FreeRTOS and introduction to Linux embedded

Livio Tenze
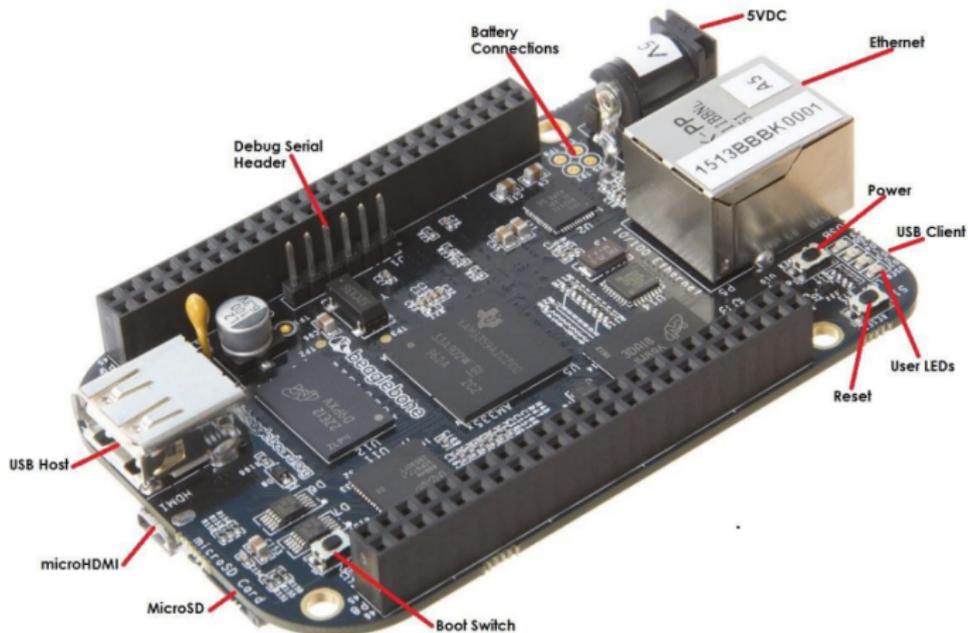
ltenze@units.it

February 12, 2025
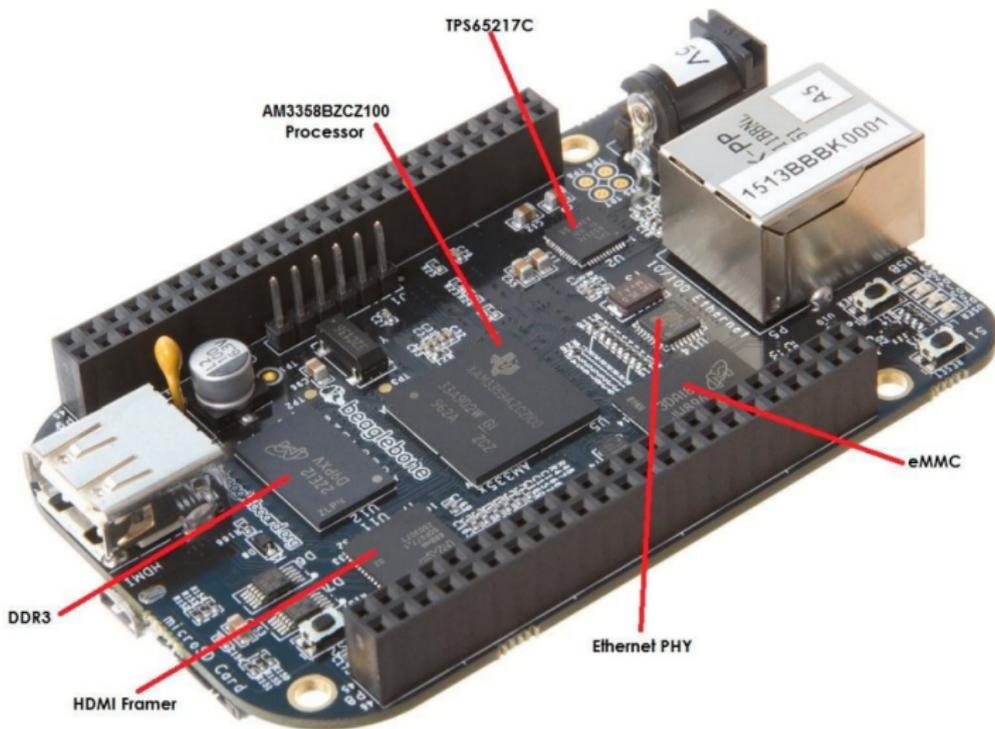
Part III
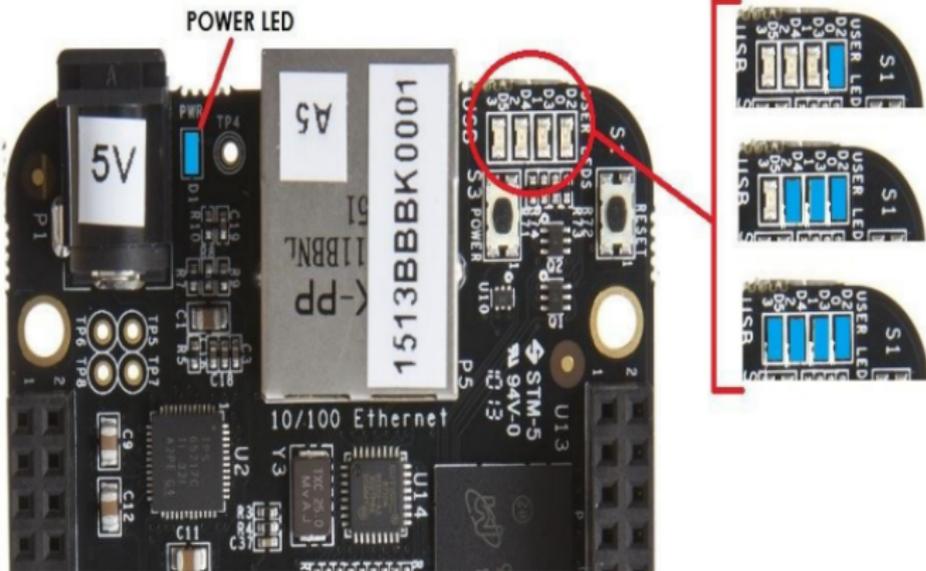
Introduction to Beaglebone Black (BBB)

