① Embedded Linux - Introduction

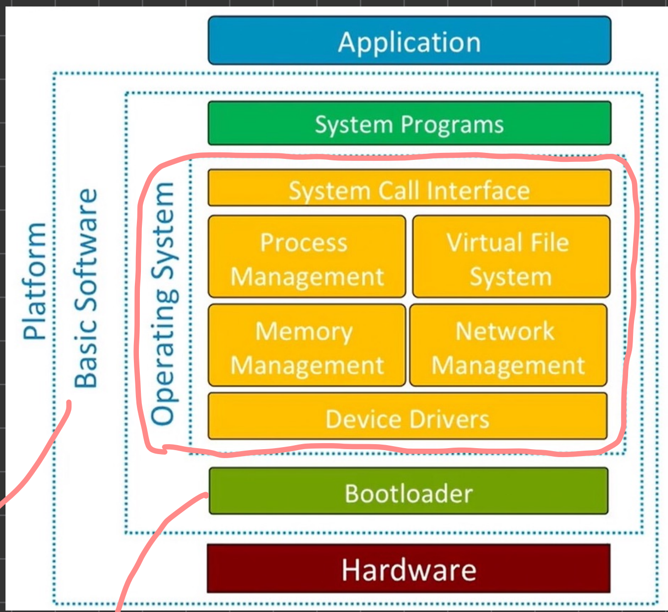
What is Embedded system

↳ special-purpose computer designed for specific app

Embedded System Components

แบ่งเป็น 2 main part

- Application
  - Software ที่ service ที่จำเป็นต่อ user
- Platform
  - hardware + basic software ที่คอยให้ service พื้นฐาน



Bootloader → programs ที่ต้อง execute ก่อน เพื่อเตรียม initialize hardware และ execution ของ OS

Basic software

- เข้าถึง resource ผ่าน system program เช่น ls
- และการเข้าถึง resource โดย OS เป็นคนจัดการ hardware

  เช่น CPU, RTOS, device driver management

OS สำหรับ Embedded System

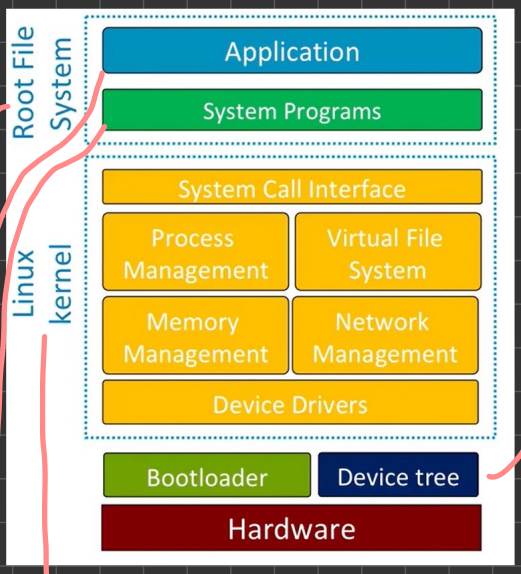
↳ มีหลายแบบขึ้นกับจุดประสงค์

Linux in Embedded system

เช่น mobile phone / In-flight entertainment system
electronics board that display the msg
gas station pump

ทำไม?
- open source
- strong community
- flexibility / adaptability
  - support new hardware เช่น SoC

2 Linux-based Embedded system component stack



Device tree : tree data structure
ที่มี node ที่แทน physical device
ใน hardware ที่ Linux kernel ใช้
ในการ initialize device driver

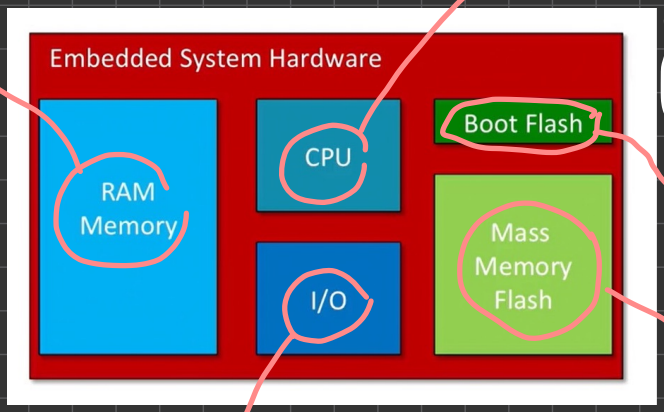
linux kernel main component!
คือ os code ที่มี service ที่จำเป็น
hardware resource

user program
งานของ user ที่ใช้ในการเข้าถึง os service

Application
software ที่ใช้ทำงานตามที่ delivered ให้ user

Root filesystem
ที่เก็บ linux kernel configuration files / the system program / app

