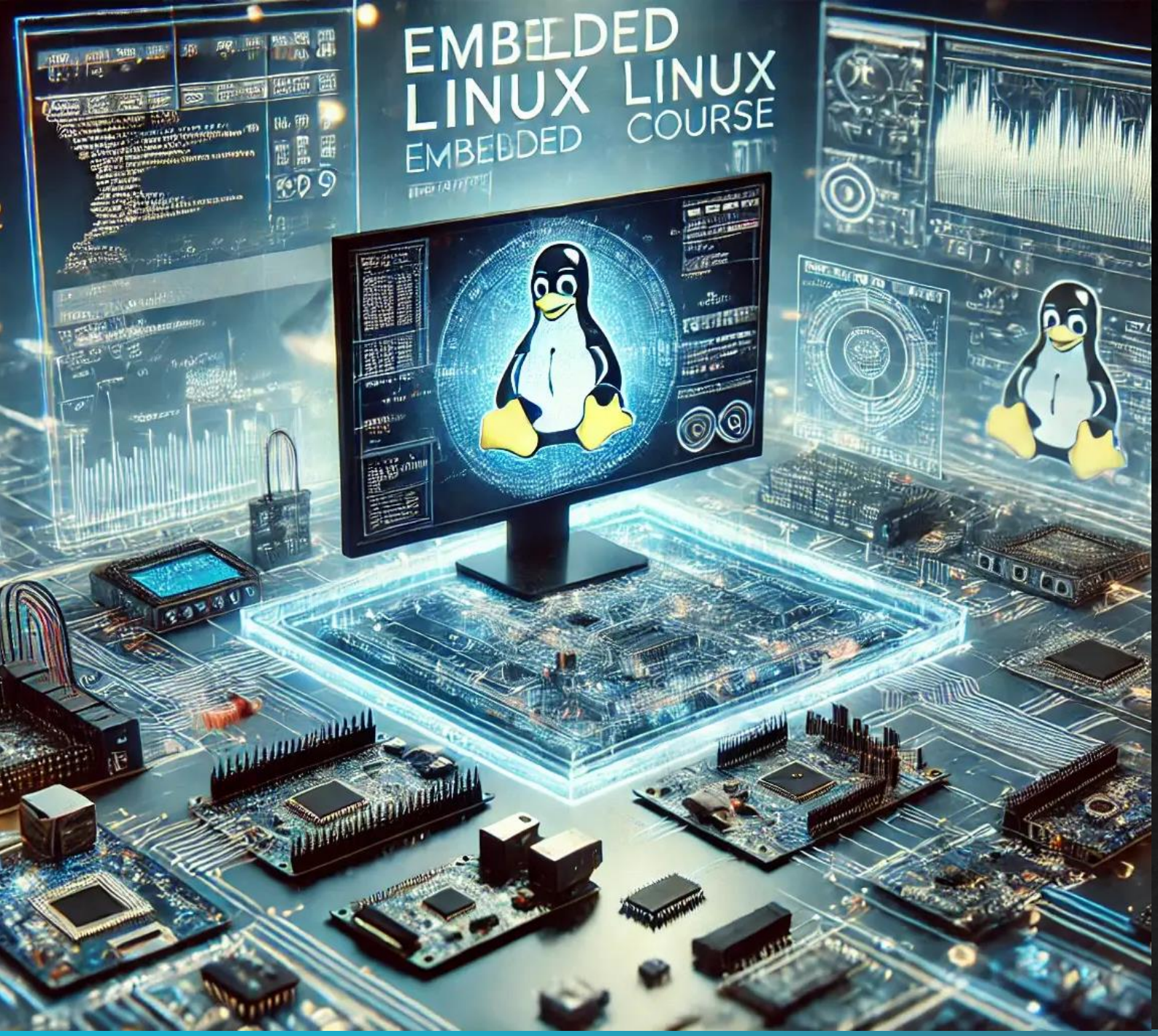


Embedded Linux

Embedded Linux with Buildroot

Course Outline

Linux Fundamentals

Cross-Compilation and Toolchains

Buildroot Fundamentals and Filesystem Generation

Bootloaders and Kernel

Device Tree and Drivers

Application Development and Debugging

Linux Fundamentals

✅ **Learning Objectives:**

- Understand the basic structure and philosophy of Linux
- Use essential Linux commands and shell features
- Navigate the Linux filesystem
- Write and execute simple shell scripts
- Manage file permissions and processes

📘 **Topics:**

- Linux history and open-source principles
- Filesystem hierarchy (/bin, /etc, /usr, etc.)
- Basic commands: ls, cd, cat, grep, find, chmod, etc.
- Shell scripting basics (bash)
- File permissions and users
- Process management (ps, top, kill)
- Text editors (nano, vim)

Cross-Compilation and Toolchains

✅ **Learning Objectives:**

- Understand the concept of cross-compilation
- Configure and use toolchains for ARM-based targets
- Compile user applications for embedded systems
- Set up the environment for cross-development

📘 **Topics:**

- Native vs. cross-compilation
- What is a toolchain? (gcc, binutils, libc)
- Types of toolchains (glibc, musl, uClibc)
- Setting up cross toolchains (Buildroot or external)
- Writing and compiling a sample C application for target
- Deploying applications to target hardware or emulator

Buildroot and Filesystem Generation

✅ **Learning Objectives:**

- Use Buildroot to generate a root filesystem, kernel, and bootloader
- Customize the root filesystem
- Add applications and libraries to the image
- Understand BusyBox and its role in embedded systems

📘 **Topics:**

- What is Buildroot? Architecture and workflow
- Buildroot menuconfig (make menuconfig)
- Filesystem types: ext4, initramfs, squashfs
- BusyBox: lightweight tools for embedded
- Rootfs customization
- Adding custom packages or applications
- Booting with QEMU or real hardware

Bootloaders and Kernel

✅ **Learning Objectives:**

- Understand the boot process in embedded Linux
- Configure and build U-Boot bootloader
- Configure and build the Linux kernel
- Load kernel and rootfs on target hardware

📘 **Topics:**

- Boot stages: ROM → SPL → U-Boot → Kernel → Rootfs
- U-Boot configuration and scripting
- Kernel source tree structure
- Device driver modules and kernel configuration (make menuconfig)
- DTS (Device Tree Source) basics
- Booting kernel via SD, TFTP, NFS

Device Tree and Drivers

✅ **Learning Objectives:**

- Understand the role of the Device Tree in embedded Linux
- Modify Device Tree sources for custom hardware
- Load and use drivers in Linux
- Identify and enable device drivers using kernel config

📘 **Topics:**

- Device Tree structure and syntax (.dts, .dtsi)
- Memory mapping and peripheral definitions
- Writing or modifying a basic Device Tree overlay
- Kernel modules vs built-in drivers
- Loading modules: modprobe, insmod, lsmod
- Debugging device drivers

Application Development and Debugging

✅ **Learning Objectives:**

- Write and deploy embedded applications on Linux
- Use debugging and profiling tools
- Connect host-target for remote debugging
- Monitor system performance and behavior

📘 **Topics:**

- Writing C applications with POSIX APIs
- Using gdb, strace, ltrace, perf
- Remote debugging with gdbserver
- Logging and system diagnostics (dmesg, syslog)
- Accessing GPIO, UART, SPI, I2C from user space
- System startup scripts and services

We will cover these skills

- Understand the basic structure and philosophy of Linux
- Learn to work efficiently with the Linux command line
- Explore the Linux filesystem hierarchy and device model
- Write simple Bash scripts for task automation
- Manage processes and access system resources in a Linux environment
- Build foundational knowledge required for embedded Linux development

Chapter 1 Linux Fundamentals

Course Introduction

- Overview of the training objectives
- Tools and platforms used (Buildroot, STM32MP1, Raspberry Pi)
- Target audience: developers preparing for embedded Linux careers

Development Environment Requirements for Embedded Linux Training

- Host System Requirements

Requirement	Description
Host OS	Linux-based (recommended: Ubuntu 22.04 LTS or Debian-based distro)
Alternative OS	macOS or Windows (with WSL2 or VirtualBox + Ubuntu VM)
CPU	x86_64, 2 cores or more (for faster Buildroot builds)
RAM	Minimum 4 GB (8 GB recommended)
Disk Space	At least 20 GB free (Buildroot outputs and sources are large)
Internet	Required for downloading Buildroot, toolchains, kernel sources, etc.