Specifications

	Feature
Processor	Sitara AM3358BZCZ100 1GHz, 2000 MIPS
Graphics Engine	SGX530 3D, 20M Polygons/S
SDRAM Memory	512MB DDR3L 800MHZ
Onboard Flash	4GB, 8bit Embedded MMC
PMIC	TPS65217C PMIC regulator and one additional LDO.
Debug Support	Optional Onboard 20-pin CTI JTAG, Serial Header
Power Source	miniUSB USB or DC Jack
PCB	3.4" x 2.1"
Indicators	1-Power, 2-Ethernet, 4-User Controllable LEDs
HS USB 2.0 Client Port	Access to USB0, Client mode via miniUSB
HS USB 2.0 Host Port	Access to USB1, Type A Socket, 500mA LS/FS/HS
Serial Port	UART0 access via 6 pin 3.3V TTL Header. Header is populated
Ethernet	10/100, RJ45
SD/MMC Connector	microSD , 3.3V
User Input	1. Reset Button 2. Boot Button 3. Power Button
Video Out	1. 16b HDMI, 1280x1024 (MAX) 2. 1024x768,1280x720,1440x900 ,1920x1080@24Hz w/EDID Support
Audio	Via HDMI Interface, Stereo
Expansion Connectors	1. Power 5V, 3.3V , VDD_ADC(1.8V) 2. 3.3V I/O on all signals 3. McASP0, SPI1, I2C, GPIO(69 max), LCD, GPMC, MMC1, MMC2, 7 4. AIN _(1.8V MAX)_, 4 Timers, 4 Serial Ports, CAN0, 5. EHRPWM(0,2),XDMA Interrupt, Power button, Expansion Board ID (Up to 4 can be stacked)
Weight	1.4 oz (39.68 grams)
Power	Refer to section-6-1-7

Cortex-A8 specs

The ARM Cortex-A8 is a 32-bit processor core licensed by ARM Holdings implementing the ARMv7-A architecture.

Cortex-A8 specs

The ARM Cortex-A8 is a 32-bit processor core licensed by ARM Holdings implementing the ARMv7-A architecture.
Compared to the ARM11, the Cortex-A8 is a dual-issue superscalar design, achieving roughly twice the instructions per cycle. The Cortex-A8 was the first Cortex design to be adopted on a large scale in consumer devices.

Cortex-A8 specs

The ARM Cortex-A8 is a 32-bit processor core licensed by ARM Holdings implementing the ARMv7-A architecture.
Compared to the ARM11, the Cortex-A8 is a dual-issue superscalar design, achieving roughly twice the instructions per cycle. The Cortex-A8 was the first Cortex design to be adopted on a large scale in consumer devices.

- Frequency from 600 MHz to 1 GHz and above
- Superscalar dual-issue microarchitecture
- NEON SIMD instruction set extension
- 13-stage integer pipeline and 10-stage NEON pipeline
- VFPv3 floating-point unit
- Thumb-2 instruction set encoding
- Advanced branch prediction unit with ¿95
- 2.0 DMIPS/MHz

https://docs.beagleboard.org/beaglebone-black.pdf
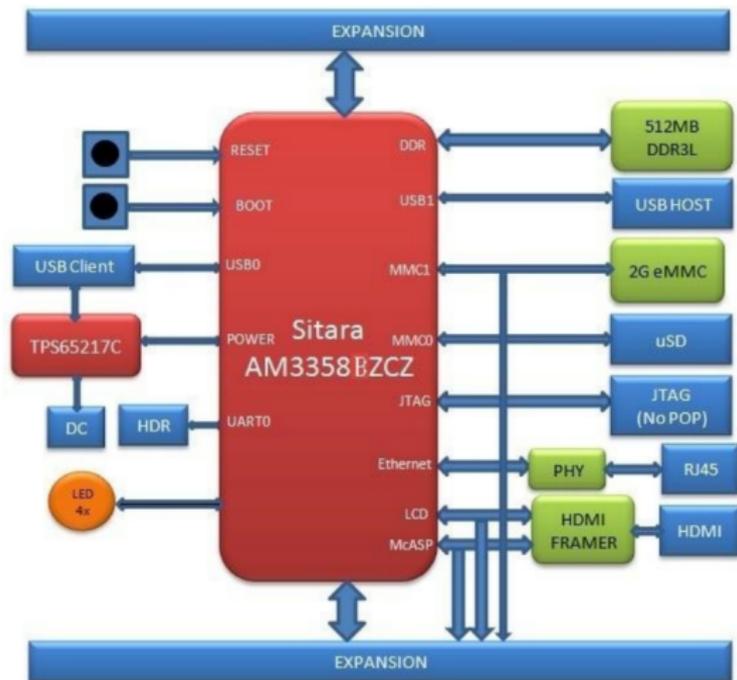
Part IV

Introduction to Linux embedded

Linux debuted as a general-purpose operating system for PC hardware with Intel x86 architecture. In his now famous post on news:comp.os.minix, Linux creator Linus Torvalds explicitly stated, "I'm doing a (free) operating system. . . . It is NOT portable (uses 386 task switching etc), and it probably never will support anything other than AT- harddisks . . . ."

Driven by the rise of the Internet, Linux quickly evolved into a server operating system providing infrastructure for web servers and networking services for many well-founded reasons.

Linux remained true to its GPOS origins in three major aspects that do not make it the premier choice of engineers for an embedded operating system at first:

Filesystem Linux is a file-based operating system requiring a filesystem on a block-oriented mass storage device with read and write access. Block-oriented mass storage typically meant hard drives with spinning platters, which are not practical for most embedded use cases.

