Embedded Linux

Anatomy of a Linux-based System

Goals

To provide a more detailed view of the Linux architecture

To illustrate the device tree details

To introduce the U-BOOT bootloader

To illustrate the detailed boot process for the UDOO NEO

Summary

Linux architecture

Device trees

The U-BOOT bootloader

UDOO NEO boot process

Summary

Linux architecture

Device trees

The U-BOOT bootloader

UDOO NEO boot process

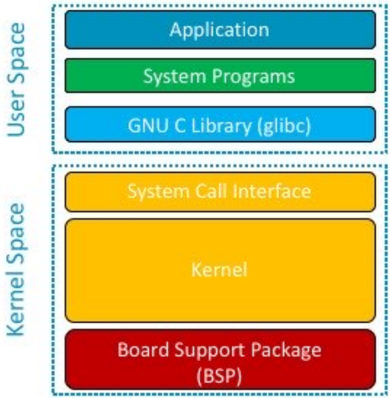
Linux Architecture

Layered architecture based on two levels:

- User space
- Kernel space

User space and kernel space are independent and isolated

User space and kernel space communicate through special purpose functions known as system calls



arm

Linux Architecture

Application

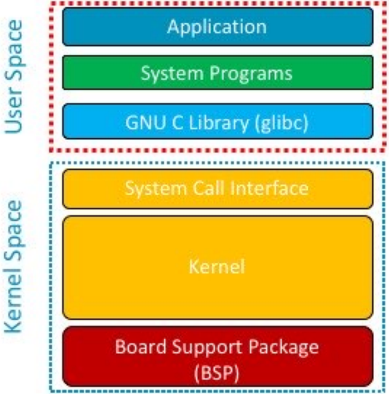
- Software implementing the functionalities to be delivered to the embedded system user

System programs

- User-friendly utilities to access operating system services

GNU C Library (glibc)

- Interface between the User Space and the Kernel Space



Linux Architecture

System call interface

- Entry points to access the services provided by the Kernel (process management, memory management)

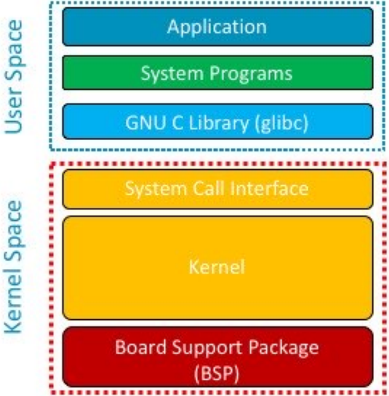
such as

Kernel

- Architecture-independent operating system code
- It implements the hardware-agnostic services of the operating system (e.g. the process scheduler).

Board Support Package (BSP)

- Architecure-dependant operating system code
- It implements the hardware specific services of the operating system (e.g. the context switch).



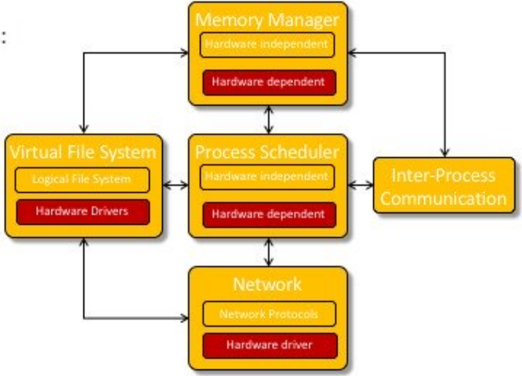
Conceptual View of the Kernel

The Kernel can be divided in five subsystems:

- Process scheduler
- Memory manager
- Virtual file system
- Inter-process communication (IPC)
- Network

Most of them are composed of:

- Hardware-independent code
- Hardware-dependent code



Process Scheduler

→ เป็น response ของการตัดสินใจ
ว่าใครจะได้ รัน ขณะนั้น

Main functions:

- Allows processes to create new copies of themselves
- Implements CPU scheduling policy and context switch
- Receives, interrupts, and routes them to the appropriate Kernel subsystem
- Sends signals to user processes
- Manages the hardware timer
- Cleans up process resources when a processes finishes executing
- Provides support for loadable Kernel modules

arm

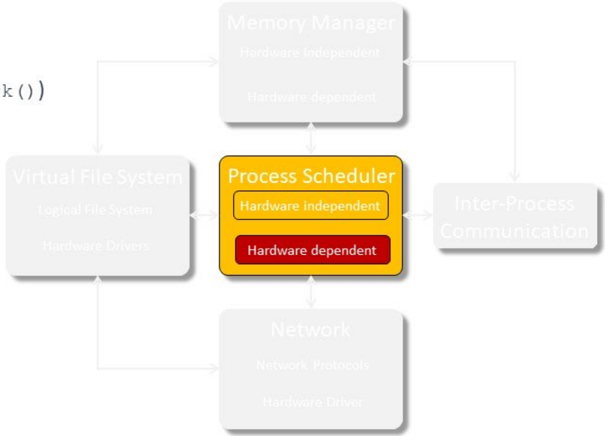
Process Scheduler

External interface:

- System calls interface towards the user space (e.g. `fork()`)
- Intra-Kernel interface towards the kernel space (e.g. `create_module()`)

Scheduler tick:

- Directly from system calls (e.g. `sleep()`)
- Indirectly after every system call
- After every slow interrupt



Process Scheduler

Interrupt type:

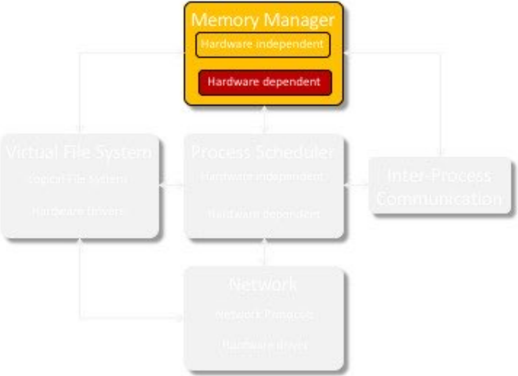
- Slow: traditional interrupt (e.g. coming from a disk driver)
- Fast: interrupt corresponding to very fast operations (e.g. processing a keyboard input)

arm

Memory Manager

It is responsible for handling:

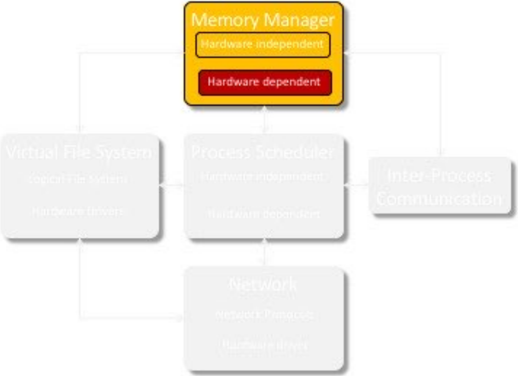
- Large address space: user processes can reference more RAM memory than what exists physically
- Protection: the memory for a process is private and cannot be read or modified by another process; also, the memory manager prevents processes from overwriting code and read-only-data.
- Memory mapping: processes can map a file into an area of virtual memory and access the file as memory.



Memory Manager

It is responsible for handling:

- Fair access to physical memory: it ensures that processes all have fair access to the memory resources, ensuring reasonable system performance.
- Shared memory: it allows processes to share some portion of their memory (e.g. executable code is usually shared amongst processes).



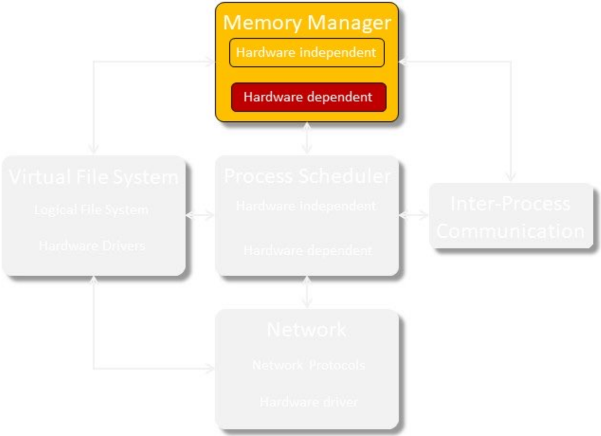
Memory Manager

It uses the Memory Management Unit (MMU) to map virtual addresses to physical addresses.

- It is conventional for a Linux system to have a form of MMU support.

Advantages:

- Processes can be moved among physical memory maintaining the same virtual addresses.
- The same physical memory may be shared among different processes.



