# 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.

Embedded Linux, though utilizing the same Linux kernel, is quite different from the standard Linux OS. Embedded Linux is specifically customized for embedded systems. Therefore, it is has a much smaller size, requires less processing power and has minimal features. Based on the requirements of the underlying embedded system, the Linux kernel is modified and optimized as an embedded Linux version. Such an instance of Linux can only run device-specific purpose-built applications.

Android OS is a type of embedded Linux, customized to be used on smartphones. Other devices on which embedded Linux is used include:

* Smart TVs
* Wireless routers
* Tablet PCs
* Navigation devices
* Other industrial and consumer electronic equipment

An embedded system is a small computer that lives within a larger structure that isn’t itself a computer. It is a bundle of computing hardware and software, designed for a specific function, that lives within a larger physical system. Rooted in a microprocessor or microcontroller, embedded systems are controlled by a real-time operating system, run on a limited amount of power and memory, and range widely in size and UI complexity. Embedded systems are all around us, existing within a vast array of consumer, industrial, medical, and military technologies.

An embedded Linux refers to a scenario where an embedded system runs on an operating system based on the Linux kernel. This Linux distribution will be specifically designed for an embedded system; it will have a smaller size than normal, possessing fewer features and less processing power. Such an instance of Linux can only run device-specific purpose-built applications.

Linux is flexible, low-cost and open source, and it has already been ported to custom-purpose microprocessors. Compared to proprietary embedded operating systems, Linux allows for multiple suppliers of software, development and support; it has a stable kernel; and it facilitates the ability to read, modify and redistribute the source code. Linux has many supported chip architectures, and so can run on devices as small as sockets and as large as mainframes. Linux enables a highly modular building-block approach to constructing a custom embeddable system, meaning added flexibility.

For these reasons, Linux has become very popular within embedded systems such as smartphones and tablets, and smart TVs and routers. Perhaps the best example of an embedded Linux is Android, developed by Google. Android is based on a modified Linux kernel, and released under an open source license. Amongst the better-known other small footprint embedded Linux versions are:

* **ETLinux**. A complete Linux distribution designed to run on small industrial computers.
* **LEM.** A small (<8 MB) multi-user, networked Linux version that runs on 386s.
* **LOAF**. "Linux On A Floppy" distribution that runs on 386s.
* **uClinux**. Linux for systems without MMUs.
* **uLinux.** Tiny Linux distribution that runs on 386s.
* **ThinLinux**. A minimized Linux distribution for dedicated camera servers, X-10 controllers, MP3 players, and other embedded applications.

Automotive computing, medical devices, Internet of Things (IoT), and factory automation are just a few of the embedded systems running on embedded Linux. We know that computers run on using specific hardware (e.g. CPU, memory, motherboard), and embedded system devices also run on microprocessors and controllers but at a smaller scale. These devices need an operating system that use fewer resources, and developers must be able to deploy to the operating system from their desktops.

Embedded Linux is often the choice for IoT developers due to its low cost, open-source code, lightweight storage requirements, and its ability to run on fewer resources. It’s a stable option, which is critical for organizations who deploy systems used by several consumers. The worry of OS crashes are non-existent. For example, the Android operating system is based on Linux and [powers 87%](https://www.statista.com/statistics/272307/market-share-forecast-for-smartphone-operating-systems/#:~:text=Smartphones%20running%20the%20Android%20operating,percent%20share%20of%20the%20market.) of the global market share, and the OS is very reliable.

A proprietary real-time operating system (RTOS) can be an attractive alternative to embedded Linux, but it comes with its own difficulties. When developers seek out IoT development platforms and methods, they will come across the options of embedded Linux or an RTOS. There are distinct differences between the two, so you must choose carefully before being married to one or the other.

RTOS can be more developer friendly, but for enterprise applications where you want to support customizations or integrate with other developer projects, embedded Linux provides a more intuitive integration environment. An RTOS is designed for time-sensitive applications. Unless you have a system that has critical time-sensitive events, an embedded Linux system provides more flexibility for developers.

Most developers know that you need to compile source code for the target system. When developing for embedded systems, developers write code on a desktop but deploy to another environment using cross-platform toolchains. Embedded system deployments are much different than desktops. Desktop computers target 8086 microprocessor architectures, but embedded systems can target several microprocessors, usually ARM, 8051, PIC and AVR.

