**Group 1**

RENESAS

**OPEN SOURCE SOFTWARE REPORT**

**Table of Contents**

[1. Introduction: 1](#_Toc465352307)

[1.1. Linux: 1](#_Toc465352308)

[1.1.1. Definition: 1](#_Toc465352309)

[1.1.2. History: 1](#_Toc465352310)

[1.2. Kernel: 1](#_Toc465352311)

[2. Linux kernel: 2](#_Toc465352312)

[2.1. Description: 2](#_Toc465352313)

[2.2. Kernel functions: 3](#_Toc465352314)

[2.2.1. Resource allocation: 3](#_Toc465352315)

[2.2.2. Process Management: 3](#_Toc465352316)

[2.2.3. Memory Management: 3](#_Toc465352317)

[2.2.4. I/O Device Management: 4](#_Toc465352318)

[2.2.5. Inter-Process Communication: 4](#_Toc465352319)

[2.2.6. Scheduling: 4](#_Toc465352320)

[2.2.7. System Calls and Interrupt Handling: 4](#_Toc465352321)

[2.2.8. Security or Protection Management 5](#_Toc465352322)

[2.3. Relationship between Linux kernels, hardware, application. 5](#_Toc465352323)

[2.4. Memory-mapped I/O and port-mapped I/O: 6](#_Toc465352324)

[2.4.1. Memory-mapped I/O: 6](#_Toc465352325)

[2.4.2. Port-mapped I/O: 7](#_Toc465352326)

[2.5. System call: 8](#_Toc465352327)

[2.6. Interact between kernel mode and user: 9](#_Toc465352328)

[3. Embedded Linux: 10](#_Toc465352329)

[3.1. Overview: 10](#_Toc465352330)

[3.1.1. Definition: 10](#_Toc465352331)

[3.1.2. Advantages of Embedded Linux: 10](#_Toc465352332)

[3.2. Boot process: 13](#_Toc465352333)

[3.2.1. Definition: 13](#_Toc465352334)

[3.2.2. Boot process of embedded Linux: 13](#_Toc465352335)

[4. Toolchain: 15](#_Toc465352336)

[4.1. Native toolchain: 16](#_Toc465352337)

[4.2. Cross-compiling toolchain: 16](#_Toc465352338)

[4.3. Native toolchain vs Cross-compiling toolchain: 16](#_Toc465352339)

# Introduction:

## Linux:

### Definition:

Linux, an implementation of UNIX, is a computer operating system assembled under the model of free and open-source software development and distribution.

Linux distribution is an operating system made from a software collection based on the Linux kernel and a package management system.

A typical Linux distribution includes a Linux kernel, GNU tools and libraries, additional software, documentation, a window system, a window manager, and a desktop environment.

Some popular distributions: Ubuntu, Debian, Gentoo, etc.

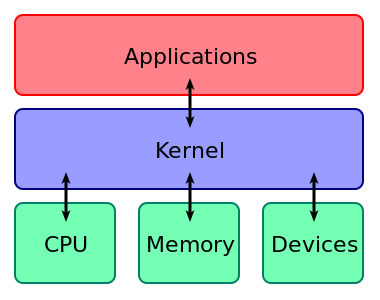
### History:

* Derived from UNIX.
* Released by Linus Torvalds in 1991.
* Has been using throughout computing, such as server systems, embedded systems, mobile devices, phones, and supercomputers.
* Nowadays, in progress of development by Torvalds.

## Kernel:

Kernel is a computer program constituting the central core of a computer’s operating system. It has complete control over everything that occurs in the system.

Kernel is the heart of the system. It manages the communication between the underlying hardware and the peripherals.