Development Environment Requirements for Embedded Linux Training

- Required Software Packages (on Host Linux)

```bash
sudo apt update
sudo apt install -y \
    build-essential git wget curl unzip \
    gcc g++ make cmake libncurses5-dev \
    gawk flex bison \
    libssl-dev libelf-dev \
    qemu-system-arm
```

Development Environment Requirements for Embedded Linux Training

- Setting Up a Host Linux System (If the Student Is on Windows/macOS)

A **Option 1: Install Ubuntu via VirtualBox**

- Download VirtualBox
- Download Ubuntu ISO (22.04 LTS): https://ubuntu.com/download/desktop
- Create a VM with:
- 2+ cores
- 4 GB+ RAM
- 30 GB+ disk space
- Install Ubuntu normally inside VirtualBox

B **Option 2: Use WSL2 (Windows 10/11 Only)**

O **Option 3: Use a Native Linux Partition**

Development Environment Requirements for Embedded Linux Training

- Optional Tools for Better Development Workflow

Tool	Purpose
VSCode	Text editor with terminal integration
minicom / screen	Serial console over UART or QEMU
gdb	Debugging applications
Wireshark	Network debugging (for later modules)
Python3 + PySerial	For creating PC tools or testing serial

What is Linux?

- Open-source Unix-like operating system
- Three major components:
  - Kernel
  - System libraries
  - User space
- Philosophy: "Everything is a file”

Linux is not just an operating system — it's a philosophy.
Its open-source nature allows complete control and customization, making it perfect for embedded development. The kernel interacts with hardware, while libraries and user-space tools provide functionality and interface.

What is a Linux Distribution?

- Definition: A **Linux distribution (or distro)** is a complete operating system built around the **Linux kernel**, bundled with essential system software, utilities, libraries, and applications needed to operate a computer or embedded system.
- Key Components of a Linux Distribution
  - **Linux Kernel** – The core of the OS, managing hardware and system calls.
  - **GNU Utilities** – Basic tools like bash, ls, cp, gcc, make, etc.
  - **Package Manager** – Used to install, update, or remove software packages (e.g., apt, yum, pacman).
  - **System Libraries** – Libraries such as glibc or musl that provide low-level functions.
  - **User Applications** – Text editors, networking tools, graphical environments, etc.

How Are Distributions Different?

- Different distributions have different goals, structures, and software packaging systems. Some are designed for desktop users, others for servers, and some are tailored for **embedded systems**.

Distribution	Use Case	Package Manager	Description
Ubuntu	Desktop / Server	apt (Debian-based)	User-friendly, large community, LTS versions
Debian	Server / Base OS	apt	Very stable, used as a base for many other distros
Fedora	Developer Focused	dnf	Cutting-edge software, upstream-first philosophy
Arch Linux	Custom / Advanced	pacman	Minimalist, rolling release, DIY configuration
Buildroot	Embedded Systems	No package manager (uses make)	Simple, fast, build-from-source for embedded systems
Yocto	Industrial Embedded	BitBake	Flexible, modular, for complex embedded products

Why Use Special Distros for Embedded Linux?

- In embedded Linux development (like for STM32MP1 or Raspberry Pi):
- **Buildroot** is a great choice because:
  - Lightweight and easy to configure
  - Generates a minimal root filesystem
  - Ideal for quick prototyping or low-resource systems
- **Yocto Project** is better suited for:
  - Industrial and commercial products
  - Advanced customization and scalability
  - But has a steeper learning curve

Linux Filesystem Hierarchy

- Single-rooted tree: starts at /
- Key directories:
  - /bin, /sbin – essential binaries
  - /etc – configuration files
  - /dev – device files
  - /proc, /sys – virtual filesystems
  - /home, /var, /tmp, /usr

In Linux, everything — from hardware to processes — is treated as a file.
Understanding directories like /proc and /sys is essential, especially in embedded systems where hardware monitoring is done through these virtual files.

Linux Command Line Basics

- Common commands: ls, cd, mkdir, cp, rm, man
- Pipes |, Redirection >, >>, <
- Command chaining and history

The command line is your best friend in Linux.
Mastering a few basic commands allows you to navigate the system, manipulate files, and monitor devices.
Piping and redirection are powerful tools for chaining and logging outputs.

Introduction to Bash Scripting

- What is a shell script?
- Key elements:
  - #!/bin/bash
  - Variables, conditions (if), loops (for, while)
- Executing a script: chmod +x, ./script.sh

Shell scripting allows you to automate tasks such as setting up devices, managing services, or collecting logs.
It’s especially useful in embedded Linux, where automation often replaces GUI interaction.

Introduction to Bash Scripting

```
#!/bin/bash

echo "========== System Information =========="
echo "Date & Time: $(date)"
echo "---------------------------------------"
echo "CPU Info:"
grep "model name" /proc/cpuinfo | uniq
echo "---------------------------------------"
echo "Free Memory:"
free -h | grep Mem
echo "---------------------------------------"
echo "Disk Usage:"
df -h / | grep /
echo "---------------------------------------"
echo "IP Address:"
hostname -I
echo "========================================"
```

Device and Process Management

- Device access via /dev, /sys
- Managing processes:
  - ps, top, kill
- Mounting file systems: mount, umount, /etc/fstab

Embedded devices often have limited UI, so managing hardware and software through the terminal is critical.
Files like /dev/ttyS0 or /dev/i2c-1 represent real hardware.
Understanding how to mount USB or SD cards is vital in most projects.

Practice Tasks

- **In-Class Practice:**
- Use cat /proc/cpuinfo to read CPU details
- Write a Bash script that:
  - Shows free memory (free -m)
  - Prints CPU model
  - Logs output to a file
- **Mini Project:**
- Create a simple CLI menu script:
  - Option 1: Show IP address
  - Option 2: Display disk usage
  - Option 3: Exit

Skills Acquired in This Section

- Navigate and use Linux command line tools
- Understand Linux directory structure and virtual file systems
- Write basic Bash scripts for automation
- Monitor and manage processes
- Access and manipulate device files

Linux Filesystem Hierarchy

Directory	Purpose
/bin	Essential user binaries (e.g., ls, cp, rm)
/sbin	Essential system binaries (e.g., fsck, init)
/etc	System configuration files
/dev	Device files (e.g., /dev/sda, /dev/tty0)
/proc	Virtual filesystem for process and kernel info
/sys	Exposes kernel and hardware info
/usr	User utilities and applications
/var	Variable data (logs, mail, spool)
/tmp	Temporary files
/home	Home directories for users
/root	Home directory of the root user
/boot	Files needed to boot (e.g., kernel, grub config)
/lib	Essential shared libraries
/mnt and /media	Mount points for temporary devices (e.g., USB drives)
/opt	Optional software packages

Linux Filesystem Hierarchy

- **/ — Root Directory**
  - This is the **starting point of the entire filesystem tree**. All files and directories in Linux are located under /, regardless of the device or partition.
- **/bin — Essential User Binaries**
  - Contains essential user commands that are required **for booting and single-user mode**. These binaries are needed by both the system and regular users.
  - **Examples:**
    - ls – list directory contents
    - cp – copy files
    - mv – move/rename
    - cat – print file content
    - rm – remove files

Linux Filesystem Hierarchy

- **/sbin — System Binaries**
  - Contains **system administration binaries**, typically used by the root user. These tools manage disks, filesystems, network, etc.
  - **Examples:**
    - fsck – filesystem check
    - ifconfig, ip – network configuration
    - mount, umount – mount/unmount filesystems
    - reboot, shutdown – system control
- **/etc — System Configuration**
  - Stores **system-wide configuration files**. Most files here are plain text.
  - **Examples:**
    - /etc/passwd – user account information
    - /etc/fstab – filesystem mount table
    - /etc/hostname – system hostname
    - /etc/network/interfaces – network settings

Linux Filesystem Hierarchy

- **/home — User Home Directories**
  - Contains personal directories for all **regular users**.
  - **Examples:**
    - /home/john
    - /home/alice
- **/root — Root User's Home**
  - The **home directory of the root (admin) user**.
  - 📌 Not to be confused with /. This is like /home/root, but specifically named /root for the superuser.
- **/lib and /lib64 — Essential Shared Libraries**
  - Holds **shared libraries** needed by programs in /bin and /sbin. They are like .dll files in Windows.
  - **Examples:**
    - libc.so.6 – standard C library
    - ld-linux.so – dynamic linker/loader

