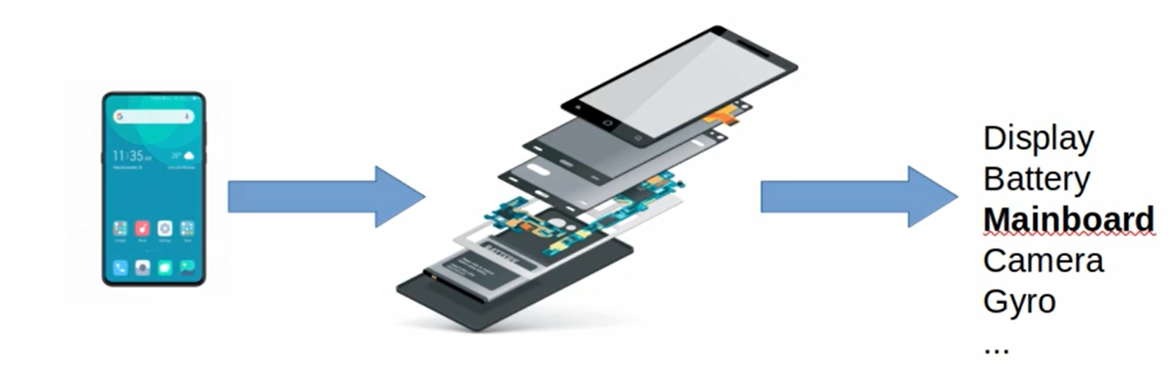
### **Embedded System:**

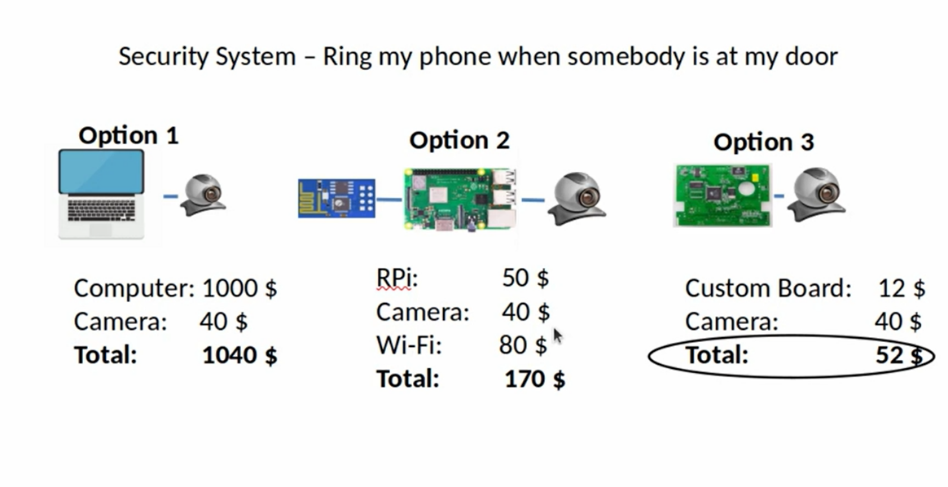
An embedded system is defined as a combination of hardware and software designed to perform a specific task

* 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.
* less cost as compared to generic purpose system



* In the above example we can call individual components as a embedded systems because each components has its own purpose and memory and processor.

**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:**

* We Cannot be present this as a product to a customer.
* It has many unused features beyond intruder detection.
* Prone to security risks.
* Overall system will take up more space.
* System boots very slow since it is running a full operating system
* Significantly expensive

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

**Option 2:** Raspberry Pi

**Approach:** Use a Raspberry Pi

**Pros:**

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

**Con’s:**

* Still not entirely optimized.
* It has unused hardware elements like Ethernet controller/port, USB controller/ports, HDMI controller/port.we don’t nees all these stuff to detect intruder and send the wifi commands.

**Conclusion:** While better, further cost reduction and compactness are possible.

**Option 3**: 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 that can achieve the same functionality as it is in option 1.

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.