Memory management Linux is a multitasking operating system.
Effective task switching mandates that individual
processes have their private memory address space
that can easily be mapped into physical memory
when the process is running on the CPU.
Microcontrollers that have been widely used for
typical embedded applications do not provide an
MMU.

Real time   Embedded systems running critical applications may require predictive responses with guaranteed timing within a certain margin of error, commonly referred to as determinism. The amount of error in the timing over subsequent iterations of a program or section thereof is called jitter. An operating system that can absolutely guarantee a maximum time for the operations it performs is referred to as a hard real-time system. An operating system that typically performs its operations within a certain time is referred to as soft real-time. Although several solutions providing real-time capabilities for Linux, most notably PREEMPT-RT, had been developed as early as 1996, they are still not part of the mainline Linux kernel.

During last years, advances in semiconductor technology have helped to overcome these hurdles for the adoption of Linux in embedded systems. Ubiquitously available, inexpensive, and long-term reliable flash memory devices used in many consumer products, such as digital cameras, are providing the necessary mass storage for the filesystem. Powerful system-on-chips (SoC) designs combining one or multiple general-purpose CPU cores with MMU and peripheral devices on a single chip have become the embedded systems engineer's choice of processor and are increasingly replacing the microcontroller in embedded applications.

Reasons for the rapid growth

Royalties
: Unlike traditional proprietary operating systems, Linux can be deployed without any royalties.

Hardware Support
: Linux supports a vast variety of hardware devices including all major and commonly used CPU architectures: ARM, Intel x86, MIPS, and PowerPC in their respective 32-bit and 64-bit variants.

Networking
: Linux supports a large variety of networking protocols. Besides the ubiquitous TCP/IP, virtually any other protocol on any physical medium is implemented.

Modularity
: A Linux OS stack is composed of many different software packages. Engineers can customize the stack to make it exactly fit their application.

Scalability
: Linux scales from systems with only one CPU and limited resources to systems featuring multiple CPUs with many cores, large memory footprints, several networking interfaces, and much more.

Reasons for the rapid growth

Source Code The source code for the Linux kernel, as well as for all software packages comprising a Linux OS stack, is openly available. Developer Support: Because of its openness, Linux has attracted a huge number of active developers, and those developers have quickly built support for new hardware.

Commercial Support An increasing number of hardware and software vendors, including all semiconductor manufacturers as well as many independent software vendors (ISV), are now offering support for Linux through products and services.

Tooling Linux provides myriad tools for software development ranging from compilers for virtually any programming language to a steadily growing number of profiling and performance measurement tools important for embedded systems development.

Landscape

Similar to desktop and server Linux distributions, an ever-evolving variety of embedded Linux distributions is developed by community projects and commercial operating system vendors. Some of them are targeted for a specific class of embedded systems and devices, while others are more general in nature with the idea to provide a foundation rather than a complete system.

- Android
- Angstrom
- OpenWrt
- Full linux distros (Debian, Ubuntu, Gentoo et cetera)

Development tools

Besides utilizing an embedded Linux distribution, you can also build your own custom Linux OS stack with embedded Linux development tools.

Baserock is an open source project that provides a build system for Linux distributions, a development environment, and a development workflow in one package. Baserock's major characteristics are

- Git as the core to manage essentially everything from build instructions to build artifacts as a means to provide traceability
- Native compilation to avoid the complexity of cross-build environments
- Distributed builds across multiple systems using virtual machines

Development tools

Buildroot is a build system for complete embedded Linux systems using GNU Make and a set of makefiles to create a cross-compilation toolchain, a root filesystem, a kernel image, and a bootloader image. Buildroot mainly targets small footprint embedded systems and supports various CPU architectures. To jump start development it limits the choice of configuration options:

- uClibc is the target library to build the cross-compilation toolchain. In comparison to the GNU C Library (glibc), uClibc is much more compact and optimized for small footprint embedded systems. uClibc supports virtually all CPU architectures as well as shared libraries and threading.

Buildroot
- BusyBox is the default set of command line utility applications.

These default settings enable building a basic embedded Linux system with Buildroot typically within 15 to 30 minutes, depending on the build host. However, these settings are not absolute, and the simple and flexible structure of Buildroot makes it easy to understand and extend. The internal cross-toolchain can be replaced by an external one such as crosstool-ng, and uClibc can be replaced with other C libraries.

OpenEmbedded is a build framework composed of tools, configuration data, and recipes to create Linux distributions targeted for embedded devices. At the core of OpenEmbedded is the BitBake task executor that manages the build process.

Yocto is not a single open source project but represents an entire family of projects that are developed and maintained under its umbrella. The main project in Yocto is Poky, the Yocto Project's reference distribution, which includes the OpenEmbedded build system and a comprehensive set of metadata.

OpenEmbedded is a build framework composed of tools, configuration data, and recipes to create Linux distributions targeted for embedded devices. At the core of OpenEmbedded is the BitBake task executor that manages the build process.

Yocto is not a single open source project but represents an entire family of projects that are developed and maintained under its umbrella. The main project in Yocto is Poky, the Yocto Project's reference distribution, which includes the OpenEmbedded build system and a comprehensive set of metadata. **We will introduce this development system.**

Let's face it—building and maintaining an operating system is not a trivial task. Many different aspects of the operating system have to be taken into consideration to create a fully functional computer system:

- Bootloader
- Kernel
- Device drivers
- Life Cycle Management
- Application Software Management

Custom distribution

Let's face it—building and maintaining an operating system is not a trivial task. Many different aspects of the operating system have to be taken into consideration to create a fully functional computer system:

- Bootloader
- Kernel
- Device drivers
- Life Cycle Management
- Application Software Management

Linux from scratch `https://www.linuxfromscratch.org/`

Understanding the architecture of a Linux system provides the context for the methods employed by OpenEmbedded Core (OE Core) to create the various system components, such as root filesystem images, kernel images, and bootloaders.

The bootloader is not strictly part of the Linux system, but it is necessary to start the system and therefore relevant in the context of building a fully functional system with the Yocto Project.