Linux Filesystem Hierarchy

- **/usr — User System Resources**
  - Contains **non-essential programs and libraries** used by users. It is often the largest directory.
  - **Key subdirectories:**
    - /usr/bin – user commands (e.g. gcc, python, vim)
    - /usr/sbin – non-critical system binaries
    - /usr/lib – libraries for /usr/bin
    - /usr/share – architecture-independent data (icons, docs)
- **/var — Variable Data**
  - Stores **variable or frequently changing files** like logs, caches, mail spools, and temporary files created by programs.
  - **Examples:**
    - /var/log/syslog – system logs
    - /var/spool/cron – scheduled tasks
    - /var/cache – package manager caches

Linux Filesystem Hierarchy

- **/tmp — Temporary Files**
  - Holds **temporary files** used by applications or users.
  - 📌 Cleared on reboot.
- **/dev — Device Files**
  - Linux represents all hardware devices as **files** in this directory.
  - **Examples:**
    - /dev/sda1 – hard disk partition
    - /dev/ttyUSB0 – USB serial port
- **/proc — Kernel and Process Information**
  - A **virtual filesystem** that exposes runtime system information provided by the Linux kernel.
  - **Examples:**
    - /proc/cpuinfo – CPU details
    - /proc/meminfo – memory usage
    - /proc/[PID]/ – process-specific info

Linux Filesystem Hierarchy

- **/sys — System and Device Tree**
  - Another **virtual filesystem**. Unlike /proc, this is structured hierarchically and is more hardware-oriented.
  - **Examples:**
    - /sys/class/net/ – network interfaces
    - /sys/block/ – block devices like disks
- **/media and /mnt — Mount Points**
  - /media – for **automounted devices** (like USB, CD-ROM)
  - /mnt – for **temporary/manual mounts** by sysadmins
  - example
    - mount /dev/sdb1 /mnt/usb
- **/opt — Optional Software**
  - Used for installing **third-party applications** or proprietary software not managed by the distribution's package manager.
  - **Examples:**
    - /opt/google/chrome/
    - /opt/teamviewer/

Linux Filesystem Hierarchy

- **/boot — Boot Files**
  - Contains all files **required to boot the Linux system**, such as the kernel and bootloader configuration.
  - **Common files:**
    - vmlinuz – compressed Linux kernel
    - initrd – initial RAM disk
    - grub/ – bootloader config files
  - 📌 Often mounted as a separate partition.

Essential Linux Commands (with Examples)

- File and Directory Management
  - pwd # Print current working directory
  - ls -l # List files in long format
  - cd /path # Change directory
  - mkdir name # Create directory
  - rm -rf name # Remove files/directories
  - cp src dest # Copy
  - mv src dest # Move or rename
- Viewing File Content
  - cat file.txt # Print file
  - less file.txt # Scroll file content
  - head file.txt # First 10 lines
  - tail -f file.txt # View logs in real-time
- Finding Files and Text
  - find /path -name "*.c"
  - grep "main" file.c

Essential Linux Commands (with Examples)

- File Permissions and Ownership
  - ls -l # Show permissions
  - chmod 755 # Change permission
  - chown user:group file
  - r = read, w = write, x = execute
  - Owner / Group / Others (e.g., -rwxr-xr--)

- Useful Tools
  - df -h # Disk space
  - du -sh * # Directory size
  - top / htop # System processes
  - ps aux # Process list
  - kill -9 PID # Kill process

Working with Package Managers

- For Ubuntu/Debian:
  - sudo apt update
  - sudo apt install package-name
  - sudo apt remove package-name
  - dpkg -l | grep package
- For Fedora:
  - sudo dnf install package-name

Networking Basics

```
ip a          # Show network interfaces
ping 8.8.8.8   # Test connectivity
wget http://...  # Download files
scp file user@host:/path  # Secure file copy
ssh user@ip    # Remote login
```

Archiving and Compression

tar -cvf archive.tar folder/

tar -xvf archive.tar

gzip file.txt

gunzip file.txt.gz

Disk and Filesystem Management

mount /dev/sdb1 /mnt
umount /mnt
lsblk
fdisk -l
df -h

Systemd and Services

systemctl status ssh
systemctl start nginx
systemctl enable apache2

What is Bash?

- Bash stands for **Bourne Again Shell**.
- It is a **command-line interpreter** used in most Linux distributions.
- Developed as a **free and improved replacement** for the original Unix shell sh.
- Bash is used for:
  - Running system commands
  - Writing shell scripts
  - Automating tasks

What is a Shell?

- A **shell** is a program that takes commands from the keyboard and gives them to the operating system.
- Acts as a **command-line interface (CLI)**.
- Types of shells:
  - sh (Bourne Shell)
  - bash (Bourne Again Shell)
  - zsh, ksh, csh, etc.

Key Features of Bash

- Interactive command execution
- Script writing and execution
- Variables and arrays
- Conditional logic (if, case)
- Looping (for, while, until)
- Functions
- Input/output redirection
- Command history and completion

Why Use Bash?

- **Pre-installed** on most Linux systems
- Great for **automating repetitive tasks**
- Used widely in:
  - Embedded Linux development
  - DevOps and system administration
  - Server configuration and deployment
- Lightweight and scriptable

The Shebang (#!/bin/bash)

- The first line in a bash script is called the **shebang**:
  - #!/bin/bash
- Tells the system to use the Bash interpreter to run the script.
- Must be the **first line** in the script file.

How to Check Your Shell:

- echo $SHELL
- Displays the path to the current shell (e.g., /bin/bash)

We will cover these skills

- Understand how Embedded Linux differs from Desktop Linux
- Learn the architecture of Embedded Linux systems
- Get familiar with bootloader, kernel, root filesystem, and device tree
- Explore toolchains and cross-compilation
- Learn how to prepare and boot Embedded Linux on real hardware (e.g., STM32MP1 / Raspberry Pi)
- Gain hands-on experience with Buildroot for building a custom Linux OS

Chapter 2 Linux for Embedded Development

What is Embedded Linux

- Embedded Linux is a **lightweight, customized Linux OS** for embedded systems.
- Used in:
  - Routers, TVs, industrial control systems, automotive ECUs
  - IoT devices, edge computing devices, etc.
  - Typically runs on **ARM architecture**
  - Focuses on **performance, size, and reliability**

🔍 **Notice:**

- Embedded Linux is not a different Linux kernel—it's the same Linux, just **customized** and **optimized** for small devices with limited resources.

Embedded Linux System Architecture

- An embedded Linux system is made up of several interdependent components, each serving a distinct purpose. These components work together to create a functional, customizable, and efficient system suitable for resource-constrained devices.
- Understanding these components is **essential** before diving into kernel development, driver integration, or system customization.
- **Key Components:**
  - **Bootloader** (e.g., U-Boot)
  - **Linux Kernel**
  - **Device Tree**
  - **Root Filesystem (rootfs)**
  - **User Applications**

🔍 **Notice:**

- This modular structure allows customization and optimization for different embedded projects.

Embedded Linux System Architecture

🧩 Bootloader: bootloader is the **first software** that runs when the embedded system is powered on. It is responsible for:

- Initializing low-level hardware (RAM, clocks, pins)
- Loading the Linux kernel and Device Tree into memory
- Passing control to the kernel

🔧 **Examples:**

- **U-Boot** (most popular in embedded Linux)
- Barebox
- Das U-Boot SPL (for minimal hardware init)

🧠 **Why it matters:**

- You can customize the bootloader to:
  - Choose from multiple kernels
  - Boot into recovery
  - Update firmware over the network (TFTP, USB, etc.)

Embedded Linux System Architecture

🧩 Linux Kernel

- 📘 **What it is:**
  - The **Linux kernel** is the core of the operating system. It interacts directly with hardware and provides an interface for user-space applications to use the system's resources.

🧠 Responsibilities:

- Hardware abstraction and driver management
- Process scheduling and memory management
- File system support
- Networking stack
- Power and resource control

🔧 Common Formats:

- **zImage** or **uImage** (compressed kernel images)
- **vmlinuz** (used in desktop/server Linux)

💡 Configuration:

- You can enable/disable kernel features using make menuconfig when compiling the kernel.

Embedded Linux System Architecture

🧩 Device Tree Blob (DTB)

- 📘 **What it is:**
  - The **Device Tree** is a data structure that describes the hardware layout to the Linux kernel, especially on platforms that do not support self-discovery (like ARM).