Reference Hardware Model



Random access memory

volatile mem เช่น data/mem

central processing unit

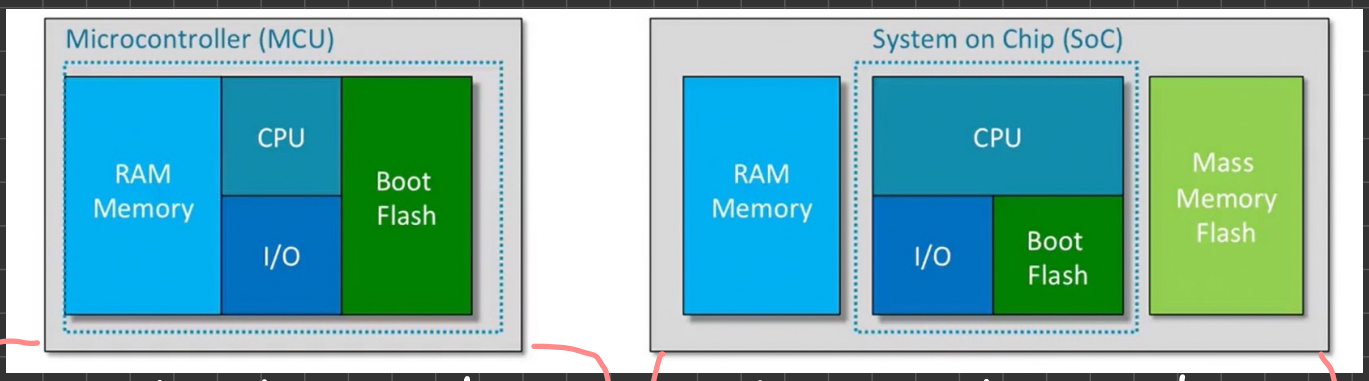
processor ของ software เช่น arithmetic logic

non-volatile mem ขนาดเล็ก

no-volatile mem ขนาดใหญ่

communicate between user and system

implement

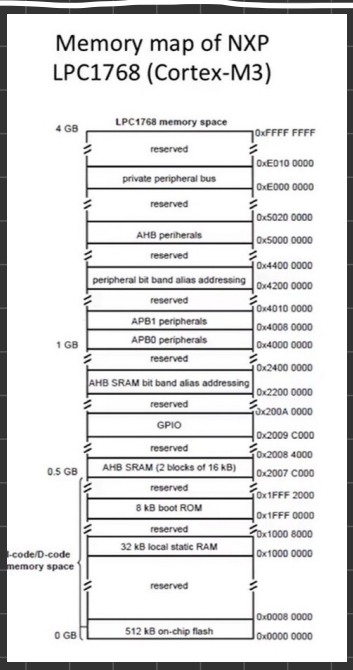


microcontroller-base implement

single device host of the reference model components

System-On-Chip implement

most of reference model components are discrete เช่น มี CPU รวมอยู่กับบางตัวด้วย



- CPU generates 2^N address ที่ต่างกัน ($0 \rightarrow 2^N-1$) โดย N คือ จำนวน bit ของ address bus
- แต่ละ device จะมี address ของมัน

Bootloader

- ทำหน้าที่ boot up machine ในขั้นแรกสุด เพื่อให้ software run
- processor ต้องรู้ว่า → where the software is located
  - → How to get access to the software location
  - → where stack is located

