Embedded Linux Development

Getting Started with the Raspberry Pi 4 Embedded Board

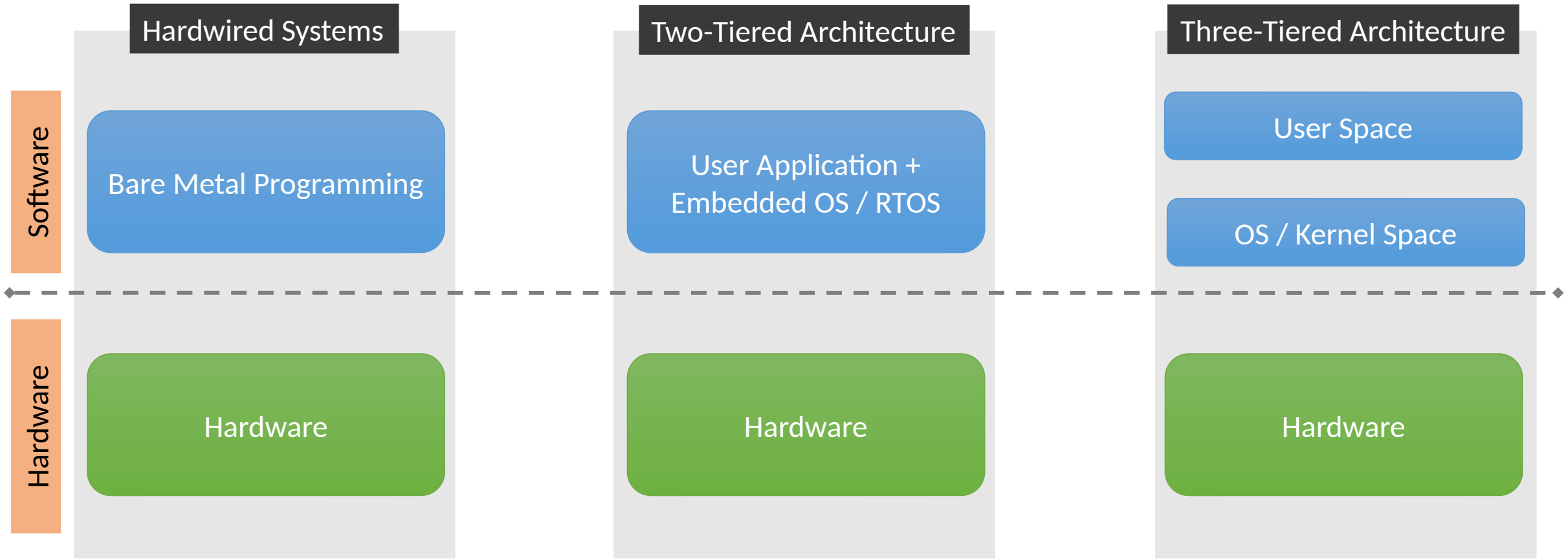
Outline

- Introduction to Embedded Operating Systems
  - Taking off from where we left
  - Host, Target, Developmen Env, Debug Env and Operating Systems
- The Linux Embedded System
  - The Basics - Tool chains, libraries, debuggers
  - The Board Support Package - Boot Loader, Kernel and File System
- Introduction to Raspberry Pi4

Classifications of Embedded Systems

- Owing to Memory Size & Power Consumption
  - **Small ES** - Low Power CPU, < 4MB ROM (Flash) and 8-16MB RAM
  - **Medium ES-** Med Power CPU, 32MB+ ROM and 64-128MB RAM with optional Secondary Storage
  - **Large ES** - High Power CPU, Large Memory Banks…
- Owing to Timing Constraints
  - Stringent Deadlines - Hard and Soft Realtime
  - Mild Deadlines - Timing is not a major issue
- Networkability
- User Experience

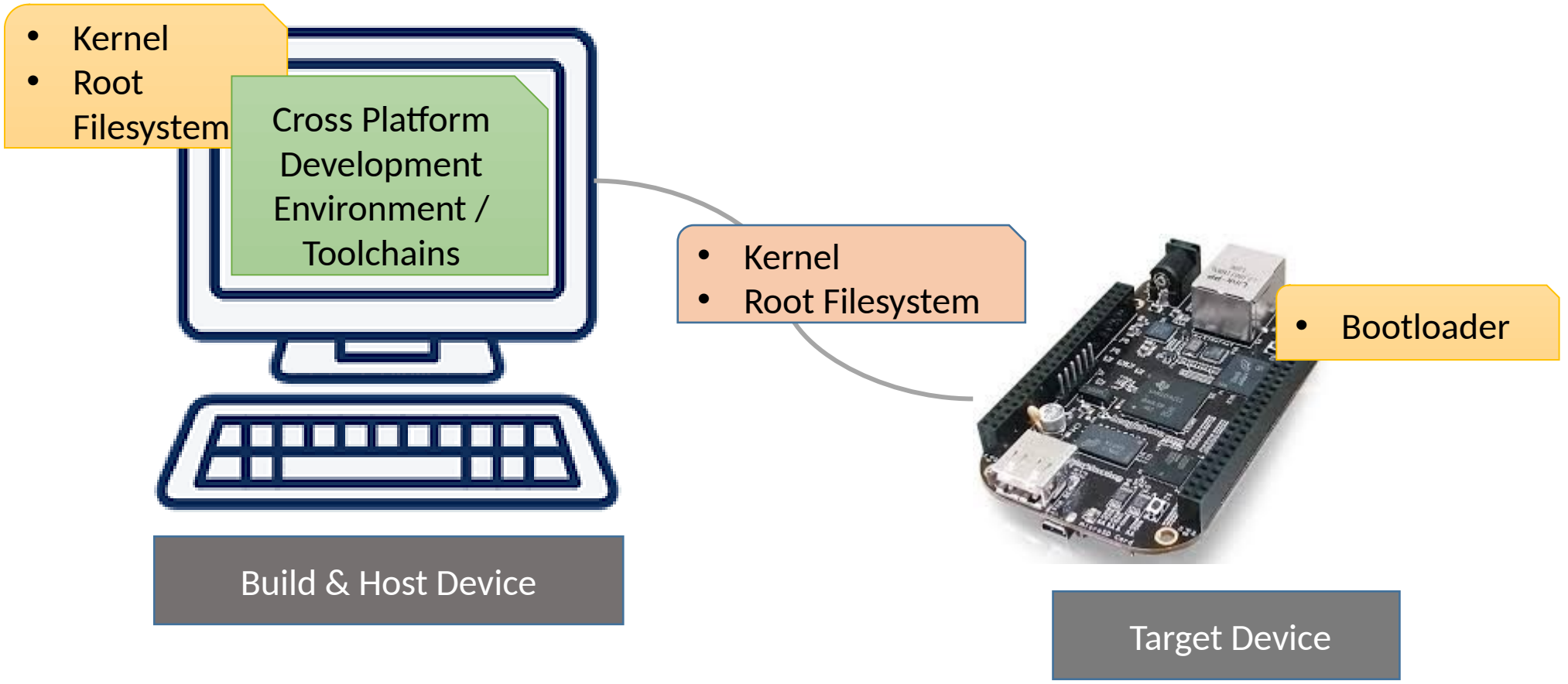
Classification of Embedded Systems Programming