**Figure 1.1: A kernel connects the application software to the hardware of a computer.**

# Linux kernel:

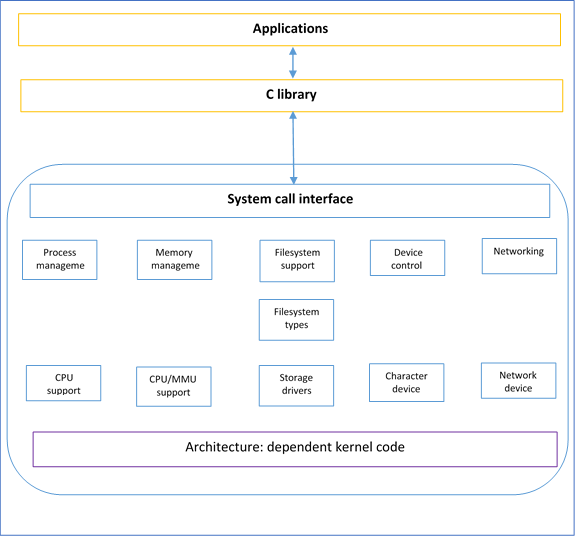
Linux kernel is a Unix-like computer operating system kernel. The Linux OS is based on it and deployed on both traditional computer systems like PC and servers, usually in the form of Linux distributions, and on various embedded systems, for example: routers, wireless access points, smart TVs, and so on.

Linux kernel main roles are:

* + Managing all the hardware resources.
  + Providing a set of portable, architecture and hardware independent APIs.
  + Handling concurrent access and usage.

## Description:

Linux Kernel architecture to be divided into two levels – User Space and Kernel Space.



**Figure 2.1: Linux kernel’s structure**

* **User Space:** This is where the user applications are executed. There is also the GNU C library. This provides the system call interface that connects to the kernel and provides the mechanism to translate between the user-space application and the kernel.
* **Kernel Space:** Here, the Linux kernel exists which can be divided into three levels. At the top is the system call interface, which implements the basic functions such as ***read*** and ***write***. Below the system call interface is the kernel code, which can be more accurately defined as the architecture-independent kernel code. This code is common to all of the processor architectures supported by Linux. Below this is the architecture-dependent code, which forms what is more commonly called a BSP (Board Support Package). This code serves as the processor and platform-specific code for the given architecture.

## Kernel functions:

### Resource allocation:

The kernel's primary function is to manage the computer's resources and allow other programs to run and use these resources. These resources are: CPU, Memory and I/O devices.

### Process Management:

A process defines which memory portions the application can access. The main task of a kernel is to allow the execution of applications and support them with features such as hardware abstraction.

To run an application, a kernel first set up an address space for the application, then loads the file containing the application's code into memory, then set up a stack for the program and branches to a given location inside the program, thus finally starting its execution.

### Memory Management:

The kernel has full access to the system's memory. It allows processes to safely access this memory as they require it. Virtual addressing helps kernel to create virtual partitions of memory in two disjointed areas, one is reserved for the kernel (kernel space) and the other for the applications (user space).

### I/O Device Management:

To perform useful functions, processes need access to the peripherals connected to the computer, which are controlled by the kernel through Device Drivers. A device driver is a computer program that enables the operating system to interact with a hardware device. It provides the operating system with information of how to control and communicate with a certain piece of hardware.

A kernel maintains a list of available devices. A device manager first performs a scan on different hardware buses, such as Peripheral Component Interconnect (PCI) or Universal Serial Bus (USB), to detect installed devices, then searches for the appropriate drivers. The kernel provides the I/O to allow drivers to physically access their devices through some port or memory location.

### Inter-Process Communication:

Kernel provides methods for Synchronization and Communication between processes called Inter- Process Communication (IPC). There are various approaches of IPC say, semaphore, shared memory, message queue, pipe (or named FIFO), etc.

### Scheduling:

In a Multitasking system, the kernel will give every program a slice of time and switch from process to process so quickly that it will appear to the user as if these processes were being executed simultaneously. The kernel uses Scheduling Algorithms to determine which process is running next and how much time it will be given. The algorithm sets priority among the processes.