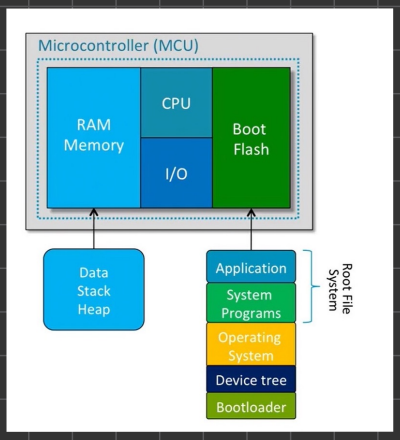
ในขณะที่เปิดโปรแกรม the program counter จะถูก set เป็น default known value เพื่อจะ execute reset vector คือ pointer ที่ชี้ไปที่ address ที่เก็บ instruction แรกที่จะ run ของ firmware

register ที่เก็บ address ของ instruction ต่อไป เพื่อ run

- bootloader คือ สิ่งแรกที่ run เพื่อจะเป็นส่วนที่จัดการในเรื่องข้อมูลต่างๆที่ต้องใช้ processor เช่น startup software location, how to access it, where stack located เพราะอาจมีที่อื่นๆขึ้นอยู่กับ system's architecture

---

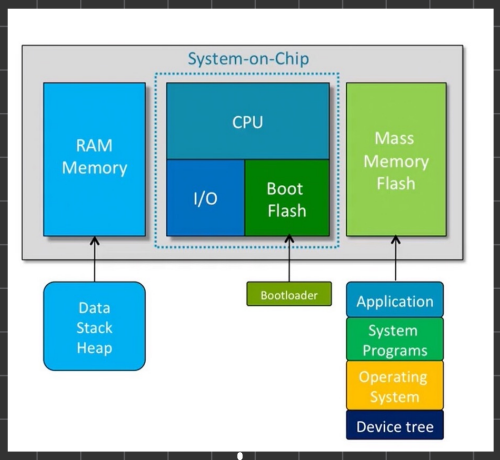
Possible scenarios



scenarios 1

Scenario 1, typical of microcontrollers

- All the sw (bootloader + device tree + operating system + root filesystem) is stored in persistent storage (boot flash) embedded in the microcontroller.
- All the sw is executed from the persistent storage.
- The CPU reset vector is located in the boot flash.
- The RAM Memory is embedded in the microcontroller and is used for data, stack and heap only.

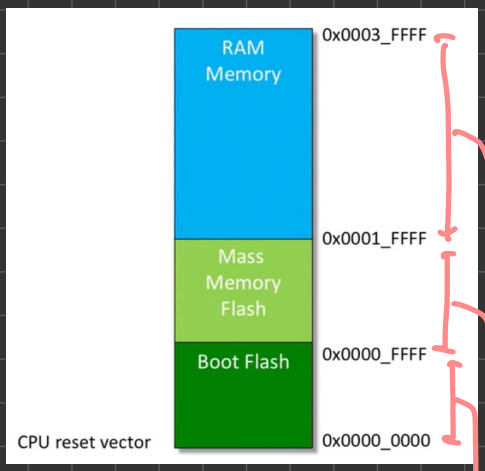


scenarios 2

Scenario 2, typical of System-on-Chip

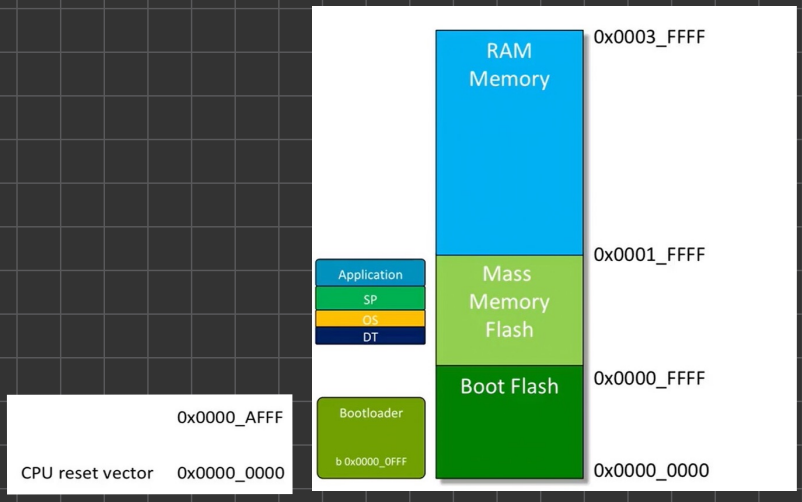
- The bootloader is stored into the boot flash.
- The CPU reset vector is located in the boot flash.
- The root filesystem, operating systems, and device tree are stored in the mass memory flash and loaded in RAM memory by the bootloader.
- The RAM memory is external to the SoC. It will store the operating system + application software, root filesystem (if configured as RAM disk), data, stack, and heap.