High End Embedded Systems – Components of the Board Development Environment

- Software Terminologies
  - **Toolchain** - Set of executables that are required to compile and build an embedded systems image
  - **Boot Loader –** Piece of code which is required to load the Kernel and subsequent OS related software
  - **Kernel** - Part of the Operating System that provides its features and services
  - **Root File System** –
    - Software that abstracts the low-level hardware details and presents a logical file storage and access interface.
    - Contains all the files of the Operating System with a mechanism and programs to access them
    - Applications are developed and reside in the RFS and are accessed using various tools provided by the OS
  - **Libraries** – Pieces of code that implement specific functionality which can be used by other software
- Build System
  - One on which the toolchain is built
- Host Device
  - Any device capable of performing cross platform development
  - Linux Workstation, Unix System and Windows
  - Build the programs and create the object file, which will be executed on the target device
- Target Device
  - The device onto which developed software has to be flashed and executed
  - X86, ARM, PowerPC, MIPS processor etc

A Typical Embedded System Environment – Linked Setup

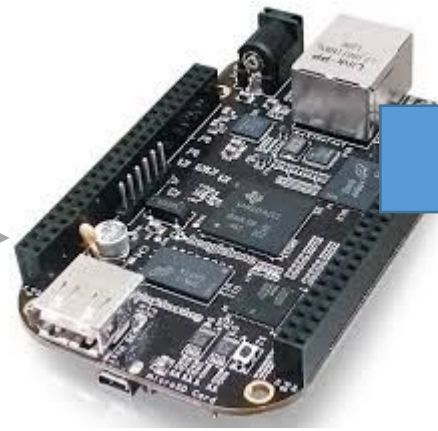


Development Configurations - Removable Storage Setup

Build & Host Device

- Secondary Bootloader
- Kernel
- Root Filesystem

Target Device



Bootloader

Development Configurations - Standalone Setup

- Bootloader
- Kernel
- Full Root Filesystem
- Native Development Environment

The Embedded Linux Software Eco System

Build System for Linux Development

Integrated Development Environment

Toolchain

Board Support Package

Test Environment

Application Development Support

Programming Flashing

Debug Environment

Compiler, Binary Utilities, Libraries

Boot Loader, Kernel,

Root File System

Simulation/ Emulation Setup

Target Hardware

The Toolchain

"Building a [...] cross-toolchain for use in embedded systems development [is] a scary prospect, requiring iron will, days if not weeks of effort, lots of Unix and Gnu lore, and sometimes willingness to take dodgy shortcuts."

-Dan Kegel, the main author of Crosstool

Toolchain

Compiler	• Cross Compiler & Cross-Native Compiler • Linux Compiler & Embedded C Compiler
Binutils	• Assembler & Linker, Profiler & Coverage • Object Copy & Object Dump
Debugger	• Kernel & Application Debugger • Tracing Analysis
Programmer	• Flash programming • Secondary Storage Setup & Removable Storage Setup
Library	• C- Runtime Libraries: Glibc & uCLibC • Linux Kernel Headers

Toolchain: 1. Compiler

- **Build System -** The system on which the **compiler is compiled**
- **Host System -** The system on which the **compiler is executed**
- **Target System -** The system for which the **compiler generates the executable** and the system on which executable runs

- Native Compiler

X86/Linux OS	X86/Linux OS	X86/Linux OS
Build System	Host System	Target System

- Cross Compiler

X86/Linux OS	X86/LinuxOS	ARM
Build System	Host System	Target System

- Cross-Native Compiler

X86/Linux OS	ARM	ARM
Build System	Host System	Target System

Compiler

- Cross Compiler
  - The cross compiler is generated by compiling the compiler source code (gcc) for a target platform (eg. ARM) using the native compiler of the build system
    - Eg: Native gcc of x86 platform is used to compile the gcc for arm-linux, setting the target to be arm architecture
  - The cross-compiler executes on the host system, but generates the executable which can be run only on the target platform
    - Eg: arm-linaro-linux-gnu-gcc –o hello hello.c
  - Generated executable will execute on the target platform only
  - Usage
    - Required to compile the Bootloader, Kernel and other drivers, required for boot process of the target board

Compiler

- Cross-Native Compiler
  - This compiler is compiled on the build system, using the **cross compiler of the target**
  - Generates an executable for the target platform, which is the native compiler for the target
  - Executes on the target platform and is used to generate executables on the target platform, for the target platform
  - Here, the host platform and the target platform remain the same
  - Usage
    - Included into the Root File System of the target platform and can be used directly on the board
    - Used to compile applications on the target platform

The Compiler (ARM-LINUX-GCC)

- ARM-GNU Compiler Collection
  - Contains the program to understand the grammar of a language and generate the object file specific to a particular hardware platforms
- Native Compiler and Cross Compiler
- Pre-built or Sources
- Popular Compiler Distributors for ARM
  - Linaro - https://releases.linaro.org/14.04/components/toolchain/binaries/
  - CodeSoucery - https://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/overview
  - Crosstool-ng - http://crosstool-ng.org/download/crosstool-ng/
  - Musl-cross - https://bitbucket.org/GregorR/musl-cross
- Application Binary Interface (ABI)
  - Allows portability across binaries generated by various toolchain vendors

Toolchain: 2. Binutils

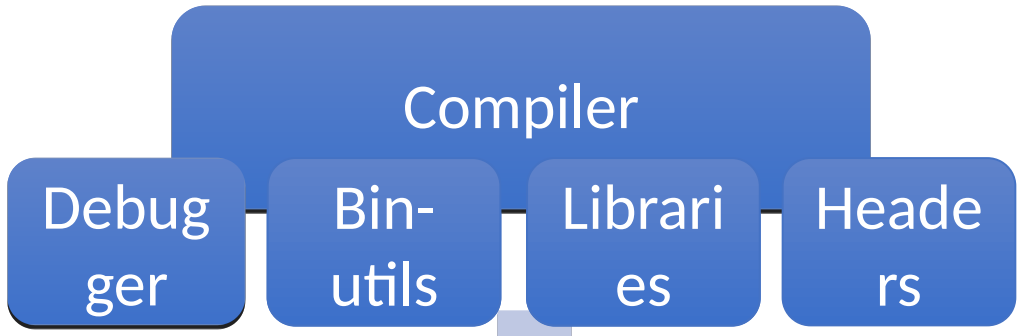
- Source: https://www.gnu.org/software/binutils/
- Latest Stable Version: 2.30