### System Calls and Interrupt Handling:

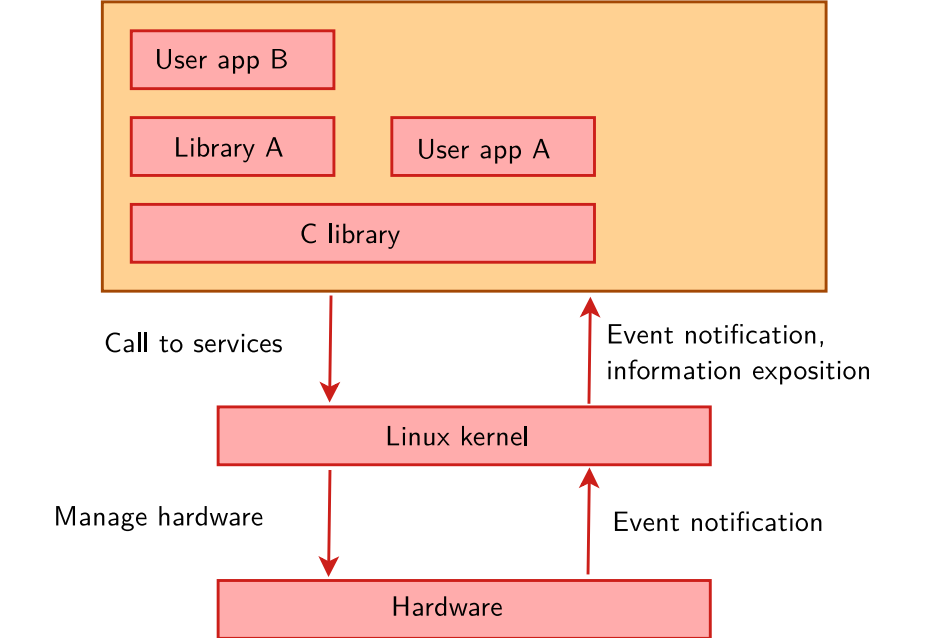
A system call is a mechanism that is used by the application program to request a service from the operating system. System calls include close, open, read, wait and write. To access the services provided by the kernel we need to invoke the related kernel functions. Most kernels provide a C Library or an API, which in turn invokes the related kernel functions.

There are few methods by which the respective kernel function can be invoked- using Software- Simulated Interrupt, or using a Gate Call, or by using a Special System Call Instruction and by using a Memory- based Queue.

### Security or Protection Management

Kernel also provides protection from faults (error control) and from malicious behaviors (Security). One approach towards this can be Language based protection system, in which the kernel will only allow code to execute which has been produced by a trusted language compiler.

## Relationship between Linux kernels, hardware, application.



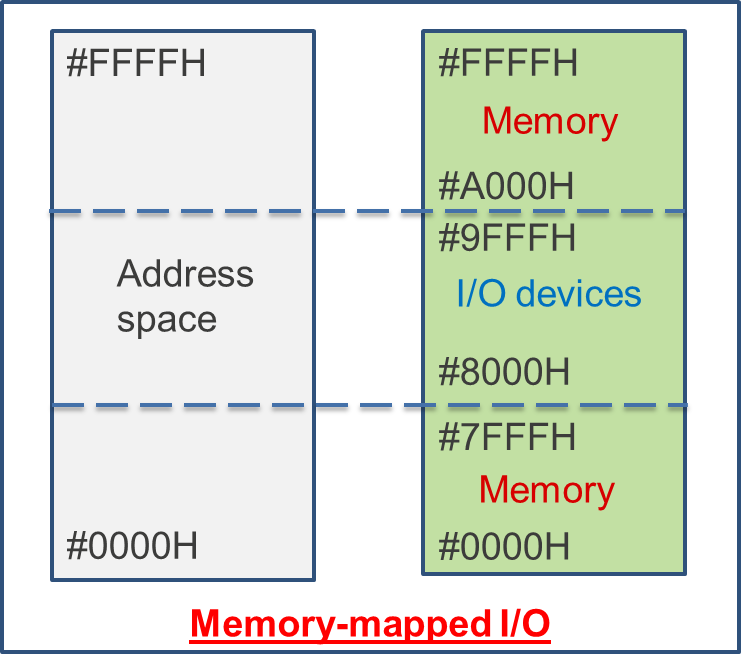
**Figure 2.2: Relationship between kernel, hardware and application**

