as base config.

make ARCH=arm mrproper

make ARCH=arm vexpress\_defonfig

**(or)**

# copy custom config file as .config under KSRC

* Please note that mrproper will remove built files, including the configuration. So run this only for any new build.

**3.4 Further Customization**

* Run menuconfig for further customization
* Resolve any host dependencies at this stage, e.g. libncurses5-dev, flex, bison etc.

**make ARCH=arm menuconfig**

Let’s do these minimal changes for now

* General Setup -> Local Version -> "-custom“

Device Drivers -> Block Devices ->

* Enable RAM Block device support

Increase default RAM disk size to suitable limit, say 65536

* Enable the block layer

Support for large (2TB+)

## **3.5 Build the kernel**

* Run menuconfig for further customization
* Build kernel image

make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabi- zImage -j <n>

* Build Device Tree Binaries

make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabi- dtbs firmware

## **3.6 Test the Built Outcome**

* Collect built outcome to a temporary location

# switch to KSRC

* + cp $KSRC/arch/arm/boot/zImage
  + cp $KSRC/arch/arm/boot/dts/\*.dtb

Ensure rootfs.img is also in same location

Emulate using Qemu

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"

# In Target

* uname –r
* uname –v
* ls /boot
* ls /lib/modules

# In Host

* ls –lh $KSRC/arch/arm/boot/zImage
* ls –lh $KSRC/vmlinux

# Cross Compiling Code

**Cross**-**compilation** is the act of **compiling code** for one computer system (often known as the target) on a different system, called the host. It's a very useful technique, for instance when the target system is too small to host the **compiler** and all relevant files.

## **4.1 Simple Hello Module**

------ # **hello.c**

#include <stdio.h>

int main() {

printf("Hello World\n");

return 0; }

#**Makefile**

arm-linux-gnueabi-gcc hello.c –o h1.out

arm-linux-gnueabi-gcc hello.c –o h2.out –o static

file h1.out h2.out

ls –lh h1.out h2.out

ldd h1.out

ldd h2.out

#**Mount and Unmount Process**

# copy h1.out, h2.out to target rootfs

sudo mount –o loop,rw.sync rootfs.img /mnt/rootfs

sudo cp h1.out h2.out /mnt/rootfs/home/root

sudo umount /mnt/rootfs

----- # **Multi programming**

//**test.c** \\ **sum.c** \\ **sqr.c**

int main() { int sum(int x,int y) int square(int x)

int a,b,c,d; { {

a=10,b=20; return x + y; return x \* x;

c=sum(a,b); } }

d=square(a);

//print (c,d );

}

# **Commands**

arm-linux-gnueabi-gcc test.c –c

arm-linux-gnueabi-gcc sum.c -c

arm-linux-gnueabi-gcc sqr.c –c

arm-linux-gnueabi-gcc test.o sum.o sqr.o –o all.out

## **4.2 Static Library**

**#** prepare the source code and generate .o files as earlier

arm-linux-gnueabi-ar sum.o sqr.o –o libsample.a

arm-linux-gnueabi-gcc –L. test.o –lsample –o s1.out

arm-linux-gnueabi-gcc –L. test.o –lsample –o s2.out -static

**#** copy s1.out, s2.out to target rootfs and test

file s1.out s2.out

# compare size of s1.out, s2.out

ldd s1.out s2.out

# check size of s2.out before strip

arm-linux-gnueabi-strip s2.out

# check size of s2.out after strip

## **4.3 Dynamic Linking**

# On Host arm-linux-gnueabi-gcc –shared libsample.so sum.o sqr.o

arm-linux-gnueabi-gcc –L. test.o –lsample –o d1.out

# copy libsample.so, d1.out to target rootfs and execute

-----------------------------------------------------------------------

# On Target

LD\_LIBRARY\_PATH=. ./d1.out

--------------------------------------

file d1.out

# check size of d1.out

ldd d1.out

# Working with U-Boot

Download tarball from ftp://ftp.denx.de/pub/u-boot/ and extract, choose any stable version like u-boot-2020.10.tar.bz2