Utility	Use
as	GNU assembler
ld	GNU linker
gasp	GNU assembler pre-processor
nm	Lists the symbols in an object file
objcopy	Copies and translates object files
objdump	Displays information about the content of object files
ranlib	Generates an index to the content of an archive
readelf	Displays information about an ELF format object file
size	Lists the sizes of sections within an object file
ar	Creates and manipulates archive content

Toolchains: Binutils

- Use Cases
  - If binutils is to be used to generate executables for the target platform
    - The binutils are to be compiled with the native compiler of the host, specifying the appropriate target
  - If binutils is to be used in the target platform,
    - It should be compiled using the cross compiler of the target and shipped with the rootfs of the target

Creating the Build Environment



- Use pre-built compiler (or)
- Compile the compiler, debugger and binary utilities using native compiler and configurations

Build the toolchain which will be used to build other components for the board support package. This will execute on the host system

Use the Cross Tool Chain to build components that will run on the Target Device. These components will be loaded in the Root File System

CN Compiler

Bin-utils

Libraries

- Compile the libraries in the presence of its headers and create the images, using cross compiler

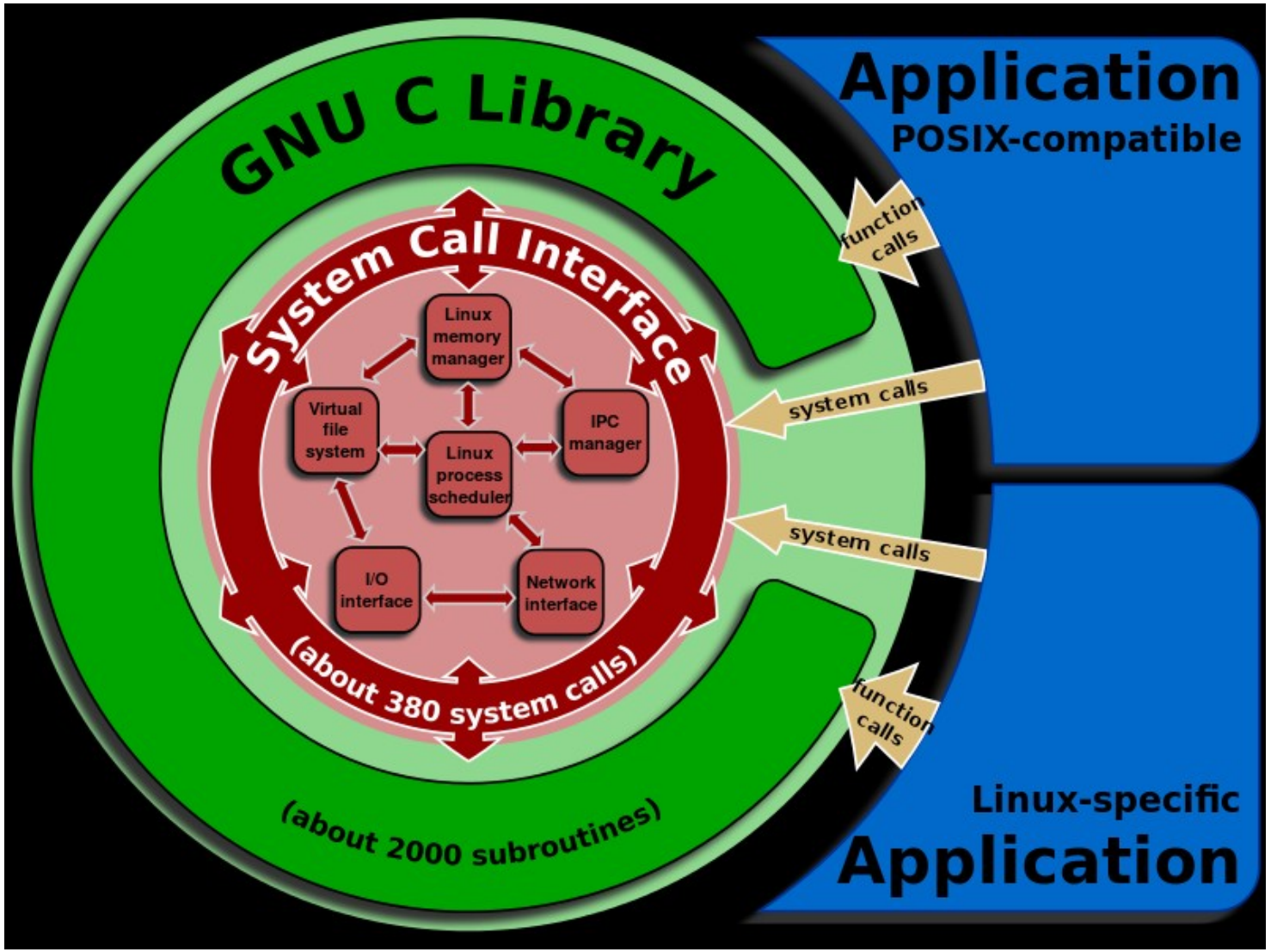
Debugger

- Compile the debugger

Environment

- Set the environment for development

Toolchains: 3. Libraries



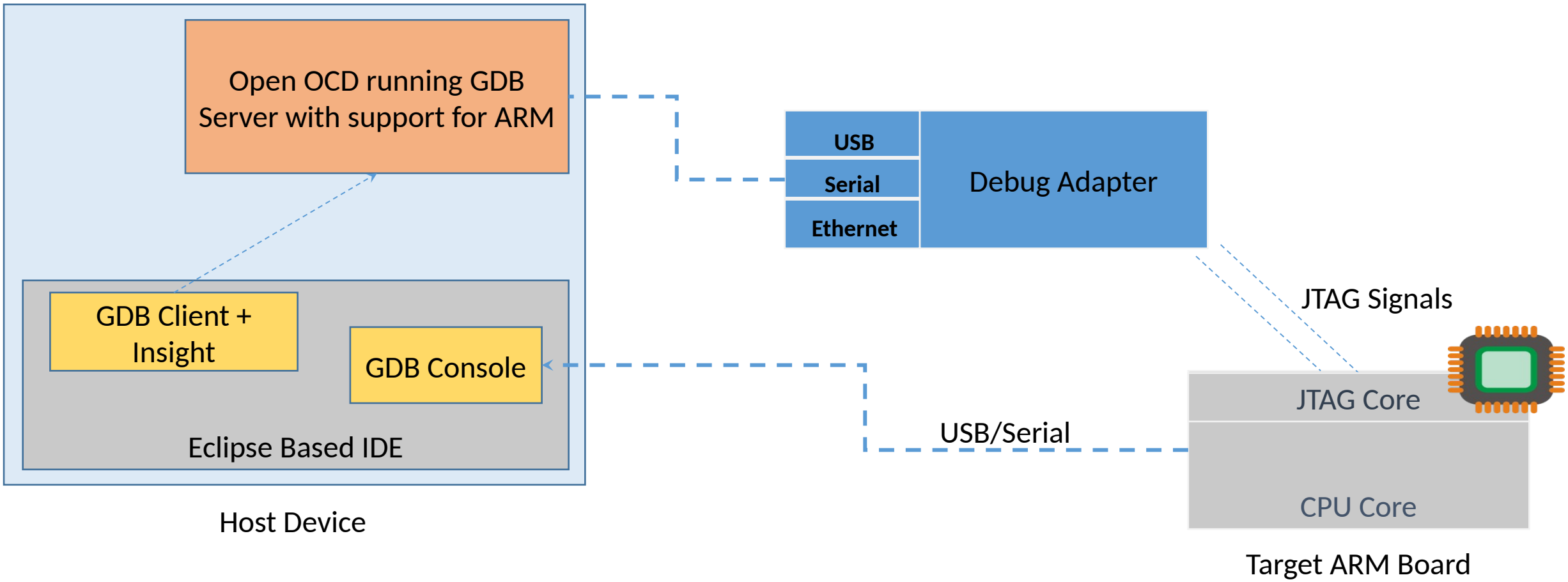
Toolchains: 3. Libraries

- GNU C Library (glibC)
  - Source - https://www.gnu.org/software/libc/
  - Current Stable Version: 2.24
  - Portable, Extensive, High Performance
  - Supports ISO C99, POSIX 1.c, 1.j and 1.d, UNIX 98
  - Good for Host Systems, but falls short in Embedded Systems owing to large size. Minimum size is 2MB
  - Optimized for performance and not memory…
- uCLibC
  - Small C Standard Library for Linux Kernel based Embedded Systems
  - Free and Open Source, complying to LGPL library
  - Light Weight supporting MMU-Less Linux
  - Supports number of hardware platforms