Linux architecture

Linux architecture
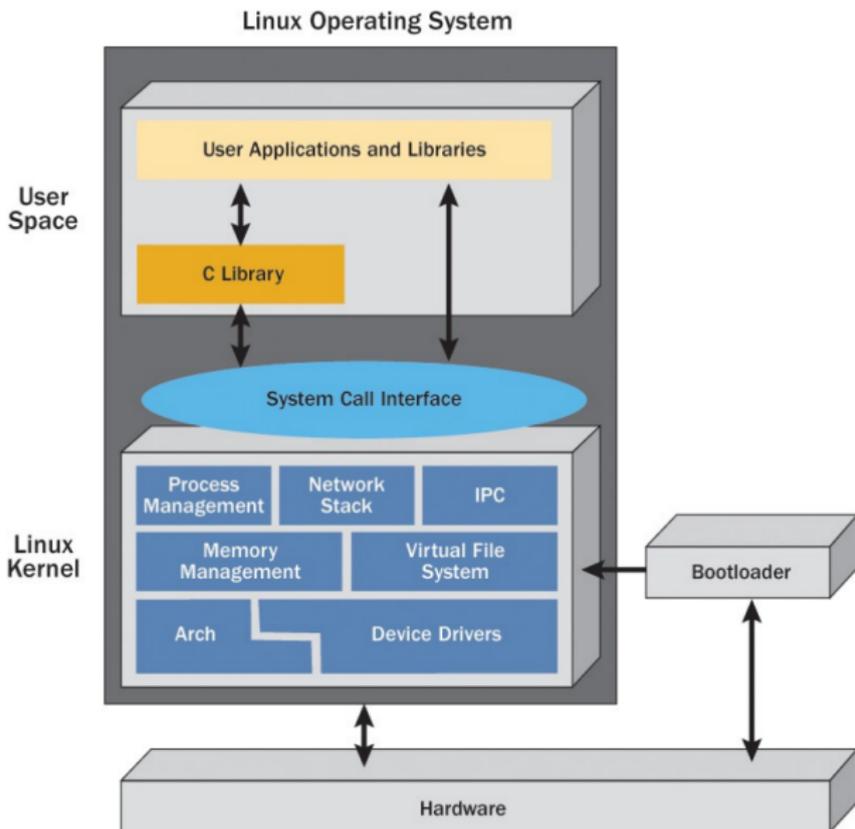
A Linux OS can be divided into two levels, the **kernel space** and the **user** or application space. This distinction is not just conceptual but originates from the fact that code from the kernel space is executed in a different processor operating mode than code from the user space. All kernel code is executed in unrestricted or privileged mode. In this mode, any instruction of the instruction set of the architecture can be executed. Application code, by contrast, is executed in restricted or user mode. In this mode, instructions that directly access hardware—input/output (I/O) instructions— or otherwise can alter the state of the machine are not permitted. Access to certain memory regions is typically also restricted.

Bootloader

Although the bootloader plays only a very short role in the life cycle of a system during startup, it is a very important system component. To some extent, configuring a bootloader is a common task for any Linux system running on standard PC-style hardware. For embedded systems, setting up a bootloader becomes a very special task, since hardware for embedded systems not only differs significantly from the standard PC architecture but also comes in many different variants. That is true not only for the different CPU architectures but also for the actual CPU or system-on-chips (SoC) as well as for the many peripherals that, combined with the CPU or SoC, make up the hardware platform.

Frequently, bootloaders are divided into two categories:

- loaders
- monitors

In the former case, the bootloader provides only the mere functionality to initialize the hardware and load the operating system.

In the latter case, the bootloader also includes a command-line interface through which a user can interact with the bootloader, which can be used for configuration, reprogramming, initialization, and testing of hardware and other tasks.

After power is applied to a processor board, **the majority of the hardware components need to be initialized** before any software application can be executed.

The bootloader typically just initializes the hardware necessary for the operating system kernel to boot. All other hardware and peripherals are initialized by the operating system itself at a later stage of the boot process. Once the operating system kernel takes control of the hardware, it **may reinitialize** hardware components originally set up by the bootloader.

Bootloader operations

The most important step during the early initialization of the hardware performed by the bootloader is the initialization of memory and memory controller. Commonly, CPUs start in real mode with a direct mapping between logical and physical addresses.

After the hardware is initialized, the bootloader

- locates the operating system, typically the kernel
- loads it into memory
- passes configuration parameters
- hands over control to the operating system

At this point, the bootloader has completed its responsibilities and terminates. The operating system later reclaims all memory space used by the bootloader.

Common bootloaders

As a Linux system developer you have many choices for a bootloader for your project. Bootloaders differ in functionality, processor, and operating system support. Many of them can boot not only Linux but also other operating systems. Another distinction is from what media they can boot the operating system. Besides floppy disk, hard drives, and USB storage devices, many of them can also boot from LAN via BOOTP and TFTP.

- LILO
- ELILO
- GRUB
- SYSLINUX
- BURG
- systemd-boot
- U-Boot (more than 1000 platforms)
- Redboot (ARM, MIPS, PPC, and x86)

U-Boot, the Universal Bootloader, also known as Das U-Boot, can be considered the Swiss army knife among the embedded Linux bootloaders. Based on the PPCBoot and ARMBoot projects and originally developed by Wolfgang Denk of DENX Software Engineering.

At the time of this writing, U-Boot supports more than 1,000 platforms, of which more than 600 are PowerPC based and more than 300 are ARM based. If your project uses any of these architectures, chances are that your hardware platform is already supported by U-Boot.

U-Boot also supports device trees for platform configuration.
Device trees, also referred to as open firmware or flattened device trees (FDTs), are data structures in byte code format containing platform-specific parameters, such as register locations and sizes, addresses, interrupts, and more, required by the Linux kernel to correctly access the hardware and boot. U-Boot copies the device tree data structures to a memory location.

The idea behind device trees is to provide platform configuration parameters during runtime, allowing the Linux kernel to be compiled for multiple platforms without specific information about the particular platform.

http://git.denx.de.