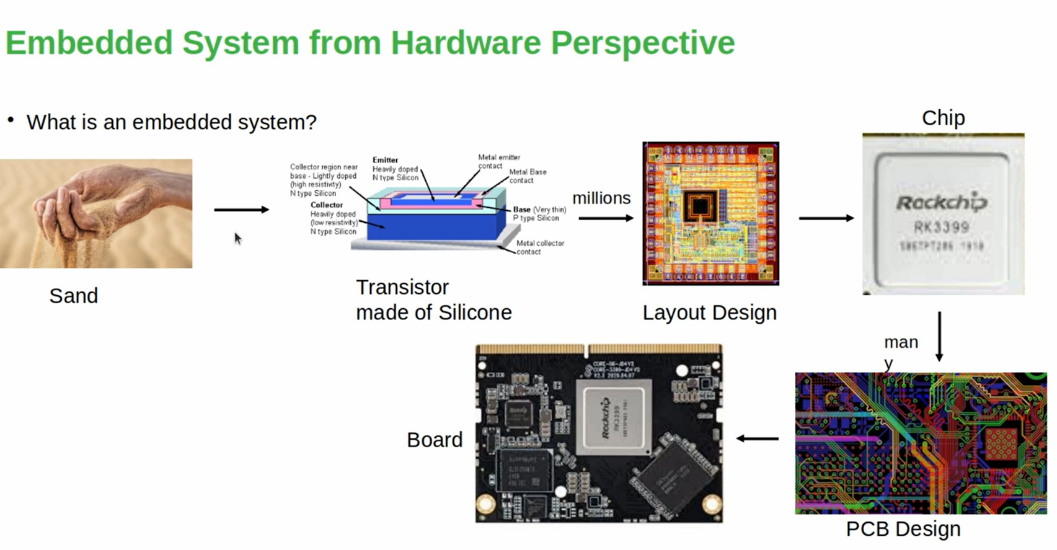
**Cons:**

* 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.

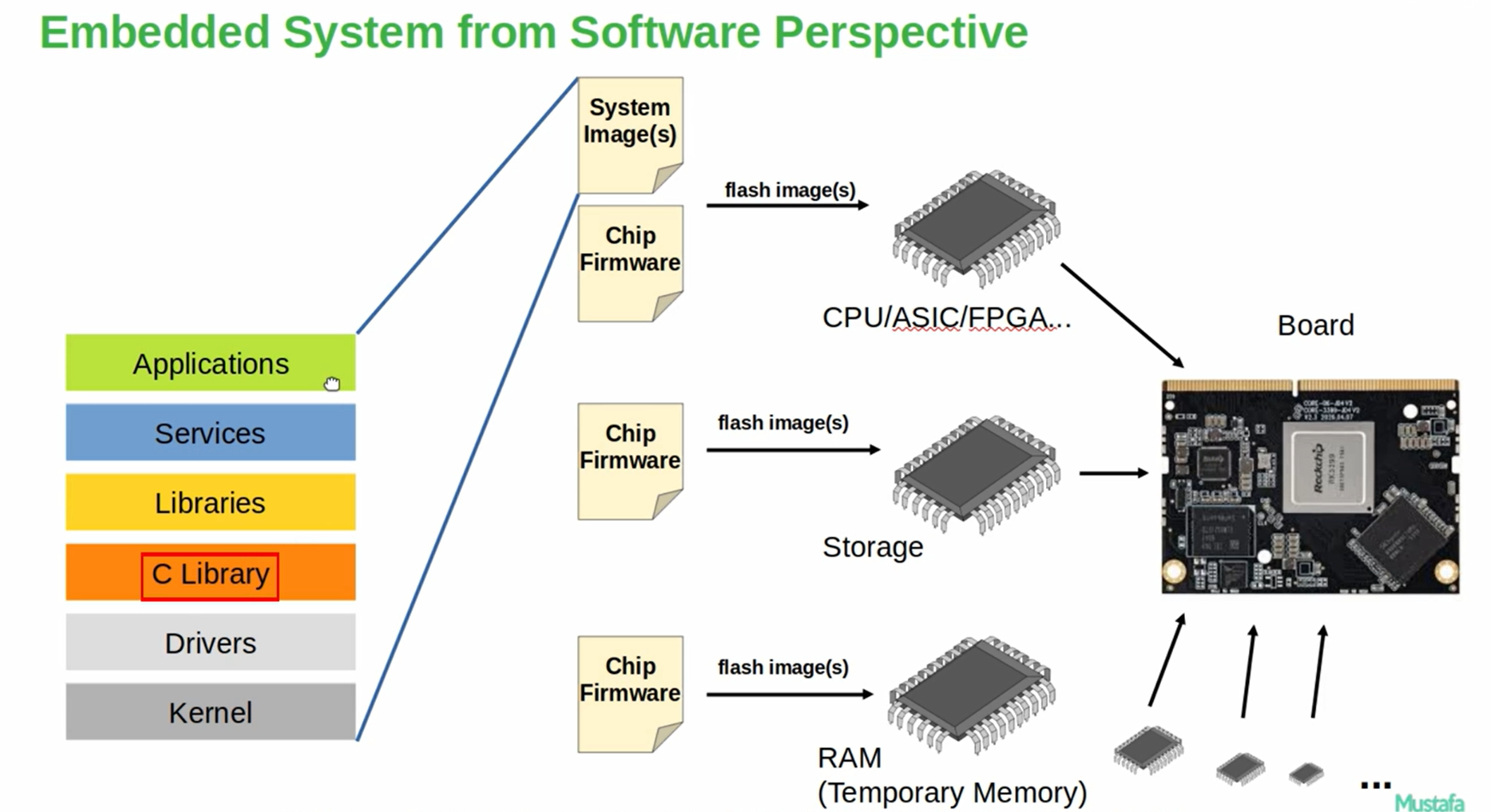
**Overall Impact**: Embedded systems simplify lives and are found everywhere. They are "cleverly designed" to serve a specific purpose, using either complex operating systems (Linux, Android) or simpler microcontrollers.