A few other differences between Desktop Linux and Embedded Linux include:

* Embedded systems use less power and usually run on batteries.
* Embedded systems are much more compact.
* Many embedded systems power critical devices such as health alert systems, medical equipment, nuclear machinery, and home security.
* An embedded Linux system contains just the components needed to function on the target processor rather than all possible drivers and libraries to run on a myriad of components installed on a desktop.

Developers choose Linux as their embedded system target operating system for a variety of reasons. The first one is that Linux is completely customizable. Developers can freely take the Linux kernel and create their own distribution of the operating system or take an existing distribution and add small changes that create their own version.

Linux is free, so it has a low startup cost for new developers who just want to explore and get started. The more developers who work with Linux, the more likely it is to be embedded into systems.

Finally, it’s widely used making it better supported by third-party developers. If the goal is to build a system that other developers could integrate into their own projects, Linux will be more intuitive.

Choosing an embedded Linux operating system depends highly on your goals, the target device operations, time to market and responsiveness required. Many developers simply go with Android, because it’s widely used and understood, but here are a few questions to consider to help you choose the right OS:

* **Is timing critical?** If you have an application that needs precise timing, a real-time operating system should be used. Linux is not an RTOS, but extensions such as [RTLinux](https://en.wikipedia.org/wiki/RTLinux) can be considered.
* **Is memory size and CPU power limited?** A typical Linux kernel is 1.5MB, which could be too large for some systems. A customized Linux kernel with only the right components for application functionality can be used.
* **Is network performance important?** Mobile apps that use Android have networking capabilities, but Android is not necessary if networking is the primary focus. An alternative built specifically for network performance is [OpenWRT/LEDE](https://openwrt.org/about" \t "_blank).
* **Will there be a graphical user interface?** [Android’s](https://www.android.com/) pre-installed GUI might be a good option for systems that depend heavily on user interface.
* **Is the time to market short?** A more common Linux distribution will help reduce development time. [Ubuntu](https://ubuntu.com/embedded) is commonly used for faster launches.

# QEMU SETUP:

The first thing is to download the QEMU source code. Extract the tar ball and go to the extracted directory:

|  |
| --- |
| $ tar -zxvf qemu-0.14.0.tar.gz  $ cd qemu-0.14.0 |

Run the configuration script. We will build QEMU for i386. (It can be built for other architectures too, like ARM, PPC, SPARC, etc.) Let’s install the Ubuntu distro in the virtual machine — that’s the reason we’ve chosen to build QEMU for the i386 architecture:

|  |
| --- |
| $ ./configure –target-list=i386-softmmu |

Hopefully, you will not run into any trouble during the configure script run. If there’s any issue, it will probably be some missing library or header files, which you can look for, and install.

Once you are done with the configure script, compile the source code with the make command. After compilation, QEMU binaries should be installed in their proper locations. On my Fedora system, I used the su command to get the necessary root privileges and install the binaries using make install.

To confirm that QEMU has been successfully installed, run qemu, and a pop-up window appears it confirm the successful installation of QEMU.

# FIRST BOOT:

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

Install linaro toolchain from ubuntu package manager

Sudo apt install gcc-arm-linux-gnueabi

Sudo apt install gcc-arm-linux gnueabihf

We’ll go for soft float for now, due rootfs compatability.

Alternatively, download latest pre-built linaro toolchain from as per host architecture.

**First Boot**

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”

Emulate using Qemu – initrd approach

qemu-system-arm –M vexpress-a9 –m 1024 –serial stdio \

-kernel zImage –dtb vexpress-v2p-ca9.dtb \

-initrd rootfs.img –append “console=ttyAMA0 root=/dev/ram0 rw”

# Custom Kernel Build

**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

**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\_defconfig

**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+)

**Build the kernel**

**Customization – menuconfig**

Run menuconfig for further customization

Build kernel image

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

Build Device Tree Binaries

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

**Booting the new kernel**

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”

## Why cross-compile?

In theory, a PC user who wanted to build programs for some device could get the appropriate target hardware (or emulator), boot a Linux distro on that, and compile natively within that environment. While this is a valid approach (and possibly even a good idea when dealing with something like a Mac Mini), it has a few prominent downsides for things like a linksys router or iPod:

* **Speed** - Target platforms are usually much slower than hosts, by an order of magnitude or more. Most special-purpose embedded hardware is designed for low cost and low power consumption, not high performance. Modern emulators (like qemu) are actually faster than a lot of the real world hardware they emulate, by virtue of running on high-powered desktop hardware.[[3]](https://landley.net/writing/docs/cross-compiling.html#footnote3)
* **Capability** - Compiling is very resource-intensive. The target platform usually doesn't have gigabytes of memory and hundreds of gigabytes of disk space the way a desktop does; it may not even have the resources to build "hello world", let alone large and complicated packages.
* **Availability** - Bringing Linux up on a hardware platform it's never run on before requires a cross-compiler. Even on long-established platforms like Arm or Mips, finding an up-to-date full-featured prebuilt native environment for a given target can be hard. If the platform in question isn't normally used as a development workstation, there may not be a recent prebuilt distro readily available for it, and if there is it's probably out of date. If you have to build your own distro for the target before you can build on the target, you're back to cross-compiling anyway.
* **Flexibility** - A fully capable Linux distribution consists of hundreds of packages, but a cross-compile environment can depend on the host's existing distro from most things. Cross compiling focuses on building the target packages to be deployed, not spending time getting build-only prerequisites working on the target system.
* **Convenience** - The user interface of headless boxes tends to be a bit crampled. Diagnosing build breaks is frustrating enough as it is. Installing from CD onto a machine that hasn't got a CD-ROM drive is a pain. Rebooting back and forth between your test environment and your development environment gets old fast, and it's nice to be able to recover from accidentally lobotomizing your test system.

Cross-Compiling Code:

Simple Hello world program:

#include<stdio.h>

int main() {

printf("Hello World\n");

return 0;

}

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

1dd h1.out

1dd h2.out

# 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

//test.c

int main() {

int a,b,c,d;

a=10,b=20;

c=sum(a,b);

d=square(a);

//print c,d

}

//sum.c

int sum(int x, int y)

return x + y;

//sqr.c

int square(int x)

{

return x\*x;

}

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

# copy all.out to target rootfs and test

Static Library

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

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

arm-linux-gnueabi-gcc -L. test.o -1sample -o s1.out

arm-linux-gnueabi-gcc -L. test.o -1sample -o s2.out -static

#copy s1.out, s2.out to target roots and test

file s1.out $2.out

# compare size of s1.out, s2.out

1dd s1.out $2.out

# check size of s2.out before strip

arm-linux-gnueabi-strip s2.out

# check size of s2.out after strip

**Dynamic Linking**

# On Host

arm-linux-gnueabi-gcc -shared -o libsample.so sum.o sqr.o

arm-linux-gnueabi-gcc -L. test.o -1sample -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

1dd d1.out

versioning shared object files

# On Host

arm-linux-gnueabi-gcc -shared -W1, -soname,libsample.so \ -o libsample.so.1.0.1 sum.o sqr.o

ln -s libsample.so.1.0.1 libsample.so

arm-linux-gnueabi-gcc -L. test.o -1sample -o d1.out

#copy d1.out to target rootfs

#copy libsample.so.1.0.1 to /mnt/rootfs/opt/mylibs # add an entry to /opt/mylibs

#add an entry to /optmylibs in /mnt/rootfs/etc/ld.so.conf

# On Target

1s /opt/mylibs./d1.out # will give error initially

1dconfig

./d1.out

1s /opt/mylibs

# observer generated symlink

Working with U-BOOT

Obtaining Source

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

Let's call the cloned/extracted source as USRC

Alternative:-

Checkout U-Boot source from <https://gitlab.denx.de/u-boot/u-boot>

Switch to any stable branch like v2020.10

You may also download offline tarball from gitlab.denx.de

**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

Simple Boot - Rootfs in SD Card

qemu-img create simplesd.img 64M

sudo mikfs.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-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

Partitioning SD card

Preparing partitioned SD card

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 -1 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/boot

#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

Rootfs

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/mmcb1k0p2 rw rootfstype=ext4'

bootz 0x60200000 - 0x60100000

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

sudo apt install tftpd

# create /etc/xinetd.d/tftp

# with specified content

# replace server\_args as per your machine

/etc/init.d/xinetd restart

sudo modprobe tun

sudo ifconfig tape 192.168.0.1

**TFTP Boot**

**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 server ip 192.168.0.1

ping 192.168.0.1

tftp 0x60200000 zImage

tftp 0x60100000 vexpress-v2p-ca9.dtb

setenv bootargs 'console-ttyAMA0 root=/dev/mmcb1k0p2 rootfstype=ext4'

bootz 0x60200000 - 0x60100000