Redboot

The Red Hat Embedded Debug and Bootstrap Firmware (RedBoot) is a bootloader based on the eCos hardware abstraction layer. eCos is a free, open source, real-time operating system targeting embedded applications.

eCos and RedBoot were originally developed and maintained by Red Hat, but the company has discontinued development and all sources have since been relicensed under the GPL.

Because of its eCos heritage, RedBoot supports many embedded hardware platforms for virtually all architectures, among them ARM, MIPS, PPC, and x86. RedBoot's networking support includes BOOTP and DHCP. It can download images over Ethernet using TFTP.

Monitor functionality with an interactive command-line interface allows RedBoot configuration, image download and management, as well as boot scripting.

The two primary functions of an operating system's kernel are to

- Manage the computer's resources
- Allow other programs to execute and access the resources

The core resources of a computer are typically composed of:

- CPU
- Memory
- I/O devices

Kernel

Operating system kernel architectures are typically categorized as monolithic kernel or microkernel.

- Monolithic kernels execute all kernel functions, including device drivers within the core kernel process and memory context.
- Microkernels (e.g. Minix OS) execute only corefunctions, such as process and memory management within the core kernel context, and execute device drivers as separate user space processes.

A microkernel allows for easier system configuration and maintenance because device drivers can be loaded and unloaded while the system is running. That convenience, however, comes at a performance cost, since microkernels use interprocess communication (IPC) to exchange data between kernel modules.

Although the Linux kernel provides loadable kernel modules for device drivers that can be loaded and unloaded during runtime, **Linux is considered a monolithic kernel, since these modules are directly inserted into the kernel's execution context**. Because they are running within the kernel's execution context, Linux kernel modules have access to all system resources, including memory. Hence, IPC is not necessary for exchanging data, and therefore there is no performance hit.
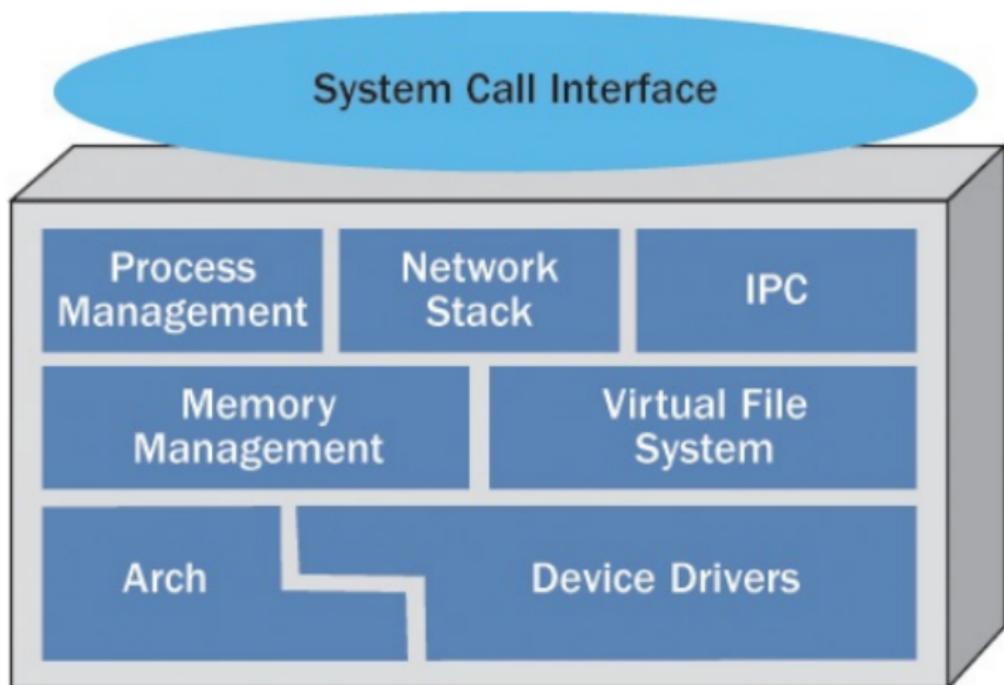
Architecture-Dependent Code

Although the majority of the Linux kernel code is independent of the architecture it is executed on, there are portions that need to take the CPU architecture and the platform into consideration. Inside the Linux kernel source tree, all architecture and platform-specific code is located in the linux/arch subdirectory. Within that subdirectory is another directory for each architecture supported by the Linux kernel. Every architecture directory contains a subdirectory kernel containing the architecture-specific kernel code.

Device drivers

Device drivers handle all the devices that the Linux kernel
supports. In most cases, these are hardware devices, but some
drivers implement software devices. One example is the software
watchdog timer.
Device drivers make up the vast majority of the Linux kernel code.
Inside the Linux source tree, device driver code is located in the
`linux/drivers` directory, which is further divided into
subdirectories for the various drivers supporting certain device
categories, such as Bluetooth, FireWire, I2C SCSI, and many more.

Memory management

Only the kernel has unrestricted access to the system's physical memory and is therefore responsible for safely allowing processes to access it. Most modern CPUs contain an MMU providing virtual addressing of a memory space that is typically much larger than the actual physical memory of the system. Virtual addressing allows each process to have its own private memory space that is protected (by the kernel) from other processes.

Virtual addressing allows allocation of more memory than is physically present in the system. When physical memory is exhausted, the kernel can move pages from the memory onto external storage such as a hard drive.

You can find the memory management code in the linux/mm directory of the kernel source tree.

Virtual filesystem

A filesystem is an organizational scheme for persistent data storage after an application terminates. It provides mechanisms to write, read, update, and erase data and manages the available space on the storage medium.

Unlike other operating systems, Linux gives users the choice of many different file- systems for a variety of applications and storage media. Besides the core Linux filesystems ext2, ext3, and ext4, Linux offers support for many others, including VFAT, NTFS, ZFS, and the new Btrfs.

For this purpose, the Linux kernel provides a common abstraction interface for file operations known as the virtual file system (VFS).

Virtual filesystem

The VFS is a switching fabric between the underlying filesystem implementations and file operations of the system call interface (SCI). The filesystem implementations are essentially data management plugins that reside between the VFS layer above and a unified data buffer below. The purpose of the data buffer is to optimize data access to the physical storage devices.