**Embedded systems from hardware perspective**:

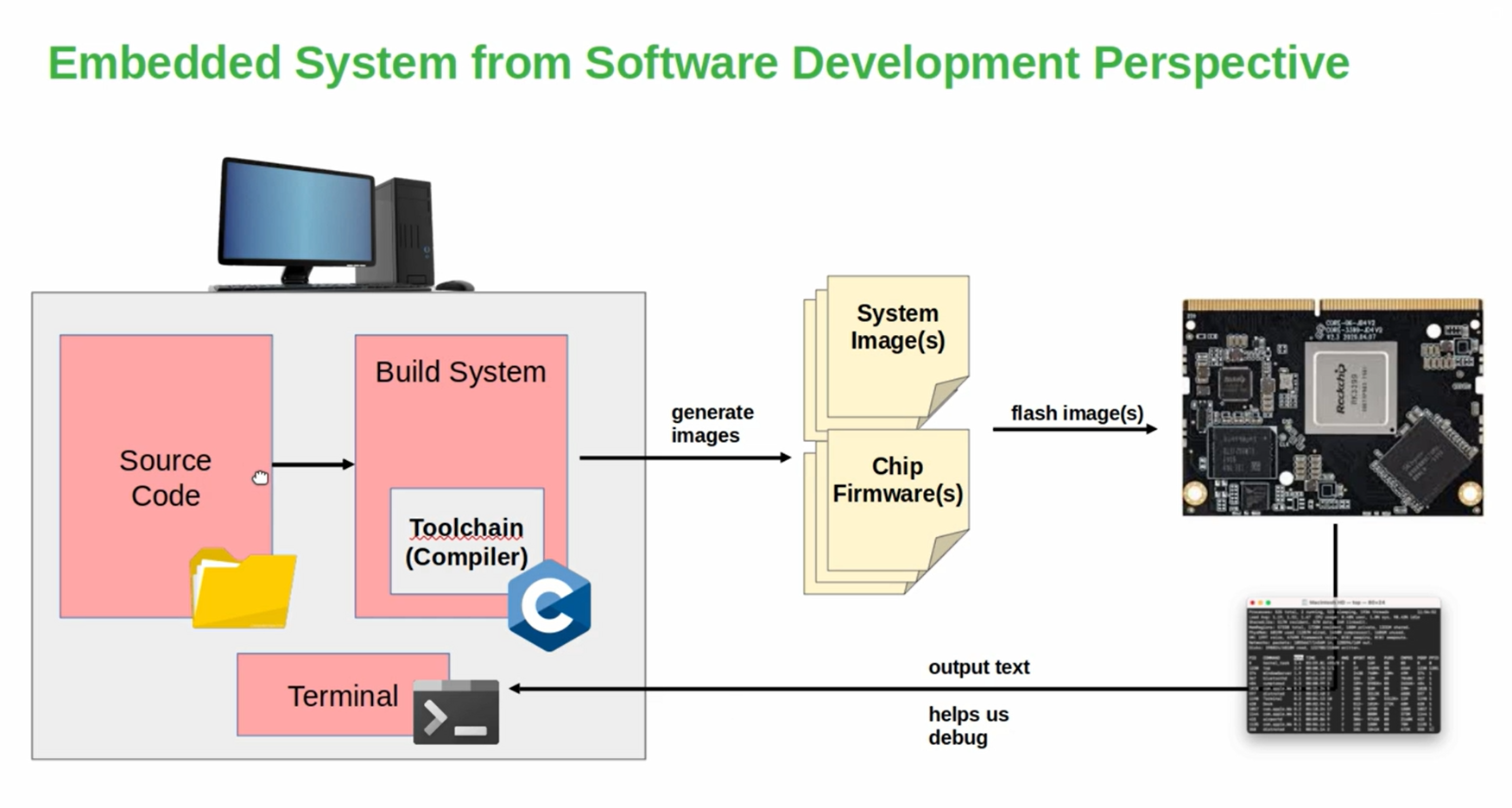
* **Hardware Perspective:** Existing designs are taken, and unnecessary components are removed, while necessary components are added.



**Embedded systems from sotware perspective**:



**Generation of system Image and flashing it into board:**



🡪First we need source code for all the components that makeup our OS. We can get it from either SOC vendor or can download from web

🡪compile source code using Toolchain(compiler) to generate binaries,libraries and object files. ARM processor uses gcc compiler