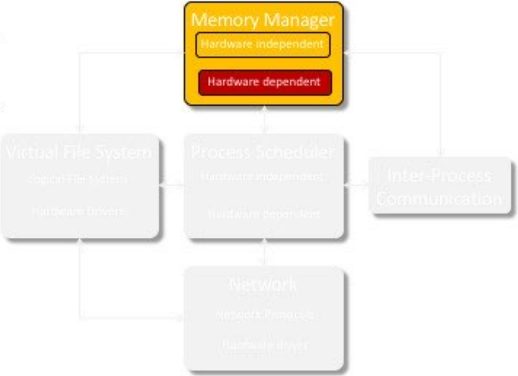
Memory Manager

It swaps process memory out to a paging file when it is not in use:

- Processes using more memory than physically available can be executed.

The `kswapd` Kernel-space process (also known as daemon) is used for this purpose.

- It checks if there are any physical memory pages that haven't been referenced recently .
- These pages are evicted from physical memory and stored in a paging file.

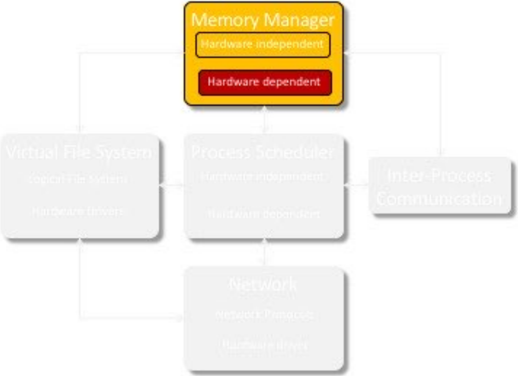


Memory Manager

The MMU detects when a user process accesses a memory address that is not currently mapped to a physical memory location.

The MMU notifies the Linux Kernel the event known as page fault.

The memory manager subsystem resolves the page fault.

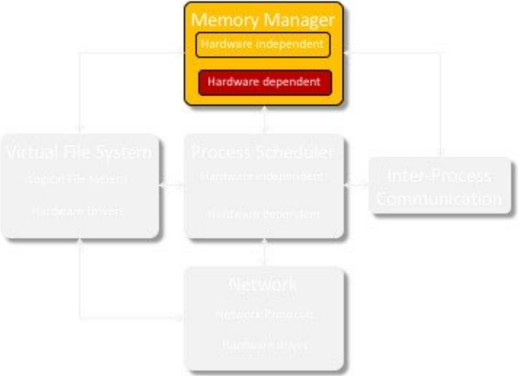


Memory Manager

If the page is currently swapped out to the paging file, it is swapped back in.

If the memory manager detects an invalid memory access, it notifies the event to the user process with a signal.

If the process doesn't handle this signal, it is terminated.



Memory Manager External Interfaces

System call interface:

- `malloc()/free()`: allocate or free a region of memory for the process's use
- `mmap()/munmap()/msync()/mremap()`: map files into virtual memory regions
- `mprotect()`: change the protection on a region of virtual memory
- `mlock()/mlockall()/munlock()/munlockall()`: super-user routines to prevent memory being swapped
- `swapon()/swapoff()`: super-user routines to add and remove swap files for the system

Intra-Kernel interface:

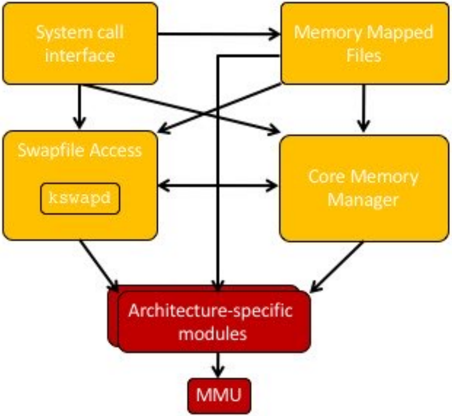
- `kmalloc()/kfree()`: allocate and free memory for use by the kernel's data structures
- `verify_area()`: verify that a region of user memory is mapped with required permissions
- `get_free_page()/free_page()`: allocate and free physical memory pages

Memory Manager Architecture

ระบบถูกแบ่งเป็น small component

Main components:

- System call interface: it provides memory manager services to the user space.
- Memory mapped files: it implements memory file mapping algorithms.
- Core memory manager: it is responsible for implementing memory allocation algorithms.
- Swapfile access: it controls the paging file access.
- Architecture-specific modules: they handle hardware-specific operations related to memory management (e.g. access to the MMU).