An interesting aspect of the VFS is that it is not limited to filesystems residing on physical storage devices but likewise provides the same uniform interface for network filesystems, such as Network File System (NFS) and Server Message Block (SMB), as well as for pseudo-filesystems such as the proc filesystem. Sources for VFS and the filesystem reside in the `linux/fs` directory of the Linux kernel source tree.

Process management

The application programming domain typically distinguishes between processes and threads. In this context, process refers to the execution context of an application, and thread refers to independent execution paths inside a process. A process has at least one thread, its main thread, from which it can spawn additional threads. All threads of a process share the same execution context, memory space, and other resources. Therefore, threads are also commonly referred to as **lightweight processes**.

The Linux kernel does not separate the two concepts of processes and threads. Both of them are implemented as threads that represent a complete execution context comprising code, data, stack, and CPU registers.

Process management

Process management allocates the core resource of a computer system, the CPUs. As threads compete for the available CPUs, it is the task of the scheduler to select the threads eligible to run and assign them to the available CPUs (or CPU cores).
The Linux kernel'sdefault scheduling algorithm is the Completely Fair Scheduler (CFS).

The Linux kernel also offers real-time scheduling policies with static priorities. The latest addition to the real-time scheduling policies is deadline scheduling, a policy that uses dynamic priorities based on the closest expiring deadline.

The Linux kernel's network stack is essentially modeled after the well-known ISO Open Systems Interconnection (OSI) layered architecture, as defined by ISO/IEC 7498-1. On the network layer, Linux of course supports the IPv4 and IPv6 protocols but also AppleTalk, IPX, X.25, Frame Relay, and others. Transport layer protocols include TCP, UDP, SPX, and more.

The network stack implementations can be found in `linux/net` inside the Linux kernel source tree.

IPC is a set of methods for data exchange between processes or threads. IPC methods are typically subdivided into message passing, shared memory, synchronization, and data streams. Implementation of the Linux kernel IPC methods can be found in `linux/ipc`.

System Call Interface (SCI)

The SCI is the connection between the Linux kernel and applications running in user space. Through the SCI, the kernel provides a common API of function calls for process management, file management, device management, interprocess communication, and system management.

The Linux kernel's SCI comprises over 300 functions. The exact number is dependent on the architecture. The majority of the functions are common, though their implementations may be architecture-dependent. Some functions may be architecture-specific and only supported by a particular architecture. The SCI implementation can be found in `linux/kernel` with the architecture-dependent portions in the subdirectories of `linux/arch`.

The Linux kernel runs through many stages of initialization for the various hardware components and subsystems. Many of these stages are dependent on the hardware platform.

1. After the bootloader has copied the Linux kernel image into memory, it passes control to the bootstrap loader that is a prepended part of the kernel image. To save space, the kernel image is typically compressed, and it is the bootstrap loader's responsibility to create the proper execution environment for the kernel, decompress the kernel, relocate the kernel in memory, and then pass control to it. The bootstrap loader directly passes control to the kernel entry point inside a module, which is called head.o for most architectures.

2. the head.o module performs the following tasks:
   - Verifies the correct architecture and CPU
   - Detects CPU type and functionality, such as hardware floating-point capabilities
   - Enables the CPU's MMU and creates the initial table of memory pages
   - Establishes basic error reporting and handling
   - Switches to non-architecture-specific kernel startup function start_kernel() in main.c (in linux/init)

3. main.c contains the bulk of the Linux kernel startup code, from architecture setup, kernel command-line processing, and initialization of the first kernel thread to mounting the root filesystem and executing the first user space application program.

4. start_kernel() function calls rest_init(), which spawns the first kernel thread. This thread is spawned by calling kernel_thread() with the function kernel_init() as the first parameter.

5. At this point, there are now two threads running: start_kernel() and kernel_init().

   - The former kicks off the scheduler and then loops forever in the cpu_idle() function.
   - The latter becomes the init() thread, the parent of all user space processes with the process ID (PID) of 1.

Now that the kernel has completed its initialization, launched the init process, and within **it executed the first user space application, the system has entered userland or user space**. User space is all the code that runs outside the operating system's kernel and includes all the libraries and application programs. User space provides all the functionality required for the system to serve its intended purpose.

Configuration of the user space and which libraries and application programs it includes differ from system to system. However, there is always one library that virtually all systems include: the **C Standard Library (LIBC)**. Just as the init process is the parent of all processes, LIBC can be considered the parent of all libraries.

```c
#include <stdio.h>

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

This program calls printf(), which is one of many functions provided by the LIBC APIs, alleviating the burden on application programmers to perform the more tedious tasks of implementing core functionality required by virtually any program.

The LIBC APIs are specified by the ANSI C Standard. For UNIX systems, the ANSI C Standard is described as part of the POSIX library, which is a superset of it. POSIX is an IEEE standard.

Many of the LIBC APIs directly map to the kernel's systems calls. In fact, frequently, the implementation of the function is merely a wrapper around the system call. For Linux systems, multiple implementations of LIBC are available.

They vary by footprint of the library itself, compatibility with the ANSI C Standard, performance, modularity, and configurability.

//AGGIUNGERE LISTA

Yocto is the prefix of the smallest fraction for units of measurements specified by the International System of Units. It gives the name to the Yocto Project, a comprehensive suite of tools, templates, and resources to build custom Linux distributions for embedded devices.

Yocto is the prefix of the smallest fraction for units of measurements specified by the International System of Units. It gives the name to the Yocto Project, a comprehensive suite of tools, templates, and resources to build custom Linux distributions for embedded devices.

We start *in medias res* with setting up the OpenEmbedded build system, provided by the Yocto Project in the Poky reference distribution, and building our first Linux OS stack relying entirely on the blueprint that Poky provides by default.

Yocto base requirements

- to build a Linux system with the Yocto Project tools, you need a build host running Linux
- To build Linux OS stacks, the Yocto Project tools require a build host with an x86 architecture CPU. Both 32-bit and 64-bit CPUs are supported. A system with a 64-bit CPU is preferred.
- Memory is also an important factor. BitBake, the Yocto Project build engine, parses thousands of recipes and creates a cache with build dependencies (4 GB recommended).

- Disk space is another consideration. A full build process, which creates an image with a graphical user interface (GUI) based on X11 currently consumes about 50 GB of disk space.
- The OpenEmbedded build system that you obtain from the project's website contains only the build system itself—BitBake and the metadata that guide it. It does not contain any source packages for the software it is going to build. These are automatically downloaded as needed while a build is running. Therefore, you need a live connection to the Internet, preferably a high-speed connection.

Yocto: how to download

There are several ways for you to obtain the Yocto Project tools, or more precisely, the Yocto Project reference distribution, Poky:

- Download the current release from the Yocto Project website. `https://www.yoctoproject.org/downloads`
- Download the current release or previously released versions from the release repository.
- Download a recent nightly build from the Autobuilder repository.
- Clone the current development branch or other branches from the Poky Git repository hosted by the Yocto Project Git repository server.

`https://wiki.yoctoproject.org/wiki/Releases`

Yocto: prepare the host system

Before starting the compilation, some steps are needed:

- install host tools such as gawk, wget, git-core, diffstat and other
- download the Yocto poky source
- configure the build environment
- *modify the configuration files* (at least conf/local.conf)

Instead of configuring from scratch the compilation environment, I prepared a docker container where all requirements should be met:

```
docker run --name yoctobbb -d -v ./<builddir>:/home/ubuntu/yocto/bbb/build
    -p 127.0.0.1:8000:8000/tcp livius147/yoctobbb:latest
```

In local.conf, various variables are set that influence how BitBake builds your custom Linux OS stack. You can modify the settings and also add new settings to the file to override settings that are made in other configuration files.

```
BB_NUMBER_THREADS ?= "${@bb.utils.cpu_count()}"
PARALLEL_MAKE ?= "-j ${@bb.utils.cpu_count()}"
MACHINE ??= "qemux86"
DL_DIR ?= "${TOPDIR}/downloads"
SSTATE_DIR ?= "${TOPDIR}/sstate-cache"
TMP_DIR = "${TOPDIR}/tmp"
```

- BB_NUMBER_THREADS: Number of parallel BitBake tasks
- PARALLEL_MAKE: Number of parallel make processes
- MACHINE: Target machine
- DL_DIR: Directory where source downloads are placed
- SSTATE_DIR: Directory for shared state cache files
- TMP_DIR: Directory for build output

To conserve disk space during a build, you can add

$$\text{INHERIT} \mathrel{+}= \texttt{rm\_work}$$

which instructs BitBake to delete the work directory for building packages after the package has been built.

Launching the build

To launch a build, invoke BitBake from the top-level directory of your build environment specifying a build target:

```
$ bitbake <build-target>
```

For our first build we use a build target that creates an entire Linux OS stack with a GUI. From the top-level directory of the build environment you previously created and configured, execute the following:

```
$ bitbake core-image-sato
```

The core-image-sato target creates a root file system image with a user interface for mobile devices. Depending on your build hardware and the speed of your Internet connection for downloading the source files, the build can take anywhere from one to several hours.

Build your first image

In order to create the first build with Yocto project you need to follow these steps:

- Start the docker image and exec bash to get a console access
- `source oe-init-build-env build_bbb`
- Modifiy the conf/local.conf and set
  `MACHINE ?= "beaglebone"` and
  `PACKAGE_CLASSES ?= "package_ipk"`
- Add to conf/local.conf `IMAGE_FSTYPES += " ext4 tar.gz"`
  to simplify the deployment to the SD card
- optional `INHERIT += "rm\_work"`

bitbake core-image-sato

- `bitbake core-image-sato`
- use `-k` This option can be useful when we are building images with a lot of packages and we are not sure of errors. In such cases, we will have to build with some of the failures which we can fix, and invoke our build again.
- `bitbake -c compile <package_name> -f` This option tells BitBake to ignore the stamps and run the specific task for that target, without considering whether it is already run or not.

- `bitbake -c compile <recipe name>` The BitBake recipe consists of multiple tasks; each of these is responsible for some functionality, such as fetching source code, configuring it, compiling it, and so on. It is more like targets in the case of Make. `https://docs.yoctoproject.org/ref-manual/tasks.html`

- `bitbake -c listtasks procps`

- `bitbake -C compile busybox` This option tells BitBake to invalidate a stamp for a specified task, and then run the default build task. It is useful in cases where we are sure of our fix in some specific task, and we want to run all the remaining tasks of this package.

Bitbake typical commands

- `-v` verbose
- `bitbake -e busybox` This is a very useful option. This is a kind of wiss army knife in your Yocto Project / BitBake toolbox, while working with recipes. You always need to know what is actually happening and which variable is getting what value.
- `bitbake matchbox-desktop -c devshell`
  `https://docs.yoctoproject.org/dev-manual/`
  `development-shell.html`

Building images for our desired target may take some time, depending on network speed and processing power. After the build is complete, you will have your images ready at tmp/deploy/images/beaglebone/ under your build directory, as shown in the following screenshot. This contains first-level bootloader MLO, second-level bootloader u-boot, kernel image, device tree blobs, a root filesystem archive, and a modules archive.

Navigate in the build/tmp directory to check what has been produced by the compilation.

Results after compilation

Recipe tasks

In order to build open source packages you need to follow a specific pattern. Some of the steps of this workflow you execute yourself, whereas others are typically carried out through some sort of automation such as Make or other source-to-binary build systems.

- Fetch: Obtain the source code.
- Extract: Unpack the source code.
- Patch: Apply patches for bug fixes and added functionality.
- Configure: Prepare the build process according to the environment.
- Build: Compile and link.
- Install: Copy binaries and auxiliary files to their target directories.
- Package: Bundle binaries and auxiliary files for installation on other systems.

If you are building the software package only for use on the host system you use for building, then you would normally stop after installing the binaries on your system. However, if you are looking to distribute the binaries for installation and use on other systems, you would also include the package step, which creates an archive that can be used by the package management system for installation.

The OpenEmbedded build system supports organizing Metadata into multiple layers. Layers allow you to isolate different types of customizations from each other.
Each layer contains: recipes, conf (for machine and distro), classes.
We will create a new layer with a bitbake utility.
https://docs.yoctoproject.org/dev-manual/layers.html
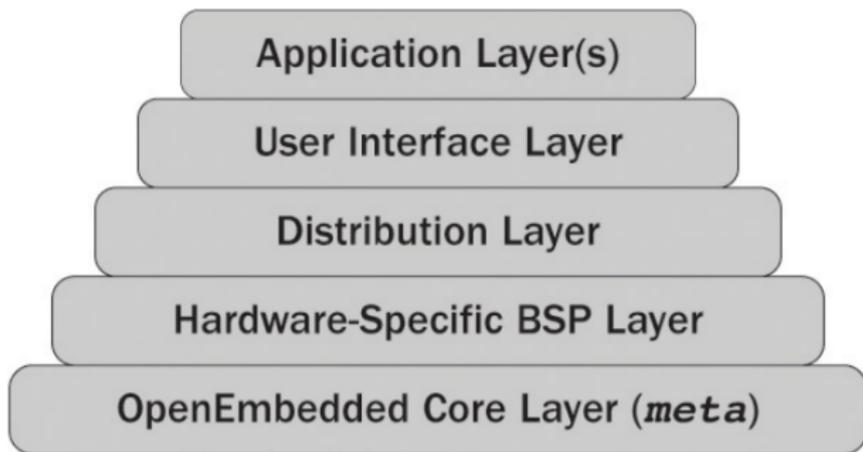
A build environment for a device operating system stack would typically include other layers, such as a BSP layer for actual hardware; a distribution layer specifying the OS configuration for user accounts, system startup, and more; and a user interface layer and application layers for the user space applications providing the device functionality.

We want to create a new layer where we want to put new recipes.
The Yocto project provides a simple command to create and
manage layers:

`bitbake-layers -h`