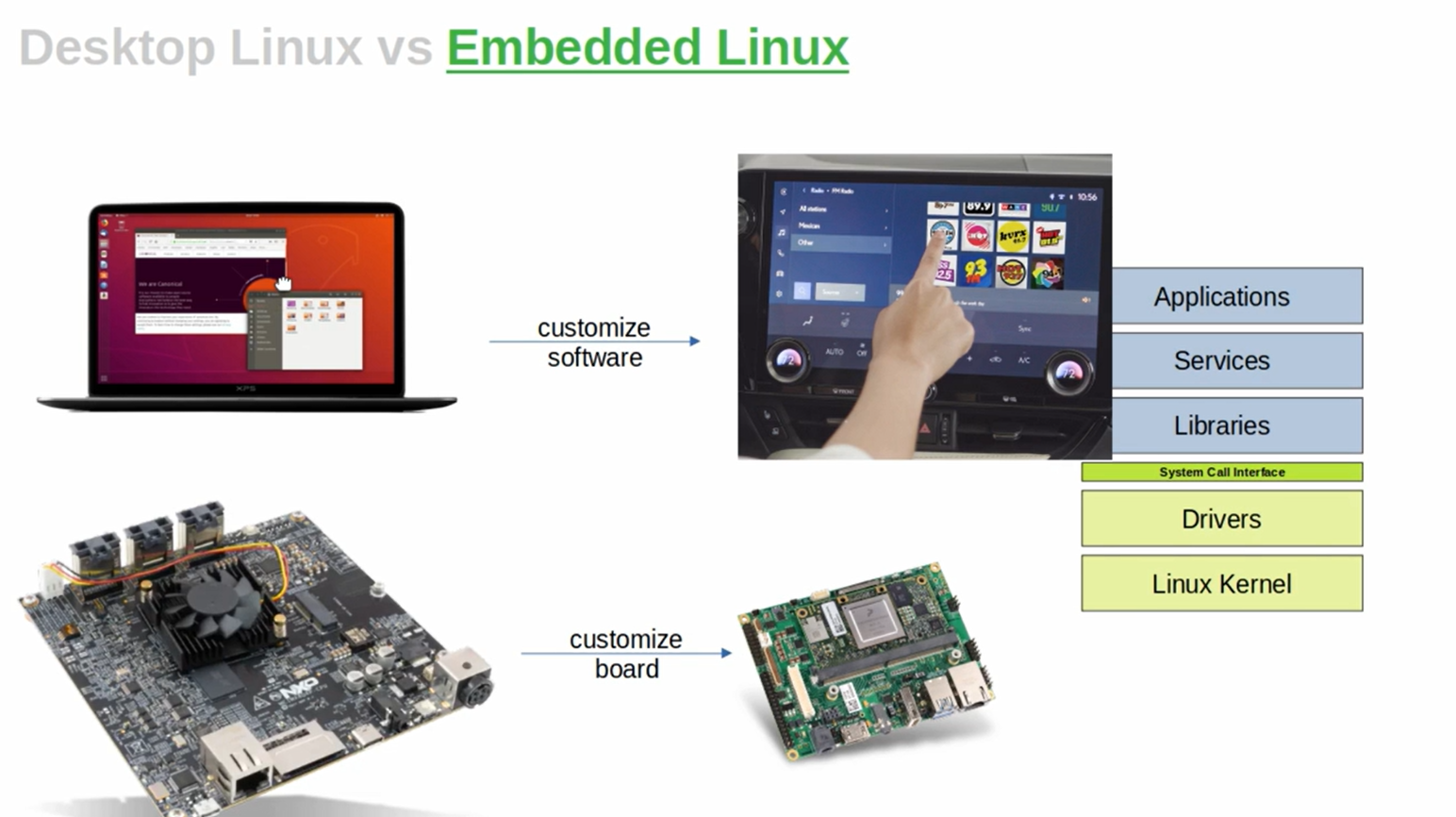
🡪A build system is a set of tools, scripts, and rules used to automate the process of converting source code into a working program or software image. Build system and tool chain will also be provided by soc vendors.

🡪Finally the system Images will be generated. We can flash them in our embedded board and test/debug it with using human interface terminal.

Here system Images contains complete OS:

* **Bootloader** (e.g., U-Boot)
* **Linux Kernel Image** (zImage or uImage)
* **Device Tree Blob** (.dtb)
* **Root Filesystem** (rootfs.ext4, .cpio, .squashfs, etc.)

**Introduction to Embedded Linux Systems:**

****

**Hardware vs Software Perspective:**

**Hardware Perspective:**

* Take an existing board design (e.g., Raspberry Pi).
* Remove unused ports/hardware like HDMI, USB, Ethernet.
* Add only essential components like Wi-Fi, Camera, etc.

**Software Perspective:**

* Start with a BSP (Board Support Package) and Linux source code.
* Remove unwanted drivers, applications, libraries.
* Add only what’s needed (e.g., custom drivers, apps).
* Goal: Fast boot, low power, smaller image size.

| **Feature** | **Linux (General)** | **Embedded Linux** |
| --- | --- | --- |
| Usage | PCs, servers | Embedded devices (IoT, TV) |
| Size | Large (GBs) | Small (MBs or less) |
| User Interface | GUI supported | Mostly no GUI |
| Customization | General purpose | Highly customized |
| Performance needs | High | Optimized for low power |