* **Bootloader:**
* Started by the hardware, responsible for basic initialization, loading and executing the kernel.
* **Linux Kernel:**
* Contains the process and memory management, network stack, device drivers and provides services to user space applications.
* Manage all the hardware resources: CPU, memory, I/O.
* Provides the well-known standard C API to ease application development.
* **C library:**
* The interface between the kernel and the user space applications.
* **Libraries and applications:**
* Third-party or in-house.

## Memory-mapped I/O and port-mapped I/O:

### Memory-mapped I/O:

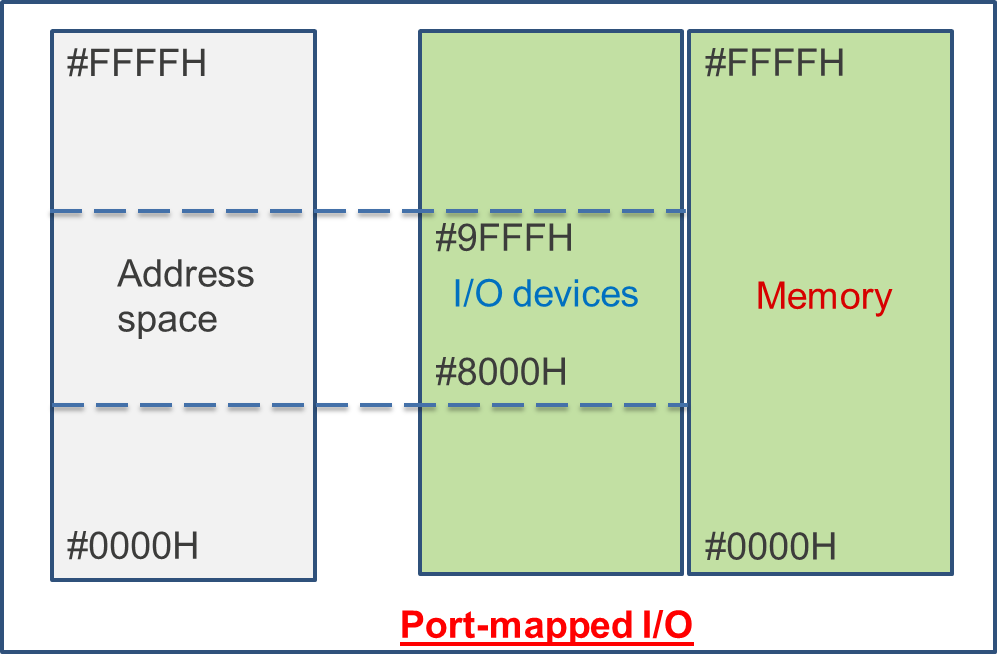
I/O devices are mapped into the system memory map along with RAM and ROM. To access a hardware device, simply read or write to those “special” addresses using the normal memory access instructions



**Figure 2.3: Memory-mapped diagram**

* **Advantage:**
* Every instruction which can access memory can be used to manipulate an I/O device.
* Faster, easy to build, consume less power, smaller size.
* **Disadvantage:**
* The entire address bus must be fully decoded for every device. This increases the cost of adding hardware to the machine. For example, a machine with a 32-bit address bus would require logic gates to resolve the state of all 32 address lines to properly decode the specific address of any device.

### Port-mapped I/O:



**Figure 2.4:** **Port-mapped diagram**

I/O devices are mapped into a separate address space. This is usually accomplished by having a different set of signal lines to indicate a memory access versus a port access. The address lines are usually shared between the two address spaces, but less of them are used for accessing ports.

For example: the standard PC which uses 16 bits of port address space, but 32 bits of memory address space.

