### **Embedded System:**

An embedded system is defined as a combination of computer hardware and software designed for a specific purpose.

* They are typically equipped with a processing unit.
* They can also be equipped with temporary memory and permanent storage.
* They are part of bigger systems.
* They are customized for a specific purpose.

In general, they are less costly compared to generic purpose system

**Key Characteristics:**

* Typically equipped with a processing unit.
* Can also have temporary memory and permanent storage.
* Part of bigger systems.
* Customized for a specific purpose.
* Generally, less costly compared to generic purpose systems.

Embedded systems became part of our day-to-day life.

Development Options for a Security System (Illustrative Example):

1. Option One: Generic Purpose Computer:

Approach: Use a generic computer, attach a camera, develop an OpenCV application to detect intruders and send Wi-Fi commands.

Pro’s:

* Development time is significantly lower due to fewer limitations compared to embedded systems.

Con’s:

* Cannot be presented as a product to a customer.
* \* Has many unused features beyond intruder detection.
* **Prone to security risks**.
* Takes up a large space.
* **Boots very slow** due to running a full operating system.
* **Significantly expensive**

**Conclusion:** This approach is not feasible for a product.

2. Option Two: Raspberry Pi

Approach: Use a Raspberry Pi

Pros:

* **Significantly reduced cost**
* Can be put into an enclosure box.

Con’s:

* Still **not entirely optimized**.
* Has unused hardware elements like Ethernet controller/port, USB controller/ports, HDMI controller/port
* Conclusion: While better, further cost reduction and compactness are possible.

3. Option Three: Custom Embedded System

Approach: Hardware designers create a **new, custom PCB based on the Raspberry Pi 5's design**, removing unnecessary ports (HDMI, Ethernet, USB) and embedding necessary components like a Wi-Fi chip. Embedded software developers then write the software.

Pros:

* **Significantly reduced cost**
* Option to embed camera modules directly into the PCB.
* **Achieves maximum price optimization** when producing millions of units
* Results in a product with **specific features and removal of generic hardware/software**.

**Cons:**

* **Increased overall development cost and time** because creating a new product is complex.
* Software development can take more time due to limitations with ARM processors commonly used in embedded systems.

Overall Impact: Embedded systems simplify lives and are found everywhere. They are "cleverly designed" to serve a specific purpose, using either complex operating systems (Linux, Android) or simpler microcontrollers.

Introduction to Embedded Linux Systems:

* Generic Embedded Systems: Used to develop custom products, often based on reference platforms from SOC and chip manufacturers.
* Hardware Perspective: Existing designs are taken, and unnecessary components are removed, while necessary components are added.
* Software Perspective: Embedded software developers take provided BSP (Board Support Package) and OS software to tailor it: removing unnecessary drivers, applications, and libraries, and adding custom drivers, applications, libraries, and features.
* Kernel Optimization/Customization: Performed to create a custom embedded product software.
* Goal of Embedded Linux: To include only necessary features to ensure fast boot times and a seamless experience for the end user.

**Why Use Linux in Embedded Products?**

Linux is used where operating system functionality is heavily needed and where the open-source ecosystem can be leveraged.

* Complexity: Modern embedded products (like televisions) are too complex for bare-metal development due to extensive functionality.
* Open-Source Benefits: Linux and its open-source community already offer solutions for these complex features
* Hardware Dependency: The hardware platform must also support these features.

Linux Software Stack Components:

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

A Linux operating system's software stack can be virtually represented in layers:

1.Applications (Top Layer):

## **Executable Binaries:**

Binaries are executables created for specific purposes.

**Examples:**

* Desktop manager applications (for windowing systems)
* Command-line utilities (e.g., cp from BusyBox)
* Python interpreter
* OpenSSL

**Common directories:**

* /sbin
* /bin
* /usr/bin

### **BusyBox**

BusyBox is a crucial utility in embedded Linux systems. It provides many standard UNIX tools in a single binary and is often one of the first things compiled for user space.

## **Services**

Services are processes launched by the kernel or init, responsible for specific system tasks.

### **Init Service**

* The **first user-space service** launched by the kernel.
* Responsible for starting all other services.

**Init System Variants:**

* systemd or upstart: Used in complex/desktop Linux systems
* System V init: Common in embedded Linux

### **Examples of Services:**

* **UDEV**: Device discovery and hotplug support
* **Bluetooth Service**
* **Network Service**
* **Compositor**: Connects graphical apps to lower layers
* **SSHD**: Secure Shell Daemon
* **BootlogD**: Logs boot messages

## **Libraries:**

Libraries are shared components used by applications and services.