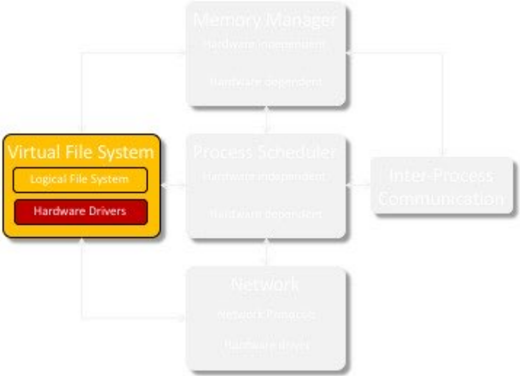


arm

Virtual File System

It is responsible for handling:

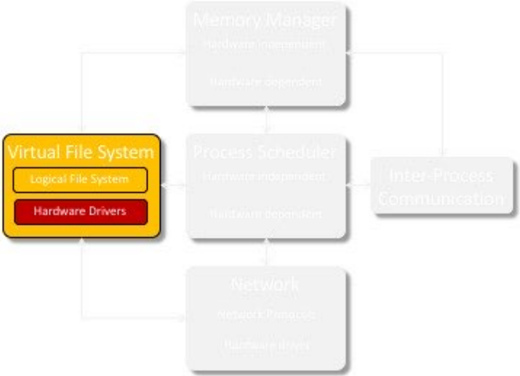
- Multiple hardware devices: it provides uniform access to hardware devices.
- Multiple logical file systems: it supports many different logical organizations of information on storage media.
- Multiple executable formats: it supports different executable file formats (e.g. a.out, ELF).
- Homogeneity: it presents a common interface to all of the logical file systems and all hardware devices.



Virtual File System

It is responsible for handling:

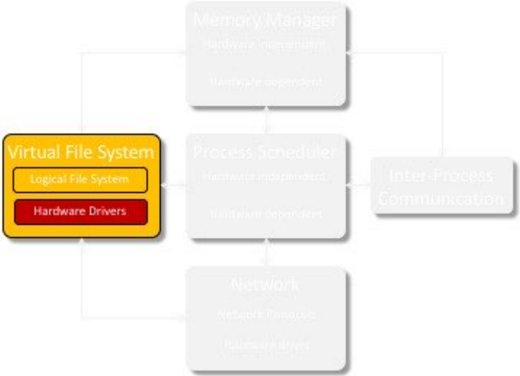
- Performance: it provides high-speed access to files
- Safety: it enforces policies to not lose or corrupt data
- Security: it enforces policies to grant access to files only to allowed users, and it restricts user total file size with quotas.



Virtual File System

External interface:

- System-call interface based on normal operations on file from the POSIX standard (e.g. open/close/read/write)
- Intra-kernel interface based on i-node interface and file interface



arm

i-node : data structure

It stores all the information about a file excepts its name and the data it contains.

When a file is created, it is assigned a name and a unique i-node number (a unique integer number).

When a file is accessed

- Each file is associated with a unique i-node number.
- The i-node number is then used for accessing the data structure containing the information about the file being accessed.

```
struct inode {
    struct hlist_node       i_hash;
    struct list_head        i_list;
    struct list_head        i_sb_list;
    struct list_head        i_dentry;
    unsigned long           i_ino;
    atomic_t                i_count;
    umode_t                 i_mode;
    unsigned int            i_nlink;
    uid_t                   i_uid;
    gid_t                   i_gid;
    dev_t                   i_rdev;
    loff_t                  i_size;
    struct timespec         i_atime;
    struct timespec         i_mtime;
    struct timespec         i_ctime;
                ...
```

i-node Interface

`create()`: creates a file in a directory

`lookup()`: finds a file by name within a directory

`link()/symlink()/unlink()/readlink()/follow_link()`: manages file system links

`mkdir()/rmdir()`: creates or removes sub-directories

`mknod()`: creates a directory, special file, or regular file

`readpage()/writepage()`: reads or writes a page of physical memory

`truncate()`: sets the length of a file to zero

`permission()`: checks to see if a user process has permission to execute an operation

`smap()`: maps a logical file block to a physical device sector

`bmap()`: maps a logical file block to a physical device block

`rename()`: renames a file or directory

File Interface (คล้าย i-node interface)

`open()`/`release()`: opens or closes the file

`read()`/`write()`: reads or writes the file

`select()`: waits until the file is in a particular state (readable or writeable)