  - Source - https://uclibc.org/
  - Current Stable Version: 0.9.33.2

Toolchains: Headers

- Required to compile the Library and User Applications for specific kernels
- Sanitized Headers
  - Post 2.6 version of kernel, a sanitized version of the headers is generated for tool chain and application usage. This differs from the kernel headers, which contain inline assembly code, compromising the kernel, if used in application code

Debugger – Target-in-line



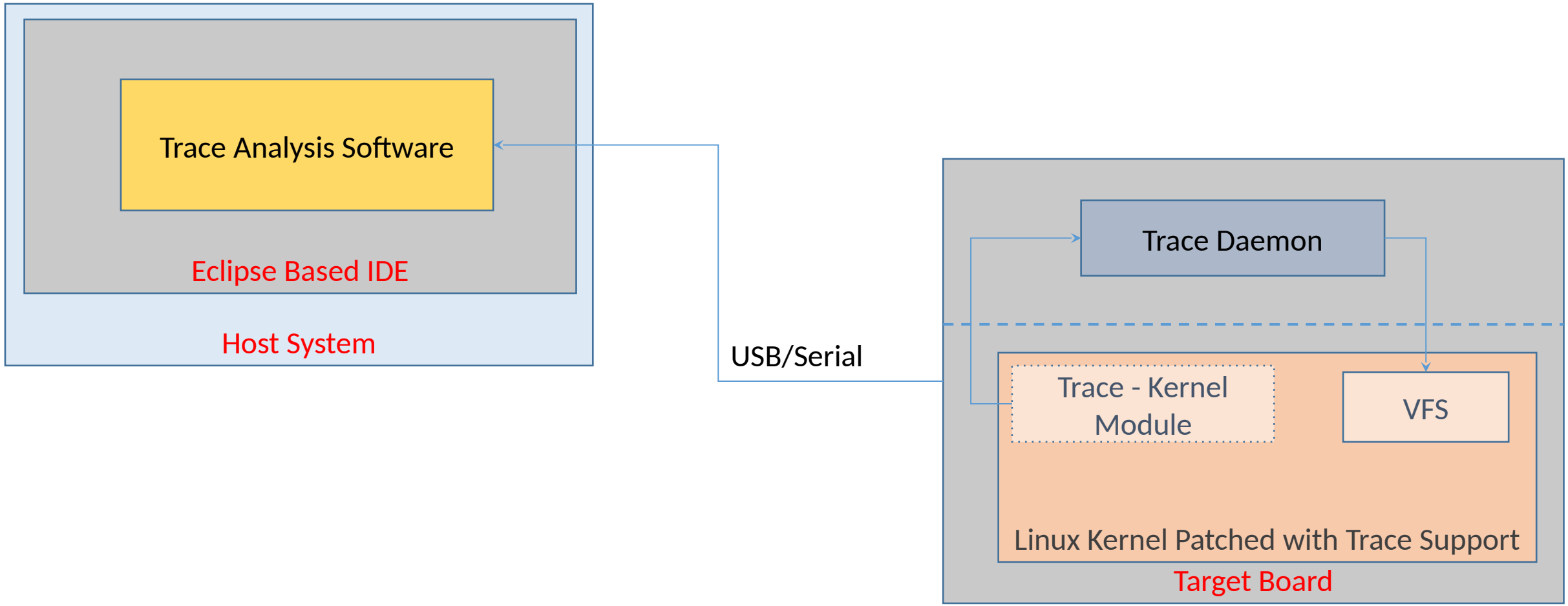
Debugger – External

- Target-in-line Debugging
  - Source level debugging of Linux Kernel & Bootloader
  - Support for debugging of dynamic modules
  - Integrates gdb with OpenOCD on the host, allowing the user to debug a remote target through a JTAG adapter
  - OpenOCD is supported by ARM architectures
  - Configuration scripts for CPU core and board should be developed for Open OCD to be used
  - Debug support for peripherals on the core and peripherals on the board may be architected as part of the design

Debugger – Internal

- Application Debugging
  - User space application may be debugged using the gdb client server model, running on the target board
  - Native debugger (gdb) for the ARM architecture is required to perform the debug
  - Added into the Root File System of the target platform

Debugger – Linux Trace Toolkit



Debugger

- Linux Trace Toolkit
  - Includes kernel components and user-level tools required to view traces
    - Linux Kernel patched with trace support
    - Linux Kernel module which collects trace events
    - Trace Daemon that receives the trace events from the linux kernel and stores it into the Filesystem
    - Trace analysis tool that can be run on a remote system (host) for analysis
  - Trace support should be integrated with the kernel as part of project

The Board Support Package

Board Support Package Development

Boot loader

- Uboot, Redboot, GRUB
- Boot Architecture - BIOS -> Level 1 Bootloader -> Level 2 Bootloader
- Peripheral Driver support for booting

Linux Kernel

- Port Code Development & Porting to target platform
- Device Tree Sources for Core and Board
- Interrupt Management