example of bootloader operation



พิจารณา scenario 2 ด้วย SoC ที่มี memory map ตามนี้

- RAM mem 512k byte (128 kword) แต่ละช่องเก็บ 32 byte long (0X0002_0000 - 0X0003_FFFF)
- mass memory flash 256 kbytes (0X0001_0000 - 0X0001_FFFF)
- boot flash 256 kbytes (0X0C0_CCCC - 0X0C00_FFFF)



ตอนก่อน power up

- boot flash จะถูก load bootloader ไว้เพื่อ
  โดย first bootloader instruction คือ 0x0000_0FFFF
  เช่น last bootloader instruction คือ 0x0080_AFFF

- เช่น mass memory flash จะ load ตัวไหน
  1. Device tree (DT)
  2. Operating system (OS)
  3. System programs (SP)
  4. Application

ตอนแรก power up

- CPU เริ่ม execute software จาก reset vector
  - ↳ มันจะกระโดดไปที่ first instruction ของ bootloader
  - ↳ และมันจะรัน bootloader software จาก boot flash memory

ตอนระหว่าง bootstrap

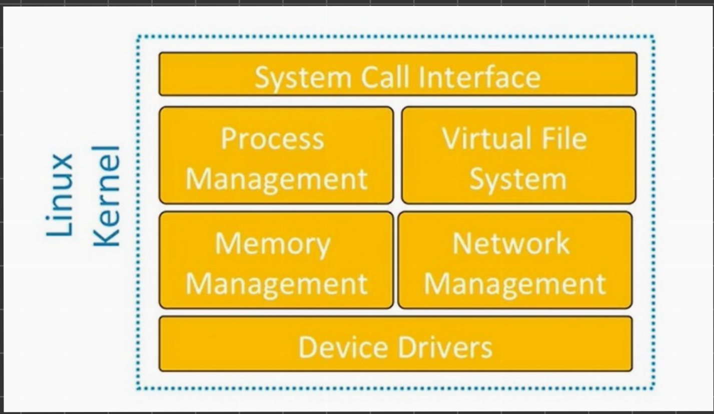
- CPU จะ execute bootloader ที่
  - ↳ initialize CPU RAM memory controller
  - ↳ set up the CPU registers เพื่อการ mapping stack และ heap ไปที่ RAM memory
  - ↳ ก๊อปปี้ device tree, operating system, system programs, applications ไปที่ RAM

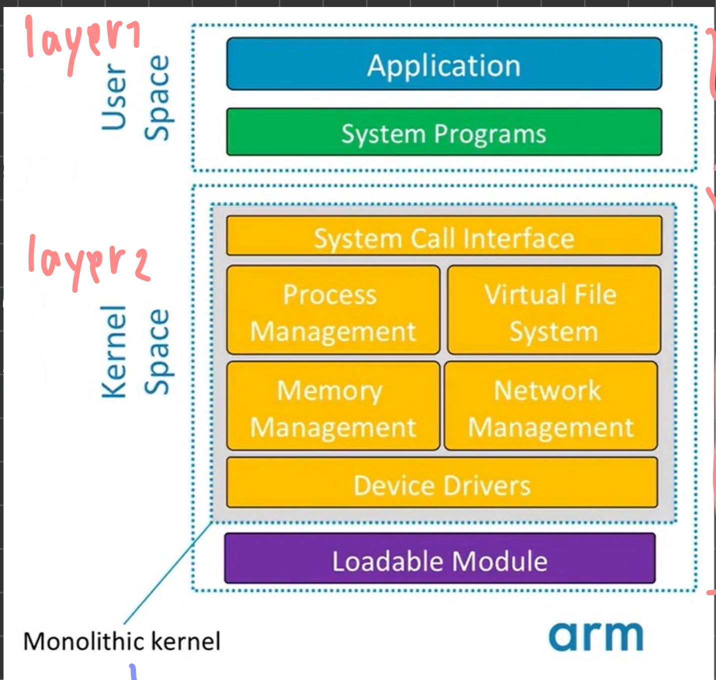
ตอนจบ bootstrap

- bootloader จะโดดไปที่ first instruction ของ OS
- CPU จะ execute OS software ที่เอาไว้ที่
  1. setting-up execution environment ของ app
  2. เริ่ม application execution