```
BitBake layers utility

options:
  -d, --debug          Enable debug output
  -q, --quiet          Print only errors
  -F, --force          Force add without recipe parse verification
  --color COLOR        Colorize output (where COLOR is auto, always, never)
  -h, --help           show this help message and exit

subcommands:
  <subcommand>
    show-layers        show current configured layers.
    show-overlayed     list overlayed recipes (where the same recipe exists in another layer)
    show-recipes       list available recipes, showing the layer they are provided by
    show-appends       list bbappend files and recipe files they apply to
    show-cross-depends Show dependencies between recipes that cross layer boundaries.
    add-layer          Add one or more layers to bblayers.conf.
    remove-layer       Remove one or more layers from bblayers.conf.
    flatten            flatten layer configuration into a separate output directory.
    layerindex-fetch   Fetches a layer from a layer index along with its dependent layers, and adds them to conf/bblayers.conf.
    layerindex-show-depends
                       Find layer dependencies from layer index.
    create-layer       Create a basic layer
```

```
bitbake-layers create-layer meta-units
bitbake-layers add-layer meta-units
bitbake-layers show-layers
```



```
https:
//docs.yoctoproject.org/dev/dev-manual/layers.html
```

We have just created a new layer and we want to create a new recipe. The idea is to download the source from the network and to obtain an installable package.

```
# We use the devtool utility to setup a new recipe based on a git source
devtool add pacman4console \
git://github.com/YoctoForBeaglebone/pacman4console.git;branch=master
# Then we try to compile the package, we obtain an error due to wrong Makefile
devtool build pacman4console
# We have to modify the Makefile by replacing gcc with ${CC}
vi workspace/sources/pacman4console/Makefile
# The system does not find ncurses libraries... add them
devtool build pacman4console
# Add DEPENDS += "ncurses"
devtool edit-recipe pacman4console
# Rebuild, package is empty
devtool build pacman4console
# Add FILES_${PN} = <list of files>
devtool edit-recipe pacman4console
```

Frame Title

```
# Now the compilation is ok
devtool build pacman4console
# We have to commit the changes in sources
cd workspace/sources/pacman4console/
git commit -a -m "Patch in makefile"
# Finally we can add the recipe to the meta-units layer
devtool finish pacman4console /home/ubuntu/yocto/bbb/meta-units/
```

https:
//docs.yoctoproject.org/dev-manual/new-recipe.html

If we want to add the pacman4console package to the final image, we can modify the conf/local.conf file as follows:

- check the pacman4console compiles with
  bitbake pacman4console
- edit the local.conf file
- add IMAGE_INSTALL += "pacman4console"
- recompile the image with bitbake core-image-sato
- Now the new recipe is deployed into the final root filesystem in /usr/local/bin, check it!

Some highlights about variables assignment

$=$ This is a simple variable assignment. It requires "" and spaces are significant.

$?=$ This is used to assign a default value to a variable. It can be overridden using $=$.

$??=$ This is a weak default value assignment. This is similar to $?=$, but it is done at the end of the parsing process. If multiple $??=$ are used, only the last one is picked.

$:=$ This is an immediate variable expansion. The value thus assigned is expanded immediately.

$+=$ This appends a value to a variable with space.

Some highlights about variables assignment

$=+$ This prepends a value to a variable with space.

   $.=$ This is a kind of string concatenation. This appends a value to a variable without any space.

   $=.$ This prepends a value without space.

_append   This is the same as $.=$ but more readable and widely used. Also, expanded immediately. Also keep in mind that it is not an immediate assignment.

_prepend   This is the same as $=.$, but more readable and widely used. Also, keep in mind, that it is not an immediate assignment.

_remove   This causes all occurrences of values from a variable to be removed.

DEMO
https://wiki.koansoftware.com/index.php/Using_
devtool_to_modify_recipes_in_Yocto

```
https://docs.yoctoproject.org/dev-manual/index.html
https://www.campana.vi.it/blog/boot-beaglebone-black/
```