### **Core: C Library**

* Acts as the foundation for other libraries.
* Interfaces with the kernel via the **system call interface**.
* **Choosing the right C library is critical for embedded systems.**

**Common C Library Variants:**

* glibc: Most common
* musl: Lightweight and fast
* uClibc: For embedded systems
* Bionic: Used in Android

### **Other Important Libraries:**

* QT: Graphics framework
* Boost: C++ library
* OpenSSL: Cryptographic operations
* POSIX libraries
* Pthread: Threading
* RT library (Real-time)
* M library (Math)
* XCB: X11 support
* EGL: Embedded graphics

## **System Call Interface**

Acts as the bridge between **user space** (apps, services, libraries) and **kernel space** (drivers, kernel).

* Implemented through functions in the C library.
* Used to access kernel services and hardware.

**Examples of system calls:**

* open
* close
* read
* write
* poll
* ioctl

## **Communication Mechanism**

When a user-space process performs a system call on a driver file:

* The **kernel driver** is notified.
* Relevant **callbacks or hooks** in the driver are executed.

## **Drivers**

Kernel-level software enabling the OS to interact with hardware.

* Represented as special files in /dev.
* Handle requests via system call hooks.

**Examples:**

* USB device drivers
* PCIe device drivers
* I²C device drivers
* Wi-Fi module drivers
* Touchscreen drivers
* Button drivers
* FM tuner drivers

## **Linux Kernel (Bottom Layer)**

The **core of the Linux OS**, managing hardware and system operations.

### **Core Components:**

* **MMU (Memory Management Unit)**: Handles virtual memory
* **Process Management Subsystem**: Schedules processes
* **IPC Subsystem**: Provides interprocess communication (queues, pipes, sockets, shared memory)
* **Virtual File System (VFS)**: Interfaces with user-space filesystems
* **Network Stack**: Enables network communication through sockets

### **Linux Kernel API:**

Used by kernel subsystems and drivers for:

* Memory allocation
* Resource management
* Device interaction

## **Additional Crucial Components:**

## **Bootloader**

The **bootloader** is the **primary software** responsible for initializing hardware and loading the **Linux kernel** into memory during the boot process.

## **Toolchain**

A **toolchain** is a set of tools used to build software, particularly for **cross-compilation** in embedded systems.

**Cross-Compilation**

* **Definition**: Building software on one platform (e.g., a PC with x86 architecture) that runs on another platform (e.g., Raspberry Pi with ARM).
* **Purpose**: Most embedded devices lack the processing power and resources for native compilation.
* **Result**: Faster, more efficient builds from a development host system.

**Common Toolchain Components**

* **GCC / G++**: GNU C and C++ Compilers
* **Binutils**: Tools for manipulating binary files (ld, as, objdump, etc.)
* **CMake / Make**: Build system configuration tools
* **Sysroot**: A directory representing the target system’s root filesystem used for linking

## **Customizing Embedded Linux Distributions**

Creating an embedded Linux distribution involves **adding** and **omitting** components across multiple software layers to match the specific requirements of the product.

### **Additions (for Product Functionality)**

To enable the desired features, the following components are typically added:

* **Applications**:
  + Custom infotainment systems
  + Multiple user-facing or background applications working together
* **Services**:
  + Display service
  + Communication service
* **Libraries**:
  + Virtual keyboard libraries
  + Media libraries
  + Communication protocol libraries
* **Drivers**:
  + Touchscreen drivers
  + Button input drivers
  + FM tuner drivers
* **Kernel Modifications**:
  + Custom kernel configurations to enable specific hardware or performance features
  + Splash screen and custom boot animations
  + Sometimes customizations to the bootloader as well

### **Omissions (for Fast Boot, Security, and Reliability)**

To optimize for performance and security, the following components are commonly removed or reduced:

* **Applications**:
  + Desktop environments or window managers
  + Unused utilities and demo apps
* **Services**:
  + Generic init systems (e.g., systemd)
  + Replaced with minimal custom init scripts
  + Minimal or no compositor (to reduce graphical complexity)
* **Libraries**:
  + Remove unused or redundant libraries to reduce attack surface
  + Improve robustness and reduce image size
* **Drivers**:
  + Disable or remove unused hardware drivers
    - Example: Wi-Fi drivers if the device has no wireless capability
    - Example: Physical keyboard drivers if only a virtual keyboard is used
* **Kernel Modifications**:
  + Remove debugging features
  + Optimize for size and reliability
  + Strip down unnecessary modules and components to achieve faster boot time

**YOCTO PROJECT:**

### **Role of Yocto Project:**

The **Yocto Project** is a powerful **build system** used in embedded Linux development. It provides:

* A **modular and flexible framework** for building custom Linux distributions.
* Access to **all software layers** — from the bootloader to applications — enabling fine-grained control.
* Support for cross-compilation, package management, and reproducible builds.

This makes Yocto especially valuable when **tailoring embedded Linux systems** for specific hardware or product requirements.