Kernel

the linux kernel ที่เป็น software ที่มีหน้าที่ในการ manage embedded system's hardware resources ให้ optimal และก็มีหน้าที่ offer services ต่างๆ





Linux kernel มีหลาย layer ที่เอามาจาก layered operating system architecture

→ high level part เพราะมี code ที่ไม่ได้ run

ที่ๆ kernel execute service for the system

2 layers นี้มี different address space

ถ้าเป็น basic services จะ delivered โดย single executable (monolithic)

แต่ services can be extended at run-time through loadable kernel modules

ข้อดี

- มีการแบ่งแยกระหว่าง app/system programs และ kernel
- bug จาก user space จะไม่ส่งผลต่อ kernel

ข้อเสีย

- bug ใน kernel component เพียงเล็กน้อย จะส่งผล system ทั้งหมด

Device Tree

I/O device, memory, etc

- ในตอนที่ kernel จะ manage hardware resources
  kernel จะต้องรู้ว่า resources ไหนบ้างอยู่ ใน embedded system

มี 2 วิธีในการทำให้ kernel รู้ information เหล่านี้ คือ

① hardcode! ลงไป ใน kernel binary code
แต่ทุกครั้งที่มีการแก้ไข hardware definition จะต้อง compile source code ใหม่

② provide it to the kernel ในตอนที่ bootloader ใช้ binary file (device tree blob)

device tree blob (DTB) file มาจาก device tree source (DTS)

↳ hardware definition จะถูกแก้ไขได้ง่ายมาก โดยใช้แค่ DTS recompilation

↳ ไม่ต้องใช้ kernel recompilation ตอนที่ hardware definition มีการเปลี่ยนแปลง

System programs

คือ executable ของ command line โดยที่พวกนี้จะเป็นส่วนช่วย environment ให้ program development และ execution

They can be divided into:

- File manipulation
- Status information
- File modification
- Programming language support
- Program loading and execution
- Communications
- Application programs

เช่น ls, cat, cd

Application

คือ software ที่ใช้เพื่อตอบสนอง user service

Examples can be found in many different products:

- Network Attached Storage (NAS)
- Network router
- In-vehicle infotainment
- Specialized lab equipment
- And more

Root Filesystem

ตอนเรา startup linux kernel ต้องใช้ file system (root filesystem) ซึ่งจะมี configuration file ที่ใช้ในการ จัดเตรียม execution environment สำหรับ application เพราะคือ first user-level process (init)

The root filesystem can be:

- ส่วนหนึ่งของ RAM ที่เป็นเหมือน filesystem (initial RAM disk) (initrd)
  ถ้า embedded ไม่ได้ จำเป็นต้องใช้ ในการเก็บข้อมูลการ ระหว่าง operation นั้นๆ
- A persistent (ถาวร) storage ใน embedded system
- A persistent storage accessed over the network

Typical Layout of the Root Filesystem

```
/                                   # Disk root
        /bin                        # Repository for binary files
        /lib                        # Repository for library files
        /dev                        # Repository for device files
                console c 5 1       # Console device file
                null c 1 3          # Null device file
                zero c 1 5          # All-zero device file
                tty c 5 0           # Serial console device file
                tty0 c 4 0          # Serial terminal device file
                tty1 c 4 1          #
                tty2 c 4 2          #
                tty3 c 4 3          #
                tty4 c 4 4          #
                tty5 c 4 5          #
        /etc                        # Repository for config files
                inittab             # The inittab
                /init.d             # Repository for init config files
                        rcS         # The script run at sysinit
        /proc                       # The /proc file system
        /sbin                       # Repository for accessory binary files
        /tmp                        # Repository for temporary files
        /var                        # Repository for optional config files
        /usr                        # Repository for user files
        /sys                        # Repository for system service files
        /media                      # Mount point for removable storage
```


arm

Quiz

Quiz 2

You have unlimited attempts.

Lecture 2:

1. "The bootloader is a piece of software responsible for..."