📂 **Format:**

- Source file: .dts (Device Tree Source)
- Compiled file: .dtb (Device Tree Blob)

🧠 **Contains:**

- CPU, RAM, peripherals
- GPIO, UART, I2C, SPI definitions
- Interrupt mappings

🔧 **Why it's needed:**

- Instead of hardcoding hardware details, the kernel reads the .dtb to know:
- What hardware exists
- How to configure it
- What drivers to load

Embedded Linux System Architecture

🧩 **Root Filesystem (RootFS)**

📘 **What it is:**

- The **Root File System** contains all the user-space binaries, libraries, configuration files, and init scripts required to run the system.

📂 **Typical directories:**

- /bin, /sbin: essential command binaries
- /etc: system configuration
- /lib: shared libraries
- /dev: device nodes
- /proc, /sys: virtual system info
- /usr: user utilities and applications

🔧 **Formats:**

- **ext4**, **squashfs**, **jffs2**, **UBIFS**

🛠️ **Created by:**

- **Buildroot**, **Yocto**, **OpenEmbedded**, or manually

Embedded Linux System Architecture

Init System

What it is:

- The **init** system is the first user-space program launched by the kernel. It manages:
- Starting services
- Mounting filesystems
- Running startup scripts

Examples:

- init from BusyBox (lightweight and embedded-friendly)
- systemd (more advanced, less common in embedded)

Config File:

- /etc/inittab (BusyBox)
- /etc/init.d/ scripts

Embedded Linux System Architecture

🧩 **User Applications**

📘 **What it is:**

- These are the actual programs, utilities, and services you write or integrate for your embedded product.

🔧 **Examples:**

- Sensor monitoring scripts
- Network daemons
- UI interfaces
- Data loggers
- Remote update clients
- These are usually written in **C/C++**, **Shell**, or even **Python** (if resources allow).

Embedded Linux System Architecture

📌 Summary Table

Component	Description	Common Tools
Bootloader	Initializes hardware, loads kernel	U-Boot, Barebox
Kernel	Core OS: manages hardware and system calls	Linux mainline, custom
Device Tree	Hardware description for non-discoverable platforms	.dts / .dtb files
Root File System	User-space libraries, tools, configurations	Buildroot, Yocto
Init System	Manages boot process and service startup	BusyBox, systemd
User Applications	Your product logic and software	C, Shell, Python

Boot Process in Embedded Linux

- **ROM Code (from SoC)**: Executes first, loads bootloader
- **Bootloader (e.g., U-Boot)**: Initializes RAM, loads kernel
- **Kernel**: Sets up device drivers, mounts root filesystem
- **Init Process**: Launches user space

Boot Process in Embedded Linux

The boot process in Embedded Linux typically follows **these main steps**:

- [1] Power On
- ↓
- [2] Boot ROM Code
- ↓
- [3] First-Stage Bootloader (SPL)
- ↓
- [4] Second-Stage Bootloader (U-Boot)
- ↓
- [5] Linux Kernel + Device Tree + Initramfs
- ↓
- [6] Root Filesystem Mount
- ↓
- [7] Init System & User Applications

Boot Process in Embedded Linux

- **Boot ROM Code (SoC-specific)**

✅ **What It Does:**

- Burned into the chip by the manufacturer
- Initializes minimal hardware (clocks, SRAM)
- Detects boot source: SD, NAND, eMMC, USB, UART
- Loads the **First-Stage Bootloader** from boot media

Boot Process in Embedded Linux

🔷 **First-Stage Bootloader (e.g., SPL - Secondary Program Loader)**

✅ **What It Does:**

- Initializes DRAM (RAM controller, timing)
- Sets up basic I/O (if needed)
- Loads the **Second-Stage Bootloader** into RAM

📌 **Common Example:**

- **SPL** in U-Boot for STM32MP1, Raspberry Pi, etc.

📂 **File:**

- Often called MLO, SPL, or part of u-boot-spl.bin

Boot Process in Embedded Linux

🔷 **Second-Stage Bootloader (e.g., U-Boot)**

✅ **What It Does:**

- Fully initializes hardware (MMC, Ethernet, UART, USB, etc.)
- Provides command-line interface over UART/Serial
- Loads:
  - **Linux Kernel (zImage/uImage)**
  - **Device Tree (.dtb)**
  - **Initramfs** (optional)
- Transfers control to the kernel

🔧 **Features:**

- Environment variables (bootargs, bootcmd)
- Network boot (TFTP, NFS)
- Firmware update via DFU, USB, MMC, etc.
- Secure boot support

Boot Process in Embedded Linux

🔷 **Linux Kernel Initialization**

✅ **What It Does:**

- Decompresses the kernel image
- Parses the Device Tree (hardware info)
- Mounts an **initial RAM disk** (initramfs), if present
- Starts the **init process** (PID 1)

🧠 **Notes:**

- This is where most kernel messages (dmesg) are printed
- Initializes drivers and memory management

Boot Process in Embedded Linux

- **Root Filesystem Mounting**
- ✅ **What It Does:**
  - Mounts the **final root filesystem** (usually on ext4, squashfs, etc.)
  - Executes /sbin/init or equivalent based on init system (e.g., BusyBox)
- 🔧 **Mount Sources:**
  - On-board flash (NAND, eMMC)
  - SD card

Boot Process in Embedded Linux

🔷 **Init System Execution**

✅ **What It Does:**

- Initializes system services
- Mounts system partitions (/proc, /sys, /dev)
- Starts background daemons and user apps
- Prepares system for normal operation

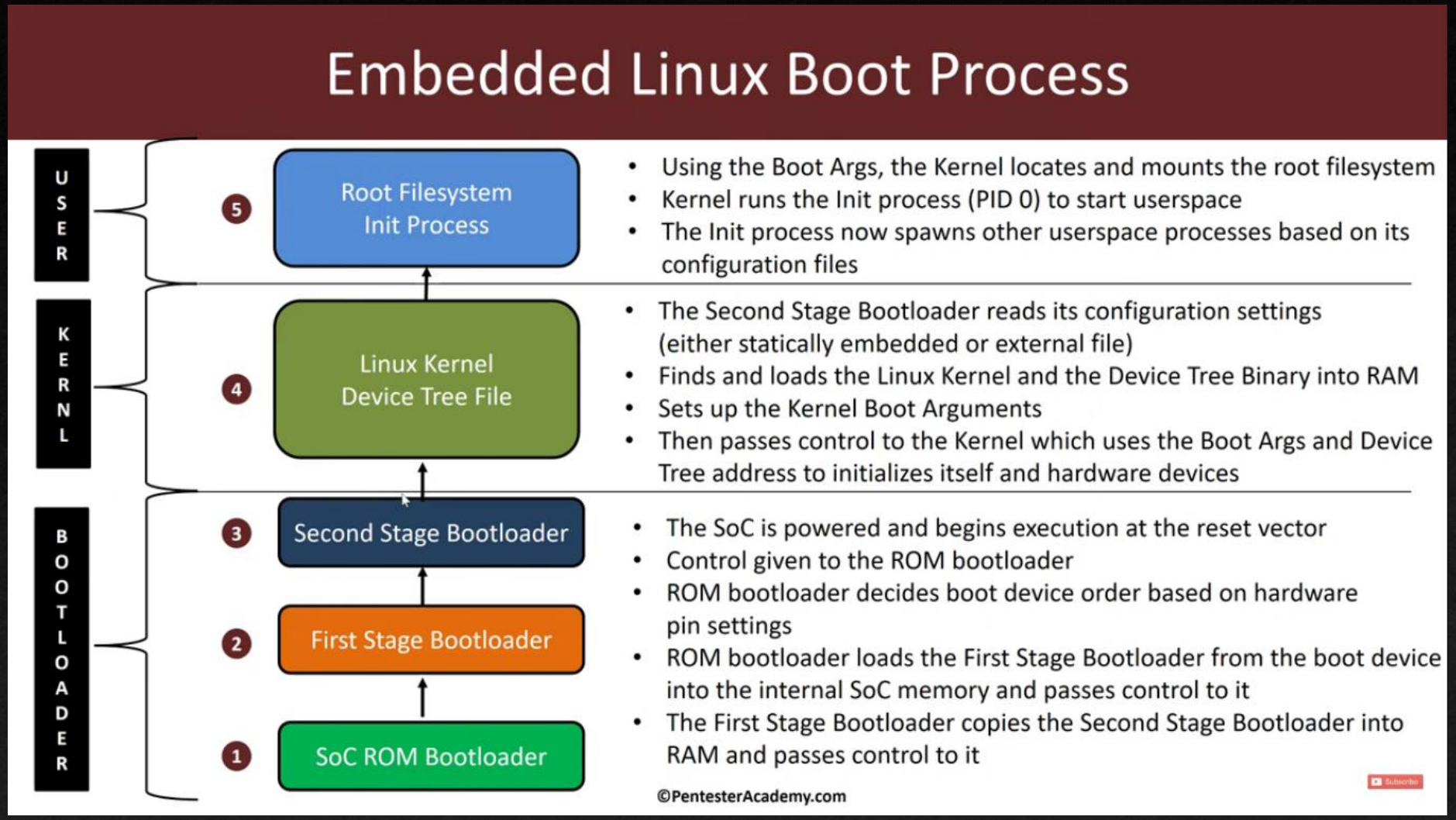
🧠 **Common Init Systems:**

- **BusyBox init** (lightweight, common in Buildroot)
- **systemd** (advanced, less common in embedded)
- **OpenRC, SysVinit** (optional)