`lseek()`: moves to a particular offset in the file

`mmap()`: maps a region of the file onto the virtual memory of a user process

`fsync()`/`fasync()`: synchronizes any memory buffers with the physical device

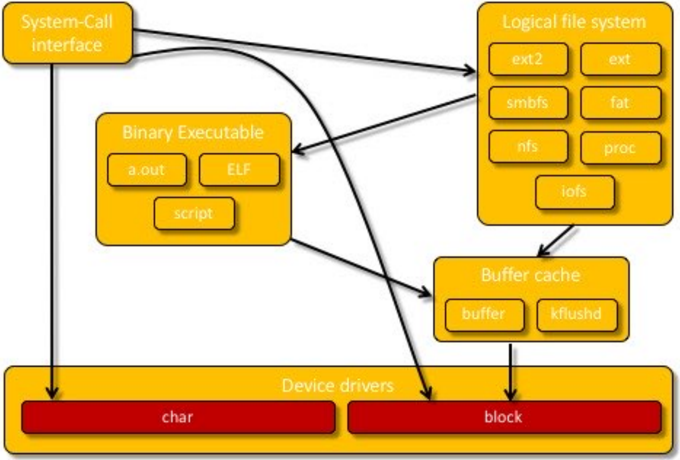
`readdir()`: reads the files that are pointed to by a directory file

`ioctl()`: sets file attributes

`check_media_change()`: checks to see if a removable media has been removed

`revalidate()`: verifies that all cached information is valid

Virtual File System Architecture



Virtual File System Architecture

System call interface: it provides Virtual File System services to the user space

Logical file system: it provides a logical structure for the information stored in a storage medium.

- Several logical file systems are supported (e.g. ext2, fat).
- All files appear the same to the user.
- The i-node is used to hide logical file system details.
- For each file, the corresponding logical file system type is stored in the i-node.
- Depending on the information in the i-node, the proper operations are activated when reading/writing a file in a given logical file system.

Buffer cache: it provides data caching mechanisms to improve performance of storage media access operations.

Binary executable: it supports different types of executable files transparently to the user.

arm

Virtual File System Architecture

Device drivers provide a uniform interface to access hardware devices:

- Character-based devices are hardware devices accessed sequentially (e.g. serial port).
- Block-based devices are devices that are accessed randomly and whose data is read/written in blocks (e.g. hard disk unit).

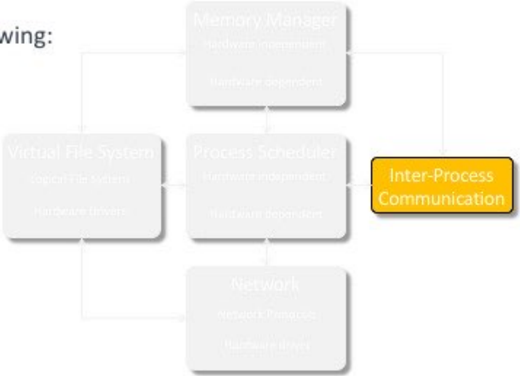
Device drivers use the file interface abstraction:

- Each device can be accessed as a file in the file system through a special file, the device file, associated with it.
- A new device driver is a new implementing of the hardware-specific code to customize the file interface abstraction (more about this later).

Inter-process Communication (IPC)

It provides mechanisms to processes for allowing:

- Resource sharing
- Synchronization
- Data exchange



arm

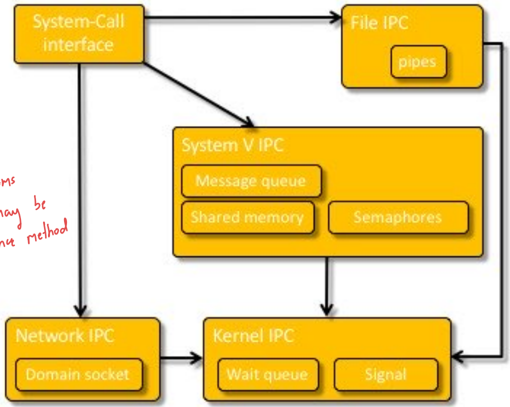
Inter-process Communication Architecture

System call interface: it provides inter-process communication (IPC) services to the user space as a system call interface

The following IPCs are supported:

IPC can take many forms as different processes may be communicate in difference method

- Pipes
- Message queues
- Shared memory
- Semaphores
- Domain sockets
- Wait queues
- Signals