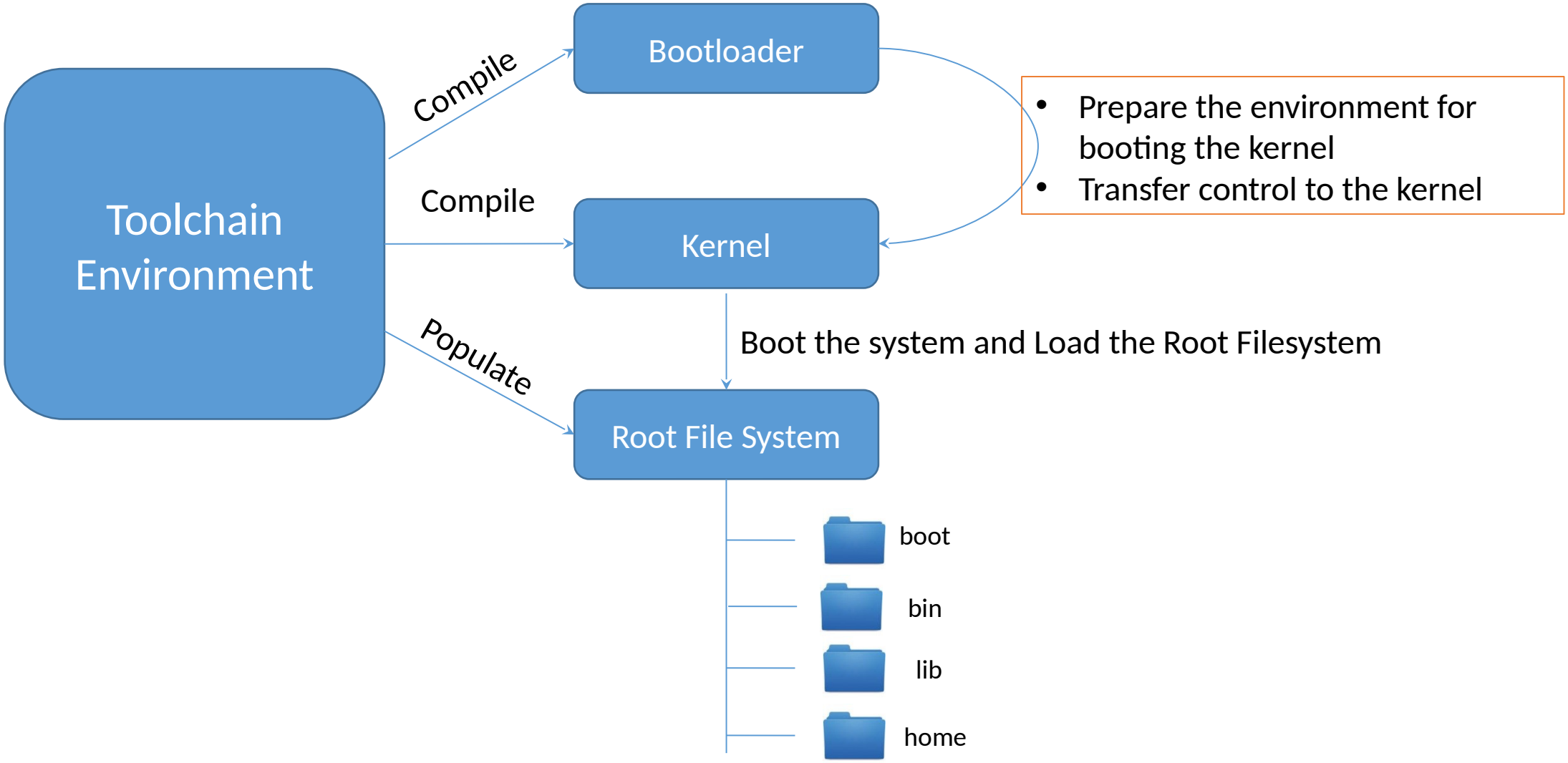
Peripheral Drivers

- Peripherals internal and external to the core
- Multicore support
- Networking Stacks

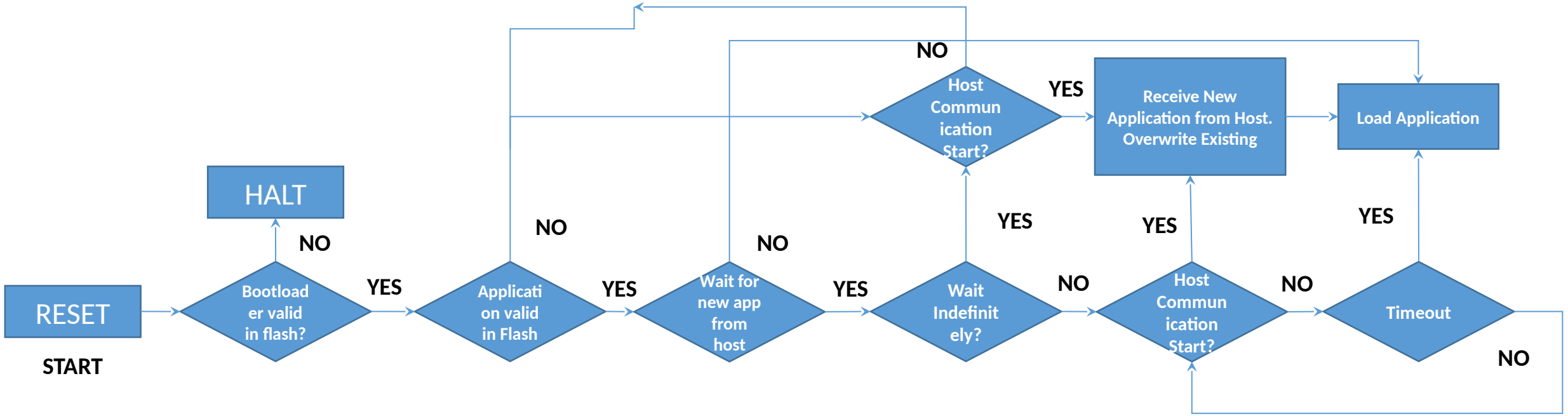
Root File System

- Creating and populating the RFS
- uCLibc/glibc
- Utilities for development (Busybox) & Init Process for user space

Building the Sources



The Bootloader



The Bootloader - GNU Grand Unified Bootloader (GRUB)

- Developed by Free Software Foundation
- Provides Multi-boot facility to load multiple operating systems
- Highly configurable through bash-like command line interface
- Supports almost all filesystem types
- Two Stage Loading
- Ref: https://www.gnu.org/software/grub/
- Supports Ethernet, USB and SATA based loading of the kernel