* **Advantage:**
* Less logic is needed to decode a discrete address and therefore less cost to add hardware devices to a machine
* **Disadvantage:**
* More instructions are required to accomplish the same task. For example: to add a constant to a port-mapped device register would require three instructions: read the port to a CPU register, add the constant to the CPU register, and write the result back to the port.
* **Compare memory-mapped I/O and port-mapped I/O:**

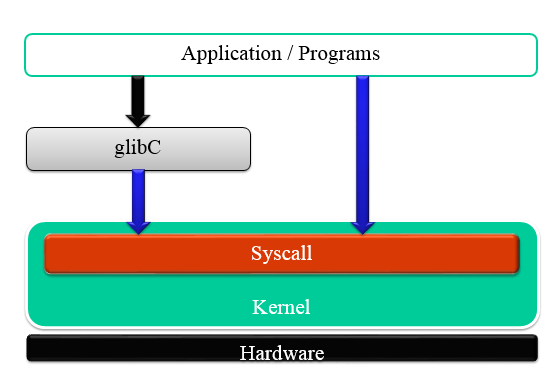
|  |  |
| --- | --- |
| Memory-mapped I/O | Port-mapped I/O |
| Using same address space and bus to address memory and I/O devices | Using different address space and bus to address memory and I/O devices |
| Uses regular instructions to access the I/O devices | Uses special instructions to access the I/O devices |

**Table 2‑1: Memory-mapped and port-mapped comparation**

## System call:

A system call is a mechanism that is used by the application program to request a service from the operation system. System calls provide an essential interface between a process and the operating system.

System calls include close, open, read, wait and write.



**Figure 2.5 System call**

Categories of system calls:

* **Process Control:**
* Load.
* Execute.
* End, abort.
* Create process.
* Terminate process.
* Get/set process attributes.
* Allocate, free memory.
* **File management:**
* Create file, delete file.
* Open, close.
* Read, write, reposition.
* Get/set file attributes.
* **Information Maintenance:**
* Get/set time or date.
* Get/set system data.
* Get/set process, file, or device attributes.
* **Communication:**
* Create, delete communication connection.
* Send, receive messages.
* Transfer status information.
* Attach or detach remote devices.

## Interact between kernel mode and user:

System calls in most Unix-like systems are processed in kernel mode, which is accomplished by changing the processor execution mode to a more privileged one, but no process context switch is necessary – although a privilege context switch does occur.

The hardware sees the world in terms of the execution mode according to the processor status register, and processes are an abstraction provided by the operating system. A system call does not generally require a context switch to another process; instead, it is processed in the context of whichever process invoked it.

In a multithreaded process system calls can be made from multiple threads. The handling of such calls is entirely dependent on the design of the specific operating system. The following list shows typical models followed by kernels:

* **Many-to-one model:** All system calls from any user thread in a process are handled by a single kernel-level thread. This model has a serious drawback – any blocking system call (like awaiting input from user) can freeze all the other threads. Also, since only one thread can access the kernel at a time, this model cannot utilize multiple cores of processor.
* **One-to-one model:** Every user thread gets attached to a distinct kernel-level thread during a system call. This model solves the above problem of blocking system calls. It is found in all major distribution of Linux, recent Windows and Solaris versions.
* **Many-to-many model:** In this model a pool of user threads is mapped to a pool of kernel threads. All system calls from a user thread pool are handled by the threads in their corresponding kernel thread pool
* **Hybrid model:** This model implements both many to many and one to one model depending upon choice made by the kernel. This is found in old versions of IRIX, HP-UX and Solaris.

# Embedded Linux:

## Overview:

### Definition:

Embedded Linux is an Operating systems based on the Linux kernel are used in embedded systems such as consumer electronics.