**Why Use Linux in Embedded Products?**

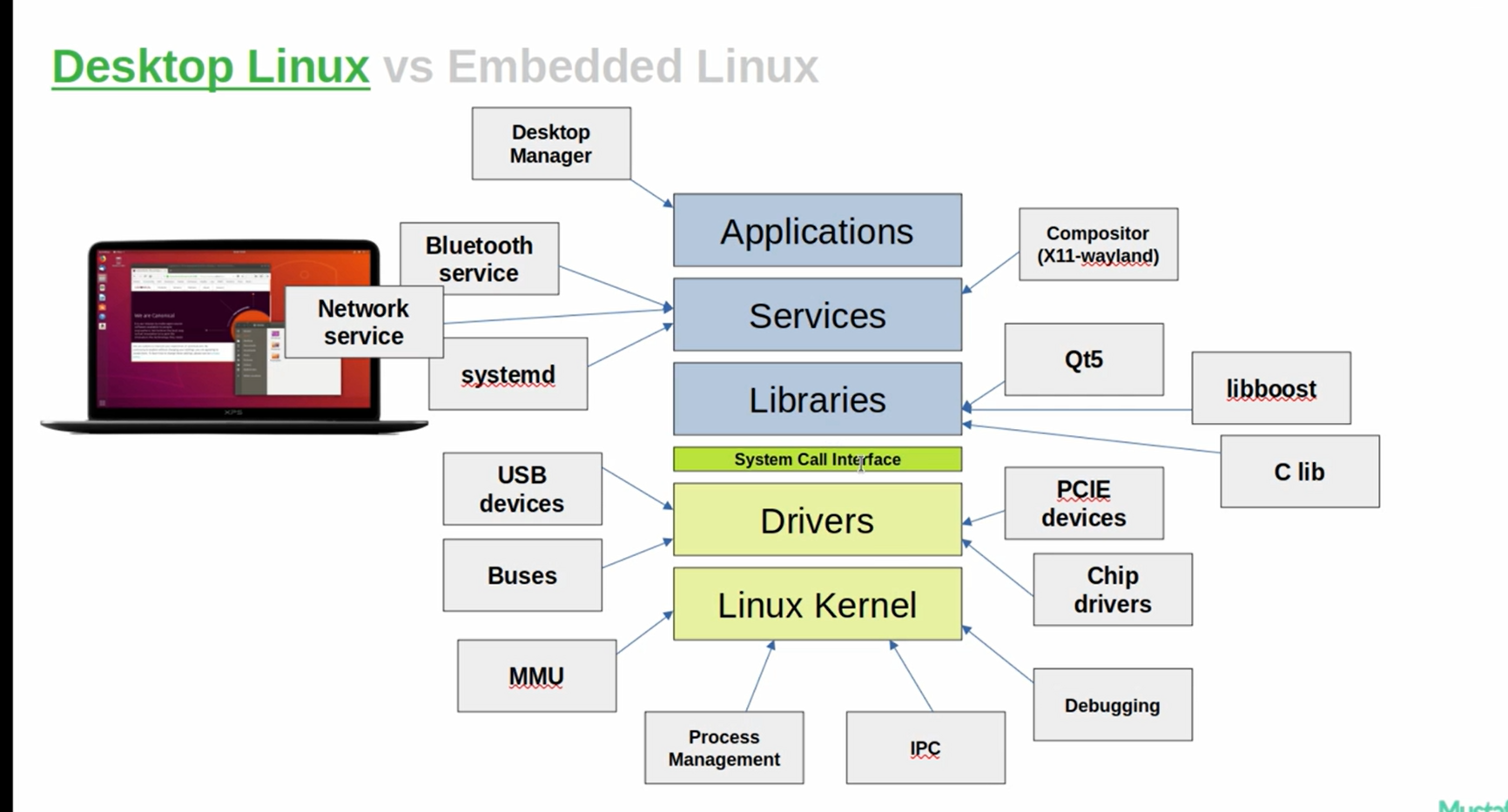
Linux is used where operating system functionality is heavily needed and where the open-source ecosystem can be leveraged.