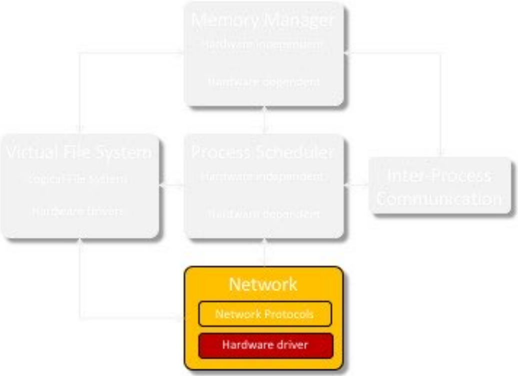


arm

Network

Provides support for network connectivity

- It implements network protocols (e.g. TCP/IP) through hardware-independent code.
- It implements network card drivers through hardware-specific code.



arm

Summary

Linux architecture

Device trees

The U-BOOT bootloader

UDOO NEO boot process

Device Trees

To manage hardware resources, the Kernel must know which resources are available in the embedded system (i.e. the hardware description: I/O devices, memory, etc).

There are two ways to provide this information to the Kernel:

- Hardcode it into the Kernel binary code. Each modification to the hardware definition requires recompiling the source code.
- Provide it to the Kernel when the bootloader uses a binary file, the device tree blob.

A device tree blob (DTB) file is produced from a device tree source (DTS).

- A hardware definition can be changed more easily as only DTS recompilation is needed.
- Kernel recompilation is not needed upon changes to the hardware definition. This is a big time saver.

Device Trees

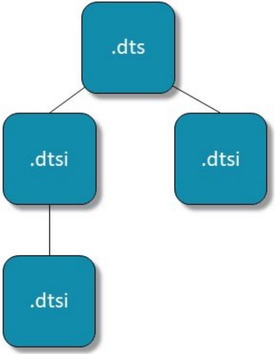
In Arm architecture, all device tree source files are now located in either `arch/arm/boot/dts` or `arch/arm64/boot/dts`.

- `.dts` files for board-level definitions
- `.dtsi` files for included files

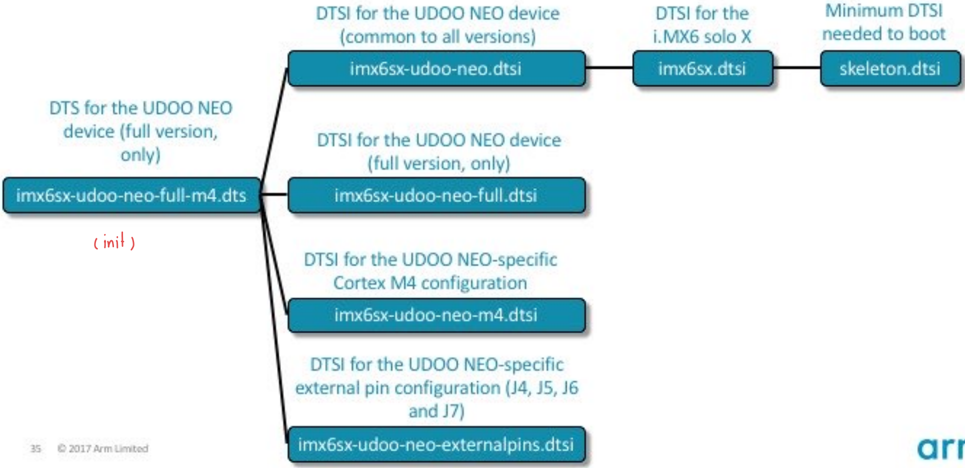
A tool, the device tree compiler, compiles the source into a binary form: the device tree blob (DTB).

- The DTB is loaded by the bootloader and parsed by the kernel at boot time.

Device tree files are not monolithic. They can be split in several files, including each other.



Device Tree Example for the UDOO NEO



Device Tree Syntax

```
/ {
 node0: node@0 {
   a-string-property = ”A string”;
   a-string-list-property = “string ”, “string 2”;
   a-byte-data-property = [0x12 0x23 0x45 0x56];

   child-node@0 {
     a-reference-to-something = <&node1>;
   };

   child-node@1 {
   };
 };
 node1: node@1 {
   an-empty-property;
   a-cell-property = <1 2 3 4>;

   child-node@0 {
   };
 };
};
```

Node name

Node label

Property value as string

Property name

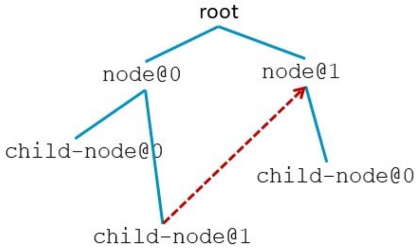
Property value as sequence of bytes

Reference to another node

Address of the node

Property value as sequence of 32-bit integers

Representation of the device tree



arm

Device Tree Content

Under the root of the device tree, we can find:

A `cpus` node, which sub nodes describe each CPU in the system

A `memory` node, which defines the location and size of the RAM

A `chosen` node, which is used to pass parameters to kernel (the kernel command line) at boot time

An `aliases` node, to define shortcuts to certain nodes

One or more nodes defining the buses in the SoC

One or mode nodes defining on-board devices

```
/ {
    alias {};

    cpus {};

    apb@80000000 {
      apbh@80000000 {
        /* some devices */
      };
      apbx@80040000 {
        /* some devices */
      };
    };

    chosen {
      bootargs = "root=/dev/nfs";
    };
};
```


Device Tree Addressing

The following properties are used:

- `reg = <address1 length1 […] >`, which lists the address sets (each defined as starting address, length) assigned to the node
- `#address-cells = <num of addresses>`, which states the number of address sets for the node
- `#size-cells=<num of size cells>`, which states the number of size for each set

Note:

- Every node in the tree that represents a device is required to have the `compatible` property.
- `compatible` is the key Linux uses to decide which device driver to bind to a device.

```
/ {
    cpus {
      #address-cells = <1>;
      #size-cells = <0>;
      cpu@0 {
        compatible = "arm,cortex-a9";
        reg = <0>;
      };
      cpu@1 {
        compatible = "arm,cortex-a9";
        reg = <1>;
      };
    };
};
```


arm

Device Tree Addressing

CPU addressing

- Each CPU is associated with a unique ID only.
- `#size-cells=<0>`, always

Memory mapped devices

- Typically defined by one 32-bit based address, and one 32-bit length
- `#address-cells=<1>`
- `#size-cells=<1>`

```
cpus {
  #address-cells = <1>;
  #size-cells = <0>;
  cpu@0 {
    compatible = "arm,cortex-a9";
    reg = <0>;
  };
};
```

```
/{
  #address-cells = <1>;
  #size-cells = <1>;
  gpio1: gpio@0209c000 {
    compatible = "fsl,imx6sx-gpio",
                 "fsl,imx35-gpio";
    reg = <0x0209c000 0x4000>;
  };
};
```


arm

Device Tree Addressing

External bus with chip select line

- Typically, `address-cells` uses 2 cells for the address value: one for the chip select number and one for the offset from the base of the chip select.
- `#address-cells=<2>`
- The length field remains as a single cell.
- `#size-cells=<1>`
- The mapping between bus addressing and CPU addressing is defined by the `ranges` property.

Bus address

```
external-bus {
  #address-cells = <2>
  #size-cells = <1>;

  ranges = <0 0  0x10100000   0x10000
            1 0  0x10160000
0x10000>;
  ethernet@0,0 {
    compatible = "smc,smc91c111";
    reg = <0 0 0x1000>;
  };
  i2c@1,0 {
    compatible = "acme,a1234-i2c-bus";
    reg = <1 0 0x1000>;
     #address-cells = <1>
     #size-cells = <0>;
    rtc@58 {
      compatible = "maxim,ds1338";
      reg = <58>;
    };
  };
};
```

Corresponding CPU address and range

Summary

Linux architecture

Device trees

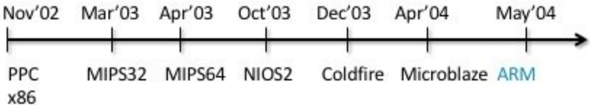
The U-BOOT bootloader

UDOO NEO boot process

The U-Boot Bootloader

Very popular bootloader among embedded system developers

Historic perspective



Today, it is the de-facto standard among embedded systems.

The U-Boot Bootloader

U-Boot is a first and second stage bootloader

U-Boot architecture is made of two halves:

1st half

- Written mostly in Assembly code
- It runs from the CPU on-chip memory (e.g., on-chip static RAM).
- It initializes the CPU RAM memory controller and relocates itself in off-chip RAM Memory.

2nd half

- Written mostly in C code
- It implements a command-line human-machine interface with scripting capabilities.
- It initializes the minimum set of peripherals to load the device tree Blob, the Linux Kernel, and possibly, the Initial RAM disk to RAM Memory.
- It starts the execution of the Linux Kernel.