Boot Process in Embedded Linux

Stage	Component	File/Tool	Location
ROM	Boot ROM	–	Fixed in SoC
1st	SPL	bootcode.bin	boot partition
2nd	U-Boot	u-boot.bin	boot partition
3rd	Kernel	zImage	boot partition
4th	DTB	bcm2835-rpi.dtb	boot partition
5th	RootFS	rootfs.ext4	root partition
6th	Init	/sbin/init	inside rootfs

Boot Process in Embedded Linux



Toolchains & Cross-Compilation

- **Cross-compilation**: Compiling on a host system (x86) for a target (ARM)
- Tools: gcc, ld, as, objcopy, etc.
- Toolchains include:
  - Compiler (e.g., arm-linux-gnueabihf-gcc)
  - C Library (e.g., glibc, musl, uclibc)
  - Debugger (e.g., gdb)
- Popular toolchains: Buildroot-generated, Yocto-generated

Toolchains & Cross-Compilation

What is a Toolchain?

- A **toolchain** is a collection of tools used to compile, assemble, and link software for a specific target system.

Tool	Purpose
gcc	Compiler (C/C++ source to object code)
as	Assembler (assembly to object)
ld	Linker (combine objects to executable)
ar	Archiver (library creation tool)
strip	Binary size reducer (removes symbols)
objdump / readelf	Inspect binary formats
gdb	Debugger
make / cmake	Build automation
C Library	glibc, musl, or uClibc

Toolchains & Cross-Compilation

🧭 Native vs. Cross Compilation

Term	Description
Native Compilation	Building software on the same machine where it will run
Cross Compilation	Building software on one machine (**host**) for a different system (**target**)

Toolchains & Cross-Compilation

📍 Why Cross-Compile in Embedded Linux?

- Target (STM32MP1, Raspberry Pi, etc.) may not have enough CPU, memory, or disk to build software
- Speeds up compilation
- Enables development on powerful hosts (x86_64) for limited devices (ARM Cortex-A)

🔄 Host vs. Target vs. Build Systems

Term	Definition
Build System	Where tools are built (e.g. when building a toolchain)
Host System	Where the compiled tools run (e.g. your dev PC)
Target System	Where the compiled program will run (e.g. STM32MP1 board)

Toolchains & Cross-Compilation

📦 Prebuilt Toolchains (Ready to Use)

- Many distributions and projects provide prebuilt toolchains:

Provider	Link	Notes
Buildroot	Generates its own toolchain	
Yocto SDK	Comes with environment-setup script	
Linaro	https://www.linaro.org	Optimized for ARM
ARM GNU Toolchain	https://developer.arm.com	Official ARM tools

Toolchains & Cross-Compilation

🧪 **Example (Linaro Toolchain for ARM)**

```
wget https://releases.linaro.org/components/toolchain/binaries/latest/arm-linux-gnueabi
tar xf gcc.tar.xz
export PATH=$PATH:/path/to/toolchain/bin
```

Now, use:

- arm-linux-gnueabihf-gcc main.c -o main

Toolchains & Cross-Compilation

🧰 Toolchain Components (File View)

```
toolchain/
├── bin/
|   ├── arm-linux-gnueabihf-gcc
|   ├── arm-linux-gnueabihf-ld
├── lib/
├── include/
├── libexec/
```

🏗️ How to Build Your Own Toolchain (Optional)

- **Buildroot** → make menuconfig → Toolchain
- **Yocto** → bitbake meta-toolchain

Toolchains & Cross-Compilation

How Toolchain Integrates with Buildroot

- In Buildroot:

```
Toolchain → Toolchain type: External / Buildroot internal
Target Architecture: ARM Cortex-A7
C Library: musl / uClibc / glibc
```

- Buildroot can:
  - Download and configure an external toolchain
  - Or build one from scratch

Toolchains & Cross-Compilation

💡 Common Issues & Tips

Problem	Cause / Solution
Wrong architecture	Check file output of compiled binary
"Command not found"	Toolchain bin not in PATH
Linking errors	Mismatch between libc and headers
"Illegal instruction"	Using wrong CPU flags for your target

Introduction to Build Systems

- Purpose: Automatically build toolchain, kernel, rootfs
- Common systems:
  - **Buildroot** ✅ (used in this course)
  - Yocto

- **Why Use Buildroot?**
  - Simple configuration via ***make menuconfig***
  - Fast build process
  - Ideal for small to medium projects
  - Easily integrates custom applications

Introduction to Build Systems

Buildroot and Yocto: Full Embedded Build Systems

Feature	Buildroot	Yocto
Simplicity	Simple, fast to learn	Complex, highly customizable
Toolchain	Can build or use external	Uses Poky or custom
RootFS	Generates minimal Linux rootfs	Generates full-featured images
Package support	Manual, defconfig style	BitBake recipes (.bb files)
Use case	Quick development	Production-grade customization

Preparing Your Host Linux System

- **Host System Requirements**
  - Ubuntu/Debian-based Linux recommended
  - Required packages:
    - sudo apt install build-essential git gcc make python3 unzip bc \
    - libncurses5-dev libssl-dev cpio rsync flex bison \
    - device-tree-compiler u-boot-tools

Folder Structure and Best Practices

- Use separate folders:
  - ~/embedded-linux/
  - ├── buildroot/
  - ├── toolchain/
  - ├── kernel/
  - ├── output/

Hands-On with Buildroot

- **Buildroot – First Build**
  - Clone repository:
    - git clone https://github.com/buildroot/buildroot.git
    - cd buildroot
    - make raspberrypi4_defconfig # or your target defconfig
    - make
  - Output:
    - Kernel image
    - Root filesystem (initramfs/rootfs)
    - Bootloader (optional)
- **Deploying on Hardware:**
  - Copy images to SD card or flash storage
  - Configure bootloader (U-Boot) if needed
  - Boot and debug using serial console

We will cover these skills

- Download, extract, and configure **Buildroot**.
- Understand Buildroot's **directory structure** (, , ,).
- Select and configure:
  - Target architecture
  - Toolchain options
  - Kernel optionse
  - Root filesystem type (ext4, squashfs, cpio, etc.)
- Build **minimal and full-featured root filesystems**.
- ntegrate **custom packages** into Buildroot.
- Generate **bootable images** for SD cards or flash memory.

Chapter 3 Buildroot Fundamentals and Filesystem Generation

Buildroot Fundamentals

What is Buildroot?

- **Definition**:
  Buildroot is an open-source tool that automates the process of creating a complete **embedded Linux system** for your target hardware.
- **Purpose**:
  It **cross-compiles** the kernel, bootloader, root filesystem, and user-space tools for embedded devices.
- **Key Features**:
  - Lightweight and simple to configure.
  - Supports many CPU architectures (ARM, x86, MIPS, PowerPC, RISC-V, etc.).
  - Generates **minimal, customized Linux systems**.
  - Integrates easily with cross-compilers and toolchains.

Why Use Buildroot?

- **Fast development** — From source code to bootable image in a single step.
- **Customizability** — Include only what's needed (reducing footprint).
- **Reproducibility** — Same configuration builds the same system every time.
- **Integration** — Works with various build systems like Makefiles, CMake, etc.