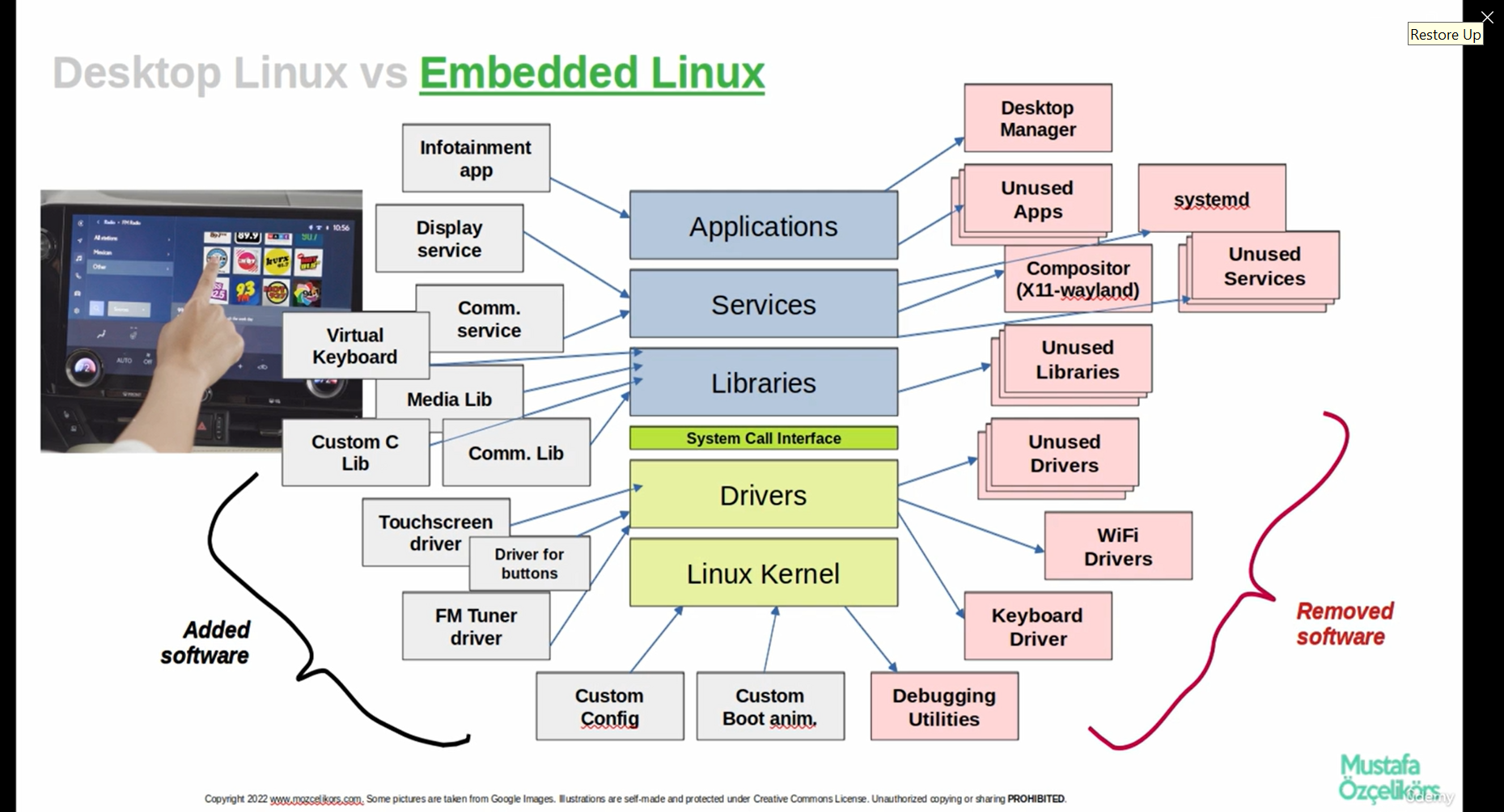
* Complexity: Modern embedded products (like televisions) are too complex for bare-metal development due to extensive functionality.
* Open-Source Benefits: Linux and its open-source community already offer solutions for these complex features
* Hardware Dependency: The hardware platform must also support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