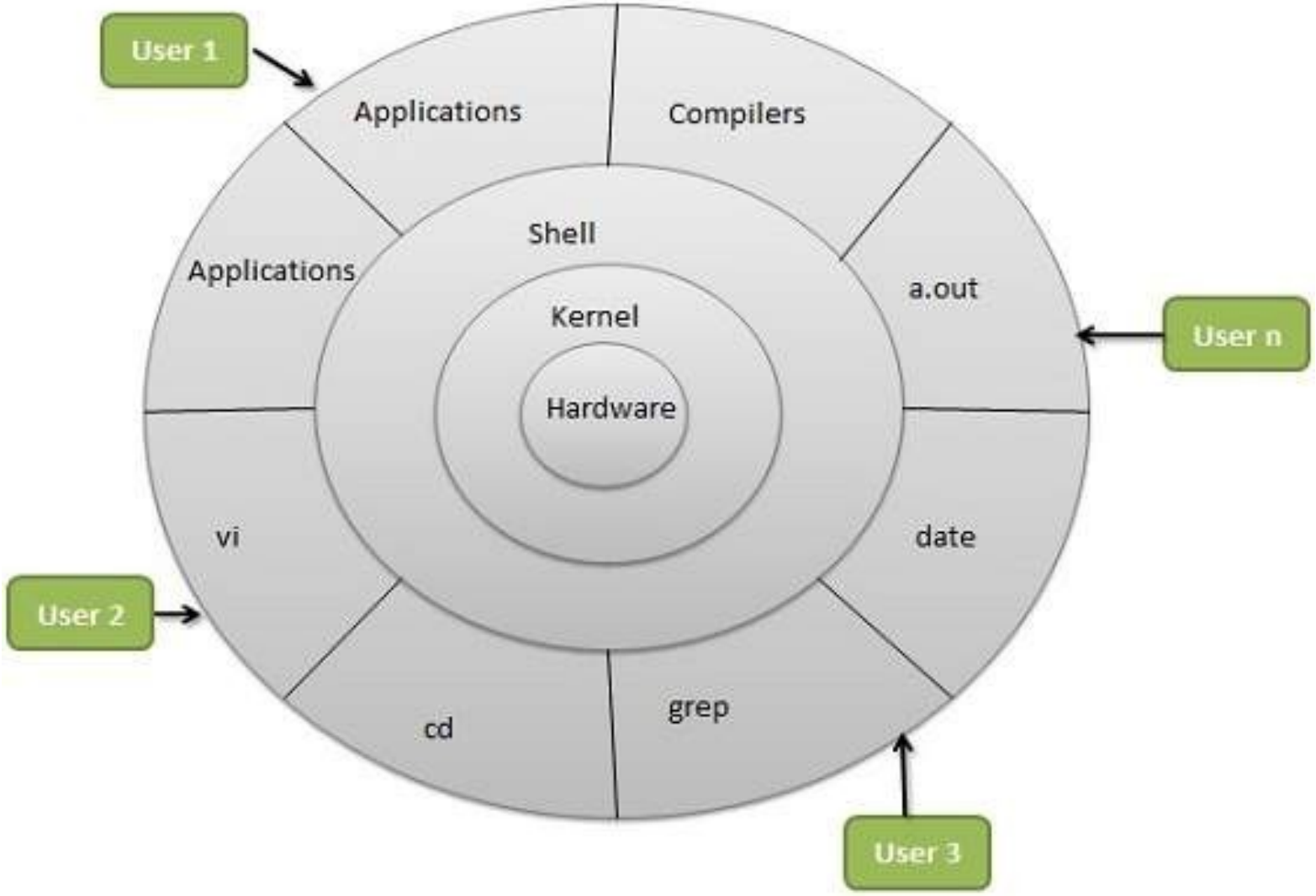
The Bootloader - uboot

- Multi-platform, open-source, universal boot-loader with comprehensive support for loading and managing boot images, such as the Linux kernel
- Features
  - **Network download**: TFTP, BOOTP, DHCP, NFS
  - **Serial download**: s-record, binary (via Kermit)
  - **Flash management**: copy, erase, protect, cramfs, jffs2
  - **Flash Types**: CFI NOR-Flash, NAND-Flash
  - **Memory utilities**: copy, dump. crc, check, mtest
  - **Mass Storage Devices**: IDE, SATA, USB
  - **Boot from disk**: raw block, ext2, fat, reiserfs
  - **Interactive shell**: choice of simple or "busybox" shell with many scripting features
- Powerful Commandline Interface via Serial Port
- Ref: https://sourceforge.net/projects/uboot/