The U-Boot Bootloader

Group 1

Processor-dependent files:

- Specific to the CPU that will run U-Boot (e.g. CPU 1, CPU 2, CPU n)

Group 2

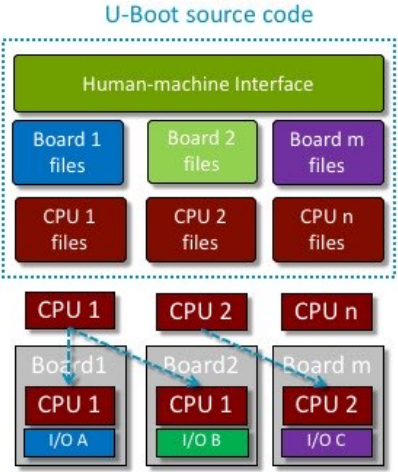
Board-dependent files:

- Specific for the boards hosting the above CPU, which may have different sets of I/O (.e.g, Board 1, hosting CPU 1, and I/O A versus Board 2, hosting CPU 1, and I/O B)

Group 3

General-purpose files:

- Suitable for all the boards/CPUs
- Implement the human-machine interface and the scripting feature of U-Boot



arm

Summary

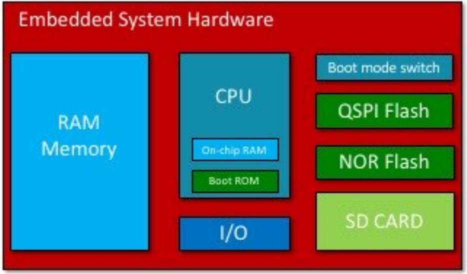
Linux architecture

Device trees

The U-BOOT bootloader

UDOO NEO Boot process

UDOO NEO Boot Process



Modern CPUs are capable of booting from multiple sources:

- ROM memory
- Parallel I/O Flash memory (e.g. NOR Flash)
- Serial I/O Flash memory (e.g. QSPI Flash)
- SD Card

An on-chip firmware (stored in on-chip Boot ROM) and I/O configuration (boot mode switch) tell the CPU where to boot from.

This firmware is know as 1st stage bootloader.

Bootloaders such as U-Boot are known as 2nd stage bootloaders.

arm

An Example: NXP i.MX6 System-on-Chip

The Internal Memory is composed of:

read only memory

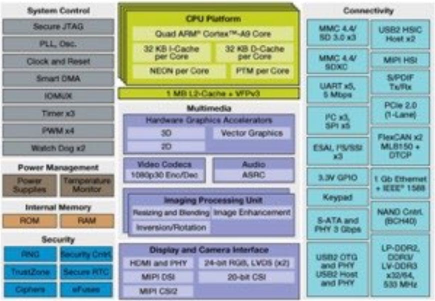
- ROM Memory, storing the 1st stage bootloader responsible for reading the boot mode switch, loading the 2nd stage bootloader in the on-chip RAM memory, and to start running it

access memory

- RAM Memory, to host the 2nd stage bootloader during execution of its 1st half

At power-up:

- The 1st stage bootloader decides where to boot from, loads the 2nd stage bootloader to internal RAM, and runs it.
- The 1st half of 2nd stage bootloader initializes the on-chip RAM controller, copies its 2nd half to external RAM memory, and executes it.



arm

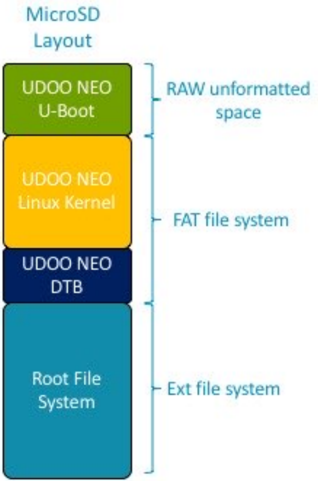
UDOO NEO Boot Process

At power-up, 1st stage bootloader (running from i.MX6 on-chip ROM) loads the U-Boot from MicroSD, stores it into i.MX6 internal RAM, then executes U-Boot 1st half.

U-Boot 1st half (running from i.MX6 internal RAM) initializes the i.MX6 RAM controller, copies the 2nd half to external RAM memory, and executes it.

U-Boot 2nd half (running from external RAM memory) loads from the boot partition of the MicroSD the Linux Kernel and the device tree Blob (DTB), stores them to external RAM memory, and then starts running Linux Kernel.

Linux Kernel starts executing, mounting the second partition of the MicroSD as root file system.



arm