Let's call the cloned/extracted source as USRC

## **5.1 Cross Building**

make ARCH=arm vexpress\_ca9x4\_defconfig

make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabi-

# Locate generated u-boot and copy to a tempdir

## **5.2 Simple Boot – Rootfs in SD Card**

qemu-img create simplesd.img 64M

sudo mkfs.vfat simplesd.img

sudo mount -o loop,rw,sync simplesd.img /mnt/sdcard

# copy zImage, vexpress-v2p-ca9.dtb, rootfs.img to /mnt/sdcard umount /mnt/sdcard

# copy simplesd.img to tempdir, where generated u-boot is copied

--------------------------------------------------------------------------------------------------------------

qemu-system-arm -M vexpress-a9 -m 1024 -serial stdio -kernel u-boot -sd sdcard.img

# Stop autoboot by hitting any key, Run the following commands in U-Boot shell

mmcinfo fatls mmc 0:0

fatload mmc 0:0 0x60200000 zImage

fatload mmc 0:0 0x60100000 vexpress-v2p-ca9.dtb

fatload mmc 0:0 0x62000000 rootfs.img

setenv bootargs 'console=ttyAMA0 root=/dev/ram0 rw rootfstype=ext4 initrd=0x62000000, 16777216’

bootz 0x60200000 - 0x60100000

# 16777216 is size of loaded rootfs image

# for this method ramdisk support should be enabled at kernel level (menuconfig --> Device Drivers --> Block Devices --> RAM Block Device Support)

## **5.3 Prepare Partitioned SD Card**

#Step By Step-------------->>

dd if=/dev/zero of=sdcard.img bs=1M count=128

# create two primary partitions in sdcard.img using cfdisk

# Keep first partition size as small as possible, say 16M

sudo fdisk -l sdcard.img # 1048576 is 2048x512, 2048 is start of first

partition # 17825792 is 34816x512, 34816 is start of second partition

sudo losetup -o 1048576 /dev/loop20 sdcard.img

sudo losetup -o 17825792 /dev/loop21 sdcard.img

sudo mkfs.vfat /dev/loop20 sudo mkfs.ext4 /dev/loop21

sudo mount -o loop,rw,sync /dev/loop20 /mnt/boot

sudo mount -o loop,rw,sync /dev/loop21 /mnt/rootfs

#copy zImage, vexpress-v2p-ca9.dtb to /mnt/boot

# extract core-image-minimal-qemuarm.tar.bz2 to /mnt/rootfs

tar -jxvf core-image-minimal-qemuarm.tar.bz2 -C /mnt/rootfs

sudo umount /mnt/boot

sudo umount /mnt/rootfs

sudo losetup -d /dev/loop20

sudo losetup -d /dev/loop21

## **5.4 Rootfs in partitioned SD Card**

qemu-system-arm -M vexpress-a9 -m 1024 -serial stdio -kernel u-boot -sd sdcard.img

#Stop autoboot by hitting any key, Run the following commands in U-Boot shell

mmcinfo Fatls mmc 0:1

fatload mmc 0:1 0x60200000 zImage

fatload mmc 0:1 0x60100000 vexpress-v2p-ca9.dtb

setenv bootargs 'console=ttyAMA0 root=/dev/mmcblk0p2 rw rootfstype=ext4’ bootz 0x60200000 - 0x60100000

## **5.5 Setup TFTP on Host**

sudo apt install tftpd

# create /etc/xinetd.d/tftp

# with specified content

# replace server\_args as per your machine /etc/init.d/xinetd restart

--------------------------------------------------------------------------------------------------------

service tftp

{

protocol = udp

port = 69

socket\_type = dgram

wait = yes

user = nobody

server = /usr/sbin/in.tftpd

server\_args = /\* \*/

disable = no

}

---------------------------------------------------------------------------------------------------------

**For Check tftp Commands** ->

sudo modprobe tun

sudo ifconfig tap0 192.168.0.1

## **5.6 TFTP Boot on Target**

--------------------------------------------------------------------------------------------------------------------------------------

sudo qemu-system-arm -M vexpress-a9 -m 256 -kernel u-boot -serial stdio \ -sd sdcard.img -net nic -net tap,ifname=tap0

--------------------------------------------------------------------------------------------------------------------------------------

setenv ipaddr 192.168.0.2

setenv serverip 192.168.0.1

ping 192.168.0.1

tftp 0x60200000 zImage

tftp 0x60100000 vexpress-v2p-ca9.dtb

setenv bootargs 'console=ttyAMA0 root=/dev/mmcblk0p2 rootfstype=ext4’ bootz 0x60200000 - 0x60100000