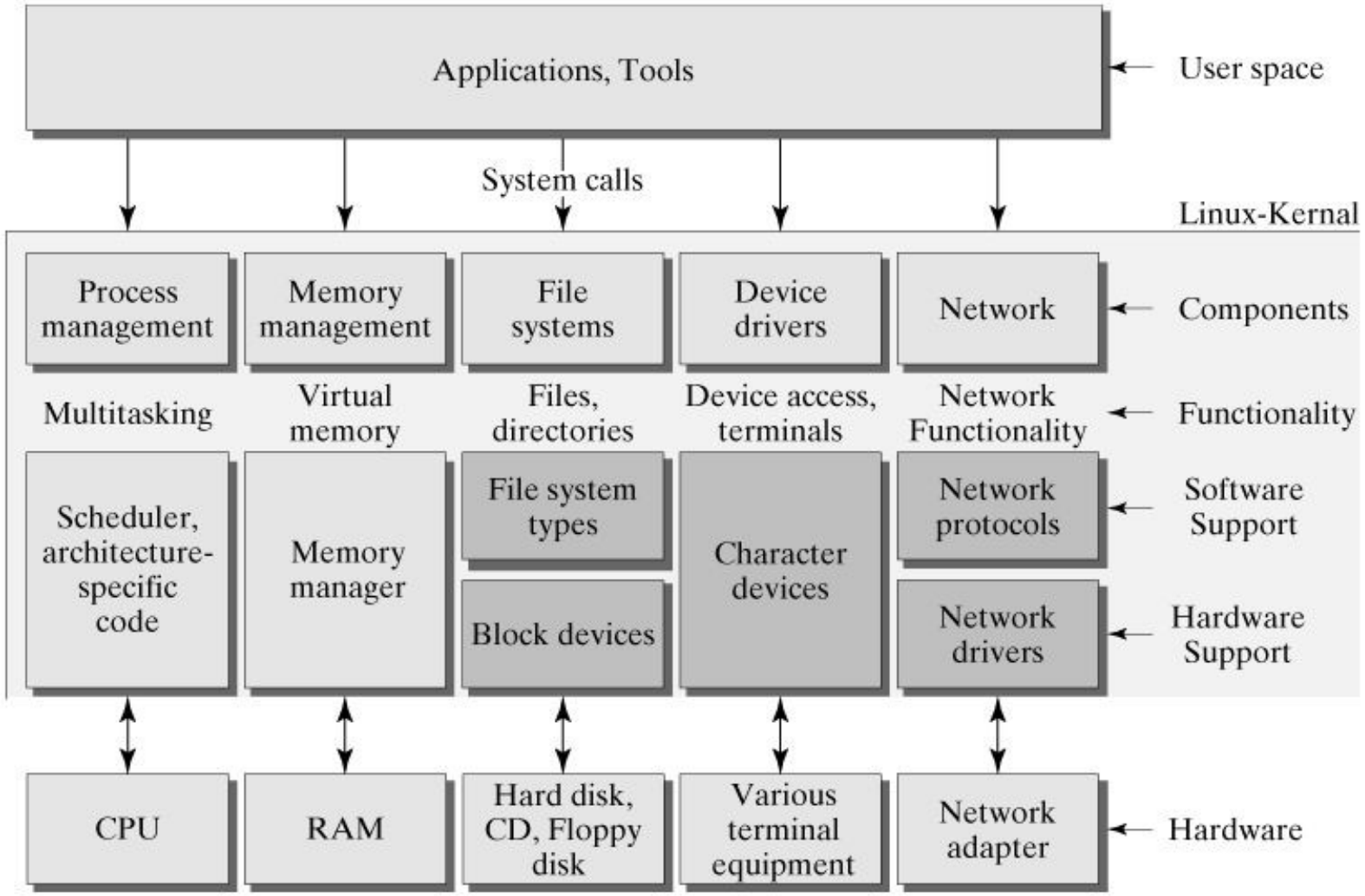
The Linux Kernel…

- Contains the Operating System code and low level drivers/modules to interact with hardware and provide services to the user….
- Latest Stable Version is 5.9.6
- Ref: www.kernel.org

The Linux Operating System Architecture



The Linux Kernel Architecture



The Rootfilesystems

- Contains the applications, services, libraries and other executables that a user may require to run his applications.
- Created with a specific filesystem type – ext2,3,4, fat32, squash, flashfs, etc
- Loaded by the kernel for user applications to proceed
- All libraries and native tools for further build are stored in the rootfilesystem

Embedded Systems – Board Development Environment

- Build System
  - One on which the toolchain is built
- Host Device
  - Any device capable of performing cross platform development
  - Linux Workstation, Unix System and Windows
  - Build the programs and create the object file, which will be executed on the target device
- Target Device
  - The device onto which developed software has to be flashed and executed
  - X86, ARM, PowerPC, MIPS processor etc
- Development Configurations
  - Linked Setup
  - Removable Storage Setup
  - Standalone Setup

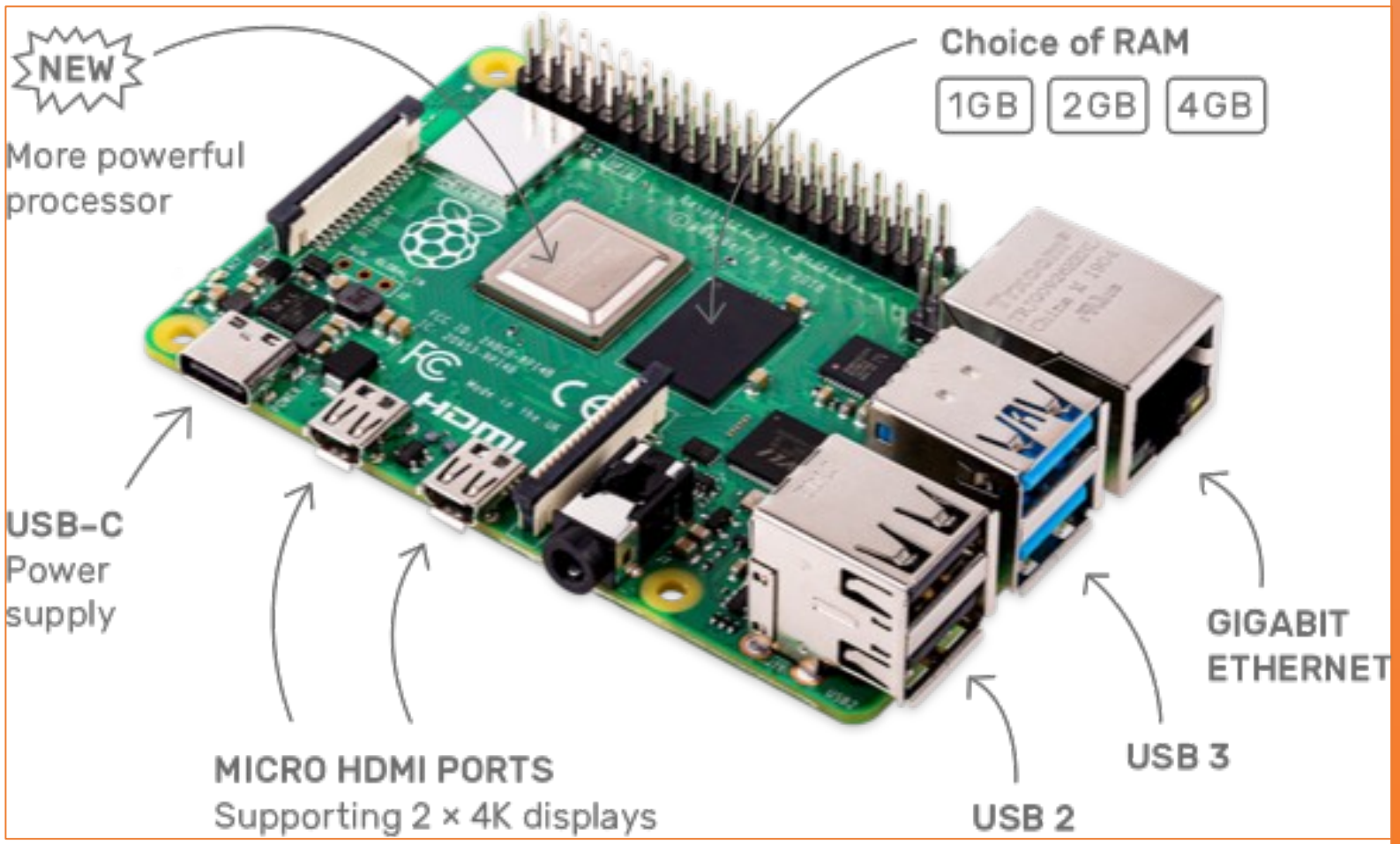
Pre-requisites to get an embedded board running Linux

- Tool Chain – Set of software tools needed to build computer software
  - Compiler
    - **Native**– Compiled on one architecture and runs on the same architecture (gcc for x86)
    - **Cross Toolchain**– Compiled on one architecture, runs on that architecture and builds code for another architecture (arm-linux-gcc)
    - **Cross Native Toolchain** – Compiled on one architecture, runs on a target architecture and builds code for the target architecture
  - Library: Glibc, uClibc, newlibC
  - Headers: For Prototype declarations and resolving
  - Bin-utils: Linker, Assembler, Object File Interpreters (objcopy, objdump), etc…
  - Debugger: gdb
- Bootloader
  - Grub, uboot, redboot,
- Kernel
  - Requires platform specific port code and configurations
  - Sometimes, Board to platform bindings are also required
- Root Filesystem