Filesystem Generation Workflow in Buildroot

The **filesystem** in Linux is the directory structure that the kernel uses to store programs, libraries, and configuration files.
Buildroot automates **root filesystem creation**:

- Select Target Configuration
  - Choose target architecture (e.g., ARM Cortex-A7).
  - Select toolchain.
  - Configure packages, kernel, bootloader, and rootfs type.
- **Download Sources**
  - Buildroot downloads required packages, kernel, and bootloader sources from mirrors or Git repositories.
- **Compile Components**
  - Uses **cross-toolchain** to compile:
    - Bootloader (U-Boot)
    - Kernel
    - User-space applications
    - Libraries

Filesystem Generation Workflow in Buildroot

- **Assemble Root Filesystem**
  - Buildroot arranges all compiled files into a root filesystem.
  - Supports formats: ext4, SquashFS, CramFS, initramfs, etc.
- **Create Final Image**
  - Generates bootable images like:
    - sdcard.img (SD card image)
    - rootfs.ext4 (filesystem image)
    - uImage or zImage (kernel)
    - u-boot.bin (bootloader)

Directory Structure in Buildroot

Directory / File	Description
arch/	Architecture-specific configurations and code. Contains subfolders for ARM, x86, MIPS, etc.
board/	Board-specific configurations, boot scripts, and kernel patches. Each vendor may have a subfolder.
configs/	Predefined configuration files (defconfig) for popular boards (e.g., raspberrypi3_defconfig).
docs/	Official Buildroot documentation in text and HTML formats.
fs/	Filesystem generator scripts for different formats (ext2, initramfs, squashfs, etc.).
linux/	Kernel build support files and patches.
package/	All supported software packages (BusyBox, dropbear, etc.) with .mk build scripts.
system/	Scripts for init, startup, and device management.
toolchain/	Scripts and configurations for building or using external toolchains.
output/ *(generated after build)*	Contains the results of the build process:
build/	Temporary build directories for each package.
images/	Final bootloader, kernel, and filesystem images.
staging/	Temporary root filesystem used during build.
host/	Tools compiled for the host (PC) environment.
target/	Final root filesystem contents before packaging.
Makefile	Main Buildroot build script. You always start builds from here.
Config.in	Menu configuration definition file for make menuconfig.

Example Filesystem Layout Generated by Buildroot

- (inside target/ or final image)

```
/
├── bin/      → Essential binaries (ls, cp, cat…)
├── dev/      → Device files
├── etc/      → System configuration files
├── lib/      → Shared libraries
├── root/     → Home directory for root user
├── sbin/     → System binaries
├── tmp/      → Temporary files
├── usr/      → User programs and data
└── var/      → Variable data (logs, spool)
```

practical list of important Buildroot packages

Development & Debugging Tools

These are essential for testing, debugging, and managing the system:

- busybox *(already included, core utilities)*
- bash – better shell than sh
- nano or vim – text editor
- htop – process monitor
- strace – trace system calls
- ltrace – trace library calls
- gdb / gdbserver – debugging
- procps – process and system information commands (ps, top, free)
- file – detect file type
- which – locate executables
- psmisc – tools like killall, fuser

practical list of important Buildroot packages

Networking Tools

f your board has Ethernet/Wi-Fi:

- dropbear or openssh – SSH server/client
- curl – transfer data over HTTP, HTTPS, FTP, etc.
- wget – download files from the internet
- iproute2 – modern networking commands (ip, ss)
- net-tools – legacy tools (ifconfig, netstat)
- iperf3 – network performance test
- ethtool – Ethernet interface settings
- tcpdump – packet capture
- bridge-utils – network bridging

practical list of important Buildroot packages

File System & Storage

For handling USB drives, SD cards, etc.:

- e2fsprogs – tools for ext2/3/4 filesystems
- dosfstools – FAT filesystem tools
- mtd-utils – flash memory tools
- parted – partitioning tool
- ntfs-3g – NTFS support

practical list of important Buildroot packages

Compression & Archive Tools

For installing/unpacking software:

- tar – archive utility
- gzip, bzip2, xz, zip, unzip – compression utilities

Time & Date

For RTC and time sync:

- ntp or chrony – time synchronization
- hwclock – manage hardware clock (comes with busybox, but util-linux version is better)

System Management

- util-linux – essential utilities (mount, umount, fdisk, hwclock)
- inotify-tools – filesystem event monitoring

practical list of important Buildroot packages

Programming Languages (Optional)

If your system runs scripts or interpreters:

- python3 – Python runtime
- lua – lightweight scripting
- perl – Perl interpreter

Extra for Embedded Development

- i2c-tools – interact with I²C devices
- can-utils – CAN bus tools
- usbutils – USB device listing (lsusb)
- pciutils – PCI device listing (lspci)
- devmem2 – access physical memory from userspace

Structure of a systemd Service Unit File

- [Unit] Section
- [Service] Section
- [Install] Section

Structure of a systemd Service Unit File

[Unit] Section

Directive	Description
`Description=`	Short description of the service.
`Documentation=`	Reference to manuals or docs (e.g., `man:nginx(8)`).
`Requires=`	Units that **must** be active for this unit to start. If they fail, this unit also stops.
`Wants=`	Units that should be active if possible, but won't cause failure if not.
`Before=` / `After=`	Controls startup/shutdown ordering.
`Conflicts=`	Units that should not run at the same time.

Structure of a systemd Service Unit File

[Unit] Section

- [Unit]
- Description=My Custom Web Application
- Documentation=https://myapp.example.com/docs
- Requires=network.target
- After=network.target

Structure of a systemd Service Unit File

[Service] Section

Directive	Description
`Type=`	Defines how systemd expects the service to behave (`simple`, `forking`, `oneshot`, `notify`, `dbus`).
`ExecStart=`	Command to start the service. Required.
`ExecReload=`	Command to reload the service without stopping.
`ExecStop=`	Command to stop the service.
`Restart=`	Restart policy (`no`, `on-success`, `on-failure`, `always`).
`RestartSec=`	Delay before restarting.
`User=` / `Group=`	Run the service as a specific user/group.
`Environment=`	Set environment variables.

Structure of a systemd Service Unit File

[Service] Section

```
[Service]
Type=simple
ExecStart=/usr/bin/python3 /opt/myapp/server.py
Restart=on-failure
RestartSec=5
User=myappuser
Environment=APP_MODE=production
```

Structure of a systemd Service Unit File

[Install] Section

Directive	Description
`WantedBy=`	Target(s) the service should start with when enabled (most common is `multi-user.target`).
`RequiredBy=`	Strong dependency for another unit (less common in normal services).
`Also=`	Additional units to enable/disable along with this one.

```
[Install]
WantedBy=multi-user.target
```

Structure of a systemd Service Unit File

```
[Unit]
Description=My Custom Web Application
After=network.target
Requires=network.target

[Service]
Type=simple
ExecStart=/usr/bin/python3 /opt/myapp/server.py
Restart=on-failure
RestartSec=5
User=myappuser
Environment=APP_MODE=production

[Install]
WantedBy=multi-user.target
```

We will cover these skills

- Understand what cross-compilation is and why it’s used.
- Locate and use the Buildroot cross-toolchain.
- Cross-compile a simple C program for the LicheePi Zero.
- Transfer and run programs on the target board.
- Add a custom package to Buildroot.

Chapter 4 Cross-Compilation

Introduction to Cross-Compilation

- **What is cross-compilation?**
  - Compiling code on one machine (host) to run on another (target) with a different architecture.
  - Example: Build on x86_64 laptop → run on ARM Cortex-A7 (LicheePi Zero).
- **Why cross-compile?**
  - Target board may have limited CPU, RAM, or storage.
  - Easier to use powerful PC for building software.
- **Key components**
  - **Host system**: Your development PC (Ubuntu, Debian, etc.)
  - **Target system**: LicheePi Zero (ARMv7).
  - **Toolchain**: Set of tools (compiler, linker, etc.) built for the target architecture.

Toolchains in Buildroot

- Buildroot automatically generates a **cross-compilation toolchain** when you build.
- Location after build:
- output/host/bin/arm-linux-*
- Main tools:
  - arm-linux-gcc → C compiler
  - arm-linux-ld → Linker
  - arm-linux-strip → Binary size reducer

Cross-Compile Workflow

- **Write source code** on host (e.g., hello.c).
- **Compile with cross-compiler**:
  - output/host/bin/arm-linux-gcc hello.c -o hello
- **Copy to target board** (via scp, SD card, etc.):
  - Scp hello root@192.168.1.50:/root/
- **Run on target**:
  - ./hello

Example – Hello World

- **Source code** (hello.c):

```
#include <stdio.h>

int main() {
    printf("Hello from LicheePi Zero!\n");
    return 0;
}
```

- **Cross-compile:**
  - output/host/bin/arm-linux-gcc hello.c -o hello
  - file hello
- Output should show:
  - ELF 32-bit LSB executable, ARM, EABI5 version 1 ...