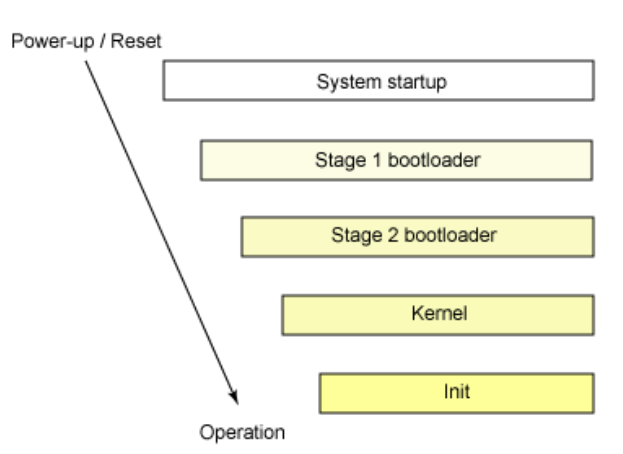
**Figure 3.1: Examples of Embedded Linux devices**

### Advantages of Embedded Linux:

* **Low cost:**
* Free software can be duplicated on as many devices as you want, free of charge.
* If embedded system uses only free software, you can reduce the cost of software licenses to zero. Even the development tools are free, unless you choose a commercial embedded Linux edition.
* Allows to have a higher budget for the hardware or to increase the company’s skill and knowledge.
* **Full control:**
* With open-source, you have the source code for all components in your system
* Allows unlimited modifications, changes, tuning, debugging, and optimization for an unlimited period of time.
* Without lock-in or dependency from a third-party vendor 🡪 To be true, non-open source components must be avoided when the system is designed and developed.
* Allows to have full control over the software part of your system.
* **Quality:**
* Many open-source components are widely used, on millions of systems.
* Usually higher quality than what an in-house development can produce, or even proprietary vendors.
* Of course, not all open-source components are of good quality, but most of the widely-used ones are.
* Allows to design your system with high-quality components at the foundations.
* **Eases testing of new features:**
* Open-source being freely available, it is easy to get a piece of software and evaluate it.
* Allows to easily study several options while making a choice.
* Much easier than purchasing and demonstration procedures needed with most proprietary products.
* Allows to easily explore new possibilities and solutions.
* **Community support:**
* Open-source software components are developed by communities of developers and users.
* This community can provide a high-quality support: you can directly contact the main developers of the component you are using. The likelihood of getting an answer doesn't depend what company you work for.
* Often better than traditional support, but one needs to understand how the community works to properly use the community support possibilities.
* Allows to speed up the resolution of problems when developing your system.
* **Taking part into the community:**
* Possibility of taking part into the development community of some of the components used in the embedded systems: bug reporting, test of new versions or features, patches that fix bugs or add new features, etc.
* Most of the time the open-source components are not the core value of the product: it’s the interest of everybody to contribute back.
* For the engineers: a very motivating way of being recognized outside the company, communication with others in the same field, opening of new possibilities, etc.
* For the Managers: motivation factor for engineers, allows the company to be.
* Recognized in the open-source community and therefore get support more easily and be more attractive to open-source developers.
* **Re-using components:**
* The key advantage of Linux and open-source in embedded systems is the ability to re-use components.
* The open-source ecosystem already provides many components for standard features, from hardware support to network protocols, going through multimedia, graphic, cryptographic libraries, etc.
* As soon as a hardware device, or a protocol, or a feature is wide-spread enough, high chance of having open-source components that support it.
* Allows to quickly design and develop complicated products, based on existing components.
* No-one should re-develop yet another operating system kernel, TCP/IP stack, USB stack or another graphical toolkit library.
* Allows to focus on the added value of your product.