Thanks

The Raspberry Pi4

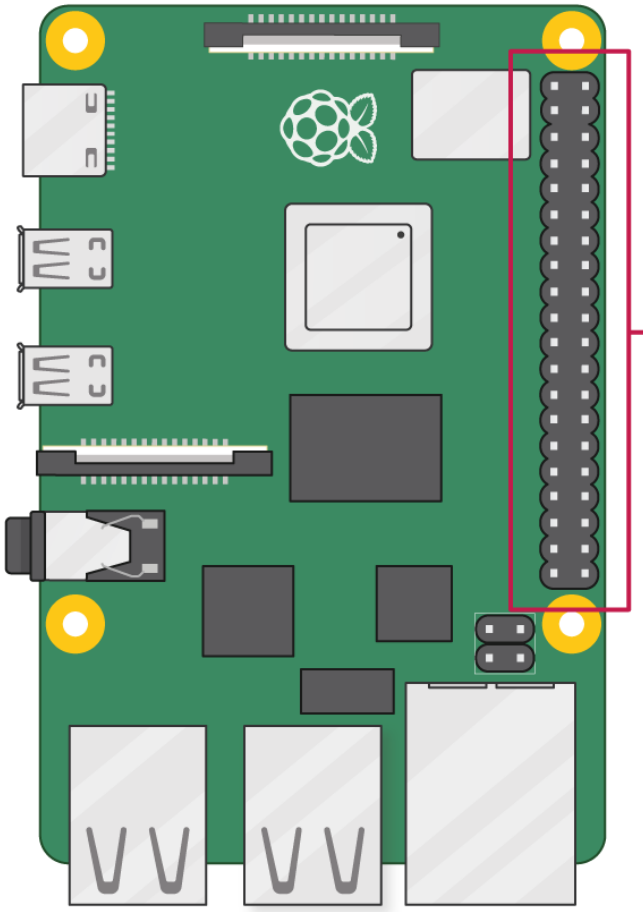
Raspberry Pi4



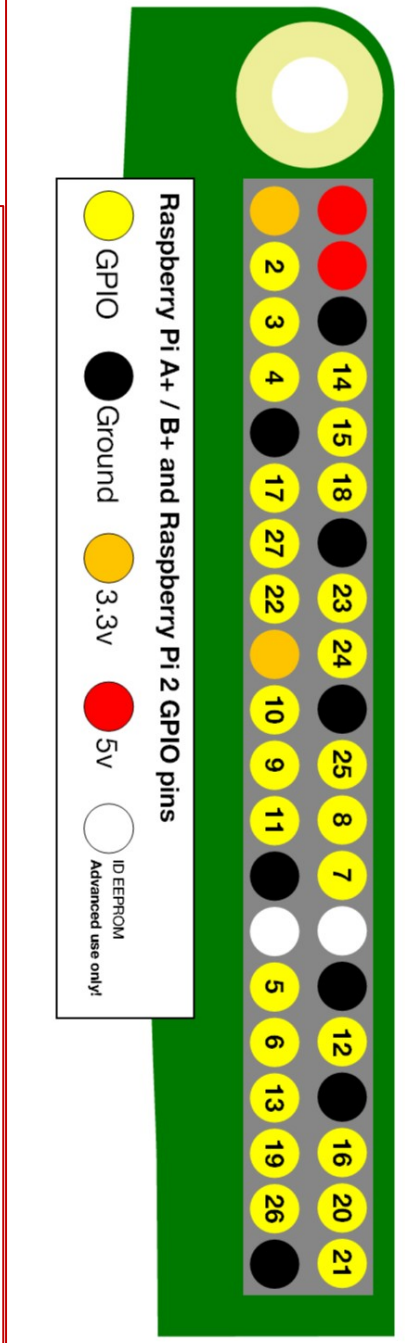
Tech Specs

Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
2GB LPDDR4-3200 SDRAM
2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE
Gigabit Ethernet
2 USB 3.0 ports; 2 USB 2.0 ports.
Raspberry Pi standard 40 pin GPIO header (fully backwards compatible with previous boards)
2 × micro-HDMI ports (up to 4kp60 supported)
OpenGL ES 3.0 graphics
Micro-SD card slot for loading operating system and data storage
5V DC via USB-C connector /GPIO header (minimum 3A*)
Power over Ethernet (PoE) enabled

GPIO Available



3V3 power	1	2	5V power
GPIO 2 (SDA)	3	4	5V power
GPIO 3 (SCL)	5	6	Ground
GPIO 4 (GPCLK0)	7	8	GPIO 14 (TXD)
Ground	9	10	GPIO 15 (RXD)
GPIO 17	11	12	GPIO 18 (PCM_CLK)
GPIO 27	13	14	Ground
GPIO 22	15	16	GPIO 23
3V3 power	17	18	GPIO 24
GPIO 10 (MOSI)	19	20	Ground
GPIO 9 (MISO)	21	22	GPIO 25
GPIO 11 (SCLK)	23	24	GPIO 8 (CE0)
Ground	25	26	GPIO 7 (CE1)
GPIO 0 (ID_SD)	27	28	GPIO 1 (ID_SC)
GPIO 5	29	30	Ground
GPIO 6	31	32	GPIO 12 (PWM0)
GPIO 13 (PWM1)	33	34	Ground
GPIO 19 (PCM_FS)	35	36	GPIO 16
GPIO 26	37	38	GPIO 20 (PCM_DIN)
Ground	39	40	GPIO 21 (PCM_DOUT)



OFF TO DEVELOPMENT

- Create the SD Card Image for the Raspberry Pi4 Board
  - Download the SD Card image from your FTP – It is stored by the name Rpi.img
  - This image has to be flashed onto the SD Card before booting the Pi
  - Insert the SD card into the SD card Reader
  - Unmout the SD Card Reader: **sudo umount /media/<check the name of the card>**
  - Transfer the image to the SD card: **dd if=Rpi.img of=/dev/sdb bs=4M**. This will take about 40 mins.
  - Once completed, insert the SD card into the board. Follow the notes mentioned here and start programming
- Creating the Cross Development Environment
  - Download the folder **DeviceDrivers.zip** from FTP
  - Unzip the folder into your home directory – **unzip DeviceDrivers.zip**
  - Change Directory to DeviceDrivers
  - Follow instructions in the **install-Rpi.txt** to install the Cross Development Environment
- NOTES
  - Default IP on the Raspberry Pi4 is set to: 192.168.0.10
  - Set your PC to 192.168.0.2 using the following command: **sudo ip addr add 192.168.0.2/24 dev <iface name>**
  - Connect to the Board by executing the following on the terminal: ssh –X pi@192.168.0.10 . Password is cdac@123
  - Start Your Programs in DD-Progs Folder.

Toolchain References

- Linux From Scratch project (*http://trac.cross-lfs.org*)

Debug Options

- Serial Interface
  - UART/USB
- Network Interface
  - Ethernet Protocols
- Debugging Hardware
  - JTAG

Buildroot

Open Embedded and Yocto Project

Device Tree Sources…