Network Protocol

- Transmission Control Protocol (TCP)-------> Connection-oriented protocol
- User Datagram Protocol (UDP)----------> part of the Internet Protocol

Network Protocol

TCP (Transmission Control Protocol)

- **Connection-oriented**: TCP establishes a connection between client and server before transmitting data (like a phone call).
- **Reliable**: Ensures data arrives in order, retransmits lost packets, and checks errors.
- **Stream-based**: Data is read as a continuous stream, not divided into distinct packets.
- **Use cases**: Web browsing (HTTP/HTTPS), file transfer (FTP), SSH, email.

🔑 TCP requires:

- **Socket creation** (socket())
- **Binding** to a port (bind())
- **Listening** for incoming connections (listen())
- **Accepting** a client connection (accept())
- **Sending/Receiving** data (send() / recv())
- **Closing** connection (close())

Network Protocol

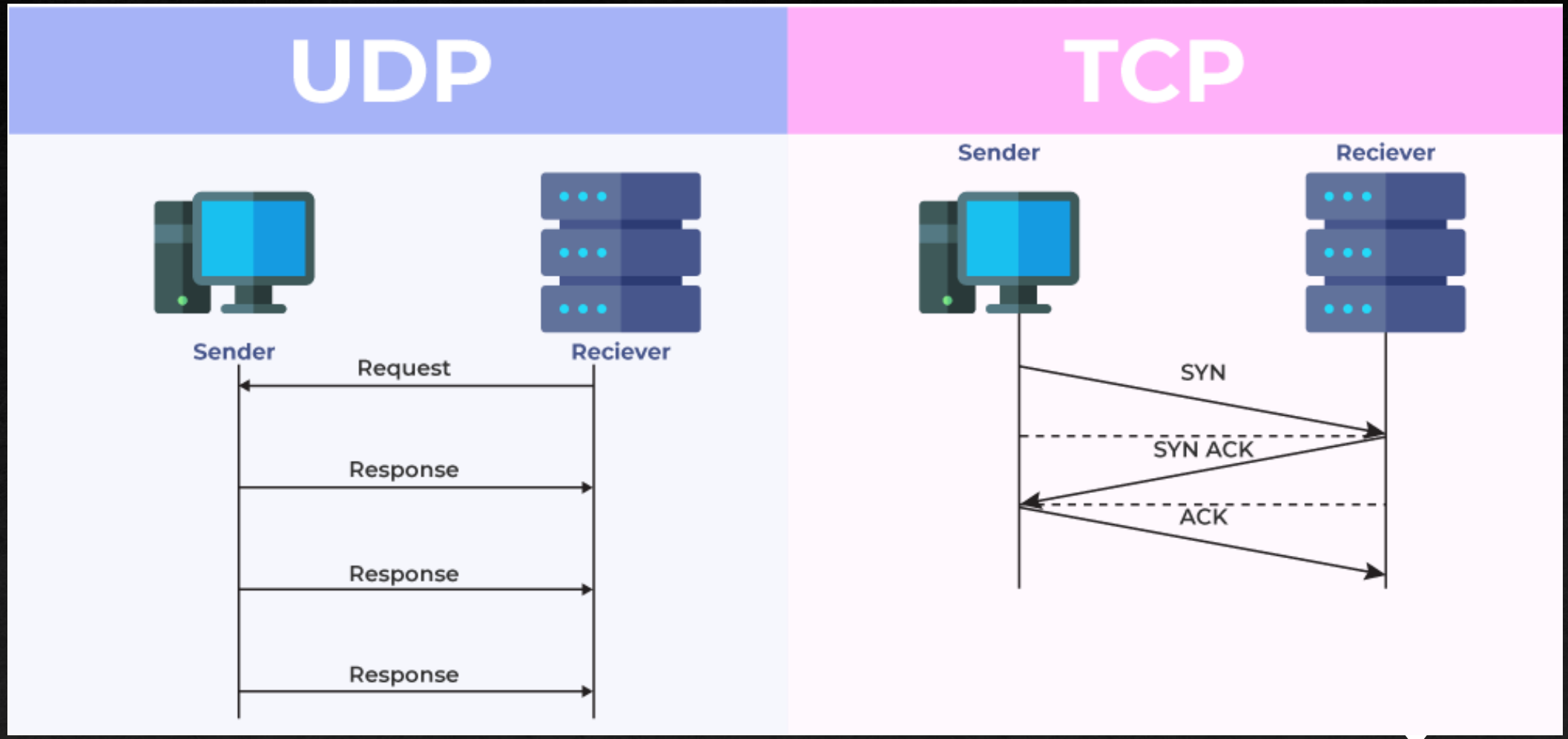
UDP (User Datagram Protocol)

- **Connectionless**: No setup required; packets (datagrams) are just sent (like sending letters without acknowledgment).
- **Unreliable**: No guarantee of delivery, order, or duplication check.
- **Message-based**: Each sendto() call corresponds to one datagram.
- **Fast and lightweight** compared to TCP.
- **Use cases**: Real-time apps like video streaming, VoIP, DNS, IoT sensors.

🔑 UDP requires:

- **Socket creation** (socket())
- **Binding** to a port (bind())
- **Receiving datagrams** (recvfrom())
- **Sending datagrams** (sendto())
- **Closing** socket (close())

Network Protocol



Chapter 5 Device Tree in Embedded Linux

We will cover these skills

- Understand the role of Device Tree in Embedded Linux.
- Differentiate between DTS, DTB, DTC and their purposes.
- Explain how Device Tree integrates into the Linux boot process (U-Boot → Kernel).
- Read and interpret the structure of a DTS file (nodes, properties, compatibility strings).

What is Device Tree?

- A **data structure** describing hardware to the Linux kernel.
- Replaces old hardcoded board files in kernel.
- Kernel = generic, Device Tree = hardware-specific.
- Benefits:
  - Kernel portability.
  - Easy hardware changes without recompiling kernel.
  - Supports multiple boards with same kernel.

Device Tree Files

- **DTS** (Device Tree Source) → human-readable text file.
- **DTB** (Device Tree Blob) → compiled binary used by kernel.
- **DTC** (Device Tree Compiler) → tool to compile DTS → DTB.
- 📂 Typical path in Linux source:
  - arch/arm/boot/dts/

Device Tree Structure

- Hierarchical structure (like a filesystem).
- **Nodes** = hardware blocks (CPU, memory, UART, GPIO).
- **Properties** = key-value pairs.
- 🔹 Example:

```dts
/ {
    model = "LicheePi Zero";
    compatible = "licheepi,zero", "allwinner,sun8i-v3s";

    memory {
        device_type = "memory";
        reg = <0x40000000 0x8000000>; // 128 MB RAM
    };

    soc {
        uart0: serial@01c28000 {
            compatible = "snps,dw-apb-uart";
            reg = <0x01c28000 0x400>;
            status = "okay";
        };
    };
};
```

Device Tree in Boot Flow

- Bootloader (U-Boot) loads:
  - Kernel image (zImage/uImage).
  - DTB (device tree blob).
- Kernel parses DTB → configures hardware.
- Drivers bind to hardware nodes via compatible.

LicheePi DTS Location

- For **Allwinner V3s SoC** (used in LicheePi Zero):
  - linux/arch/arm/boot/dts/sun8i-v3s-licheepi-zero.dts
- Includes generic SoC file:
  - sun8i-v3s.dtsi

Building DTB

- Compile manually with dtc:

```
dtc -I dts -O dtb -o sun8i-v3s-licheepi-zero.dtb sun8i-v3s-licheepi-zero.dts
```

- In Buildroot:
  - DTB built automatically with kernel.
  - DTB is placed in output/images/.

Deploying DTB

- Copy DTB to SD card /boot.
  Example:

```
cp output/images/sun8i-v3s-licheepi-zero.dtb /media/boot/
```

- Update U-Boot config if needed (boot.cmd).
- Reboot → Kernel loads new DTB.

Modifying Device Tree (Steps)

- Locate DTS file (sun8i-v3s-licheepi-zero.dts).
- Edit node → enable/disable hardware.
- Rebuild DTB (make dtbs in Buildroot).
- Deploy DTB to board.
- Verify with dmesg and /proc/device-tree/.