## Boot process:

### Definition:



**Figure 3.2: Boot process of embedded Linux**

### Boot process of embedded Linux:

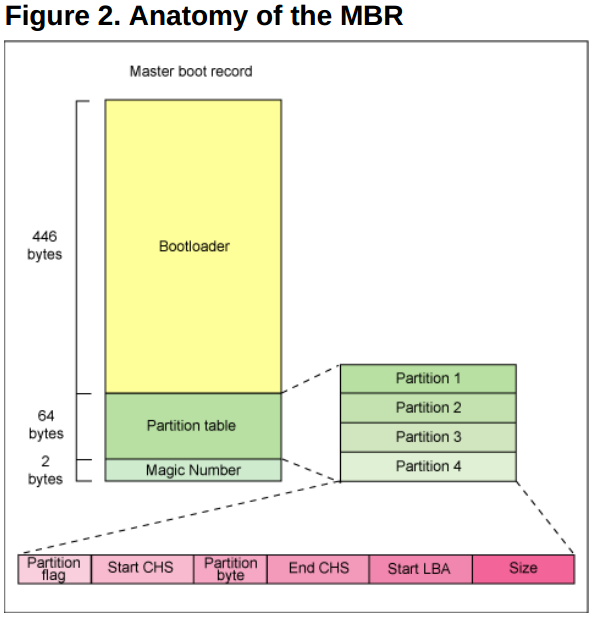
* **1st stage: system startup:**

The system startup stage depends on the hardware that Linux is being booted on. On an embedded platform, a bootstrap environment is used when the system is powered on, or reset.

Examples include U-Boot, RedBoot, and MicroMonitor from Lucent. Embedded platforms are commonly shipped with a boot monitor. These programs reside in special region of flash memory on the target hardware and provide the means to download a Linux kernel image into flash memory and subsequently execute it. In addition to having the ability to store and boot a Linux image, these boot monitors perform some level of system test and hardware initialization. In an embedded target, these boot monitors commonly cover both the first- and second-stage boot loaders.

* **2nd stage: stage 1 bootloader:**

The primary boot loader that resides in the MBR is a 512-byte image containing both program code and a small partition table (see Figure 2). The first 446 bytes are the primary boot loader, which contains both executable code and error message text. The next sixty-four bytes are the partition table, which contains a record for each of four partitions (sixteen bytes each). The MBR ends with two bytes that are defined as the magic number (0xAA55). The magic number serves as a validation check of the MBR.



**Figure 3.3: Anatomy of the MBR**

The job of the primary boot loader is to find and load the secondary boot loader (stage 2). It does this by looking through the partition table for an active partition. When it finds an active partition, it scans the remaining partitions in the table to ensure that they're all inactive. When this is verified, the active partition's boot record is read from the device into RAM and executed.

* **3rd stage: stage 2 bootloader:**

The secondary, or second-stage, boot loader could be more aptly called the kernel loader. The task at this stage is to load the Linux kernel and optional initial RAM disk

With the second-stage boot loader in memory, the file system is consulted, and the default kernel image and initial image are loaded into memory. With the images ready, the stage 2 boot loader invokes the kernel image.

* **4th stage: kernel:**

With the kernel image in memory and control given from the stage 2 boot loader, the kernel stage begins. The kernel image isn't so much an executable kernel, but a compressed kernel image. Typically this is a zImage (compressed image, less than 512KB) or a bzImage (big compressed image, greater than 512KB), that has been previously compressed with zlib.

At the head of this kernel image is a routine that does some minimal amount of hardware setup and then decompresses the kernel contained within the kernel image and places it into high memory. If an initial RAM disk image is present, this routine moves it into memory and notes it for later use. The routine then calls the kernel and the kernel boot begins.

* **5th stage: initialize:**

After the kernel is booted and initialized, the kernel starts the first user-space application. This is the first program invoked that is compiled with the standard C library. Prior to this point in the process, no standard C applications have been executed.

# Toolchain:

In software, a toolchain is the set of programming tools that is used to perform a more complex software development task or to create a software product, which is typically another computer program or a set of related programs.

A toolchain is the set of compiler, linker, librarian and any other tools you need to produce the executable (Shared libraries, etc.) for the target device. A debugger and/or IDE may also count as part of a toolchain.