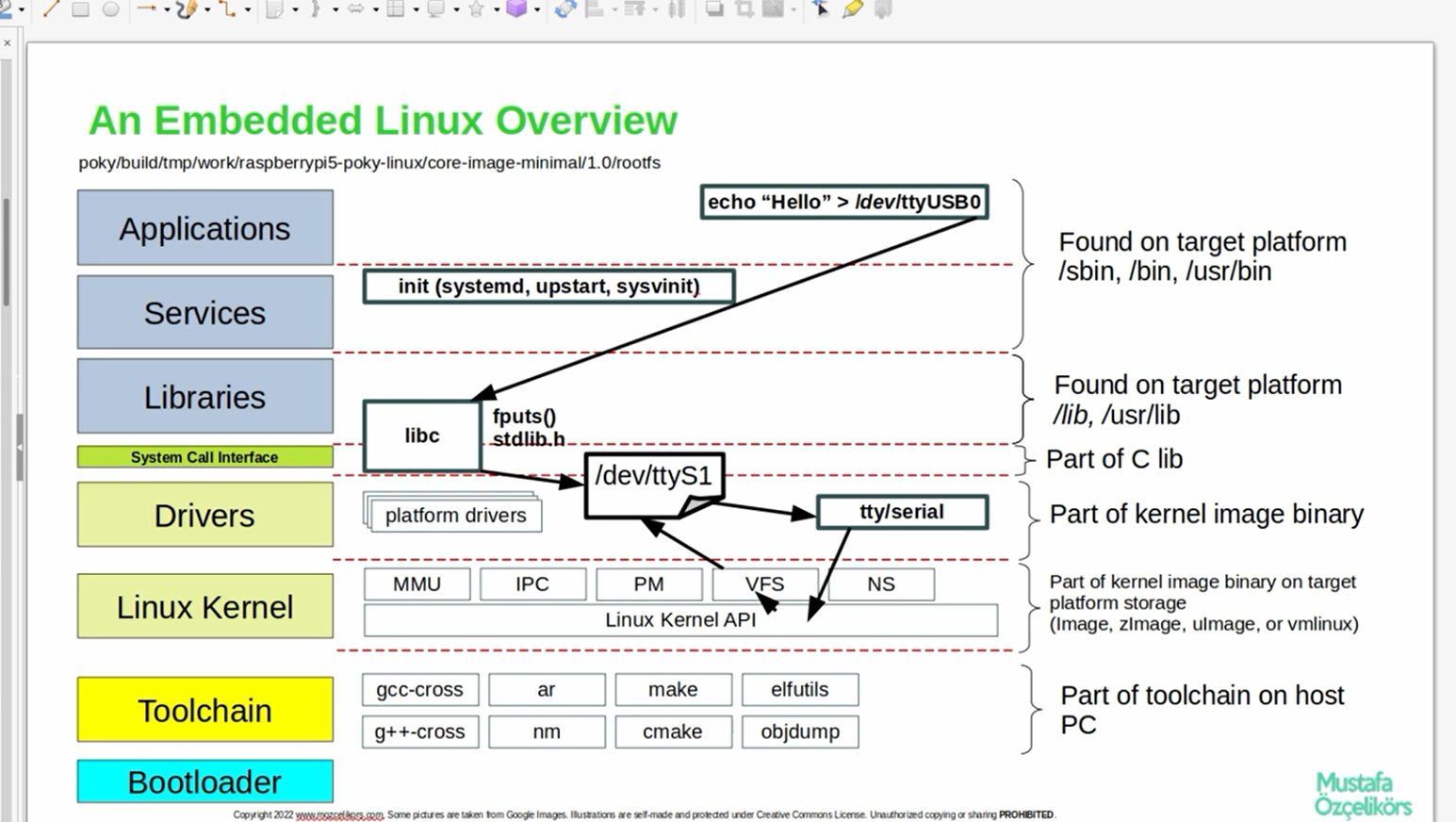
Linux Desktop Software Stack Components:

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

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





1.Applications (Top Layer):

## **Application 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 **single executable** that combines many common **Linux/Unix command-line tools** into one lightweight program

Eg:ls,cp,cat,mv,vi etc..

## **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