Example: Enable I²C on LicheePi

- Add I²C node in DTS:

```
&i2c0 {
    pinctrl-names = "default";
    pinctrl-0 = <&i2c0_pins>;
    status = "okay";
};
```

- 📌 After rebuild → check:

```
dmesg | grep i2c
ls /dev/i2c-*
```

Example: Add GPIO LED

```
leds {
    compatible = "gpio-leds";

    user_led: led-0 {
        label = "green:user";
        gpios = <&pio 0 10 GPIO_ACTIVE_HIGH>; // Port A10
        default-state = "on";
    };
};
```

Debugging Device Tree

- ls /proc/device-tree/
- dtc -I dtb -O dts sun8i-v3s-licheepi-zero.dtb > dump.dts

We will cover these skills

- Explain the role and importance of GDB in embedded system development.
- Configure and use **gdbserver** on the target and **cross-GDB** on the host.
- Cross-compile applications with debugging symbols and execute them under GDB.
- Apply fundamental debugging techniques: breakpoints, stepping, variable inspection, and call stack analysis.
- Utilize advanced debugging features including conditional breakpoints, watchpoints, core dump analysis, and runtime variable modification.
- Integrate GDB with development environments (e.g., VSCode, Eclipse) to streamline debugging.
- Adopt best practices for efficient debugging in resource-constrained embedded environments.

Chapter 6 GNU Debugger (GDB) in Embedded Linux

Introduction

- Debugging = process of finding and fixing errors in software.
- In Embedded Linux: very important since no direct access to display/logs.
- GDB = *GNU Project Debugger*.
- Allows you to:
  - Run programs step by step
  - Set breakpoints
  - Inspect variables/memory
  - Debug remotely on embedded targets

Why Use GDB?

- Traditional debugging (e.g., printf) is limited.
- Kernel/driver debugging needs more powerful tools.
- GDB advantages:
  - Source-level debugging
  - Non-intrusive
  - Works over serial or TCP/IP
  - Supports multi-threaded debugging

GDB Architecture

- Host machine (PC): runs GDB (cross-debugger).
- Target (LicheePi): runs gdbserver.
- Connection: Serial / Ethernet / USB.
- Workflow diagram:
  - [Host PC: arm-linux-gdb] ←→ [Target: gdbserver + Program]

Installing GDB

- On Host (PC):
  - Buildroot provides arm-linux-gdb in output/host/bin/.
- On Target (LicheePi):
  - Enable gdb and gdbserver in Buildroot → menuconfig.
  - Verify installation:
    - gdbserver --version
    - arm-linux-gdb --version

Compiling for Debug

- Use -g flag in GCC:
  - arm-linux-gcc -g -o test test.c
- Keeps debug symbols (needed for GDB).
- Without -g you only see assembly.

Basic GDB Commands

- run – start program
- break <line/function> – set breakpoint
- continue – resume execution
- next – step over
- step – step into function
- print <var> – show variable value
- backtrace – show call stack
- info registers – inspect CPU registers

Remote Debugging Workflow

- Copy program to target.
- Run on target:
  - gdbserver :1234 ./test
- On host:
  - arm-linux-gdb ./test
    - (gdb) target remote <IP>:1234

Example Debugging Session

- Source code: factorial function.
- Run gdbserver :1234 ./factorial on target.
- On host:
  - (gdb) break main
  - (gdb) run
  - (gdb) step
  - (gdb) print n
  - (gdb) continue

Advanced Features

- Conditional breakpoints:
  - break foo if x > 5
- Watchpoints (track variable change):
  - watch var
- Modify variable
  - set var x = 10

GDB + IDEs

- GDB can be integrated into:
  - Eclipse CDT
  - VSCode + cpptools
  - CLion with Remote Debug plugin
- Useful for GUI debugging.

Compilation Flags for GDB

- When debugging programs with **GDB** in an **Embedded Linux** environment :
  - compile your code is **critical**.
  - The compiler (GCC or cross-GCC from Buildroot)
  - can include or strip debugging information.
  - optimize code in ways that hide variables.
  - inline functions that make stepping confusing.

Compilation Flags for GDB

- **Debugging Information Flags**

Flag	Description
`-g`	Generate debugging information in the executable (symbol table, source line mapping, variable names). This is the **minimum required** for GDB.
`-g1`	Generate minimal debug info (enough for backtraces, but not variable details). Produces smaller binaries.
`-g2`	Default level of `-g`. Includes most debug info, balance between detail and size.
`-g3`	Maximum debug info. Also includes macro definitions, preprocessor info. Useful when debugging complex programs.
`-ggdb`	Like `-g`, but generates debugging info tailored specifically for GDB (may include more details than `-g`).

Compilation Flags for GDB

- **Optimization and Debugging Interaction**

Flag	Effect	Debugging Impact
`-O0`	No optimization. Code matches source exactly.	✅ Best for debugging.
`-O1`	Basic optimization. Small reordering.	⚠️ Some variables may look strange in GDB.
`-O2`	Higher optimization (default in many toolchains).	⚠️ Harder to debug: inlined functions, optimized-out variables.
`-O3`	Aggressive optimization (loop unrolling, vectorization).	❌ Debugging very hard.
`-Os`	Optimize for size.	⚠️ May affect debugging.
`-Og`	Optimize for debugging. Keeps most debug info usable while still optimizing.	✅ Good balance between speed and debuggability.

Compilation Flags for GDB

- **Warnings**

Flag	Purpose
`-Wall`	Enable most useful warnings.
`-Wextra`	Enable even more warnings.
`-Werror`	Treat warnings as errors (forces cleaner code).

- **Linker Flags**

Flag	Purpose
`-static`	Statically link binary (no shared libs). Easier deployment, but bigger binary.
`-rdynamic`	Export symbols for use in GDB (for dynamically loaded code).
`-Wl,-Map=output.map`	Generate linker map file. Helps understand addresses and symbols.

This tool allows us to :

- select kernel options
- enable/disable features
- Build the kernel for our embedded device

Chapter 7
Linux Kernel menuconfig in Buildroot

What is linux-menuconfig?

linux-menuconfig is a tool :

- provided by Buildroot
- customize the Linux kernel
- configure kernel options
- device drivers
- filesystems, and more
- Text-based kernel configuration interface
- Based on Linux kernel kconfig system
- Runs via Buildroot: make linux-menuconfig
- Stores configuration in BR2_LINUX_KERNEL_CONFIG

Why Use linux-menuconfig?

- Select needed kernel features only
- Reduce kernel size for embedded systems
- Enable hardware-specific drivers
- Configure filesystems and networking support
- Apply patches and custom options

Launching linux-menuconfig in Buildroot

- Command: make linux-menuconfig
- Must configure Buildroot kernel first
- Requires ncurses installed on host
- Opens interactive menu

Main Categories in Linux menuconfig

- General setup
- Processor type and features
- Power management options
- Bus and device support (I2C, SPI, PCI, USB)
- Networking support
- File systems (ext4, squashfs, NFS, etc.)
- Device drivers (GPIO, UART, CAN, etc.)
- Kernel hacking / debugging

General Setup

- Kernel release name and version
- Init process selection
- Default system call options
- Support for initramfs or initrd

Processor Type and Features

- Select CPU architecture (ARM, x86, MIPS)
- Specify CPU variant (Cortex-A7, A53, etc.)
- Enable floating-point support
- SMP and multi-core options

Power Management Options

- CPU frequency scaling
- Sleep modes and suspend/resume
- Device runtime power management
- Low-power optimizations

Bus and Device Support

- Enable/disable I2C, SPI, CAN, UART
- PCI and USB subsystem support
- GPIO access and drivers
- Peripheral-specific options

Networking Support

- TCP/IP stack options
- Wireless and Ethernet drivers
- Networking protocols (IPv4, IPv6, PPP)
- Firewalling and security features

Filesystems

- Select root filesystem types: ext4, squashfs, tmpfs, etc.
- Network filesystems: NFS, CIFS
- Flash and block device support
- Journaling and compression options

Kernel Hacking / Debugging

- Enable debug messages
- Kernel profiling and tracing
- Magic SysRq key
- Logging and printk options

Saving the Kernel Configuration

- Save changes in .config
- Use make savedefconfig to create minimal defconfig
- Reuse configuration across projects
- Buildroot uses BR2_LINUX_KERNEL_CONFIG