## Native toolchain:

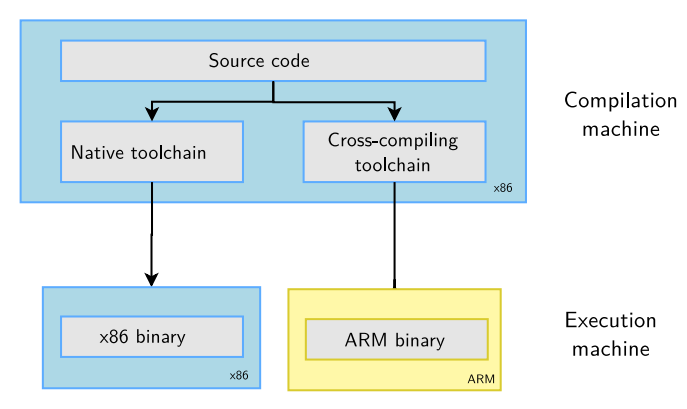
* The usual software development tools available on a GNU/Linux workstation is a native toolchain.
* This toolchain runs on workstation and generates code for the same kind workstation, usually x86.
* For embedded system development, it is usually impossible or not interesting to use a native toolchain:
* The target is too restricted in terms of storage and/or memory.
* The target is very slow compared to your workstation.
* You may not want to install all development tools on your target.

## Cross-compiling toolchain:

* Unlike native toolchain, cross-compiling toolchain is a set of software development tools available on any specific workstation platform and can generate for other target device platform.
* Cross-compiling toolchain use the cross-compiler, cross-compiler is a compiler where the target is different from the host. For example, a compiler that runs on a Windows 7 PC but generates code that runs on Android smartphone.

## Native toolchain vs Cross-compiling toolchain:

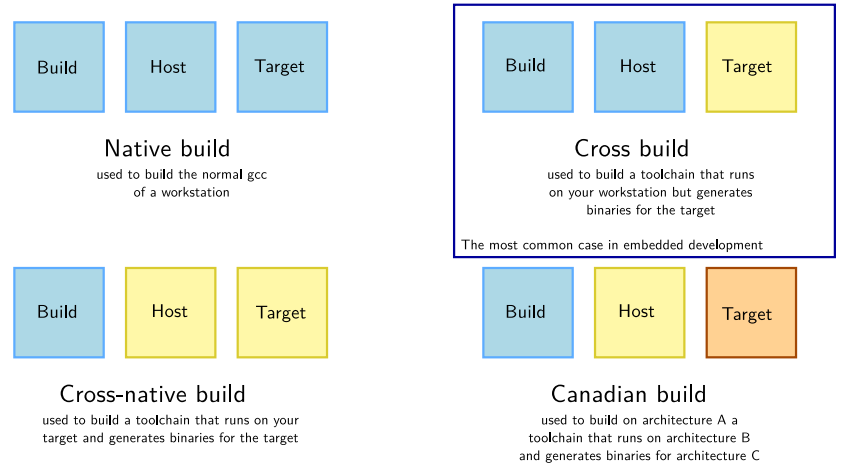
* If we define the word "host" to mean a computer on which you are compiling, and "target" as the computer on which you want to run the code, then a native toolchain is one where the target and the host are the same kind of.
* As above, the cross-compiling toolchain is one where the target and the host are usually difference of each other.
* Compilation and execution process with native toolchain and cross-compiling toolchain
* Reference: <http://free-electrons.com/doc/training/embedded-linux/embedded-linux-slides.pdf> , page 59/520



**Figure 4.1: Comparison between Cross-compiling toolchain and Native toolchain**

* **Toolchain compiler:**
* A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running.
* Native toolchain use native compiler, native compiler is a compiler that works on compilation for the same technology on which it runs. It uses the same operating system or platform as the software for which it is assembling machine language.
* Three machines must be distinguished when discussing toolchain creation:
* The build machine, where the toolchain is built.
* The host machine, where the toolchain will be executed.
* The target machine, where the binaries created by the toolchain are executed.

**Note:** Native toolchain use native compiler also have three machines but they the same with each other.



**Figure 4.2: Different kinds of toolchain**