- ○ Providing all the services to manage the hardware resources
- ✔ Setting-up the hardware to run the operating system
- ○ Implementing the functionalities to be delivered to the embedded system user
- ○ Storing the Linux Kernel Configuration files, the system programs, and the application

Your answer is correct.

2. "In the 'Reference Hardware Model', which component is responsible for volatile memory?"

- ○ Boot Flash
- ○ Mass Memory Flash
- ✔ RAM
- ○ CPU

Your answer is correct.

3. "The Linux kernel is split into..."

- ✔ Two layers: User space and kernel space
- ○ Two layers: Developer space and user space
- ○ Three layers: OS, hardware, and software

Your answer is correct.

4. "Which of the following are methods for informing the kernel of which resources are available in the embedded system?"

- ☐ Get the kernel to look it up online
- ✔ Using a device tree blob
- ☐ Look it up in a memory map
- ✔ Hardcode it into the kernel binary code

Your answer is correct.

5. "What produces a device tree blob (DTB) file?"

- ○ Device Tree Origin
- ✔ Device Tree Source
- ○ Device Tree Generation
- ○ Device Tree

Your answer is correct.

6. "In regard to the bootloader – at power-up, the program counter is set to a default value, known as the..."

- ○ Default vector
- ○ Initial vector
- ○ Primary vector
- ✔ Reset vector

Your answer is correct.

7. "A bootloader operation carried out at the power-up stage is..."

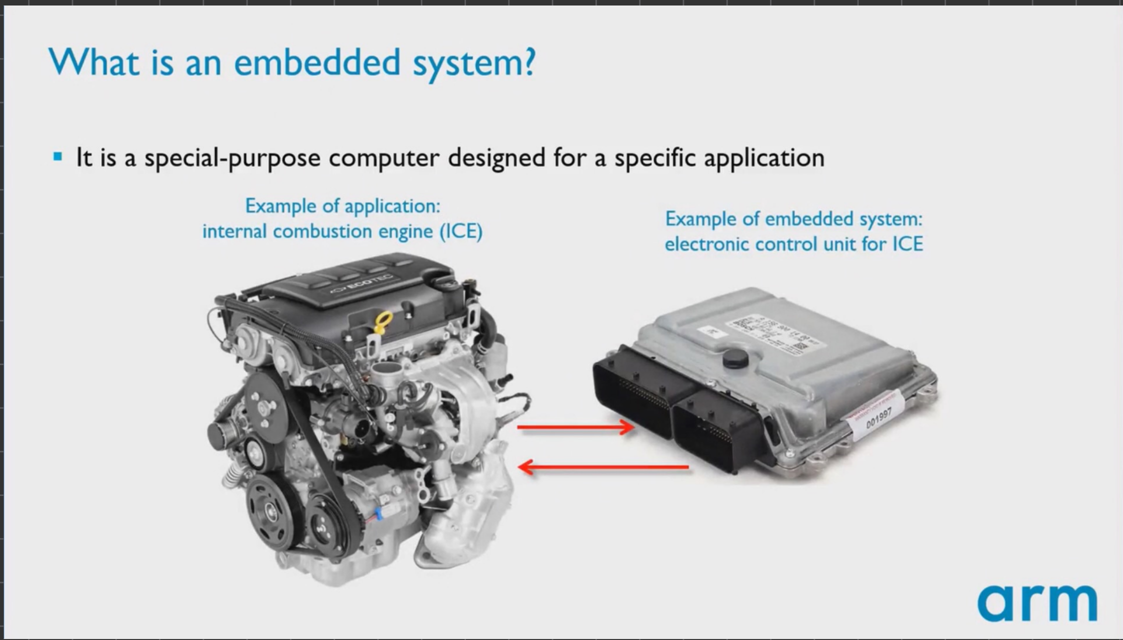
- ✔ Begins executing software from the reset vector
- ○ Preloading the boot flash with the bootloader
- ○ Jumping to the first instruction of the operating system
- ○ Preloading the mass memory flash with the device tree

Your answer is correct.

Yocto Project → open source custom linux-based distribution development tool

Poky → reference distribution of the Yocto Project

Raspberry Pi → single board computer used to host the custom linux-based distribution



Linux Based Embeded System

มี 6 ส่วนหลักๆ คือ

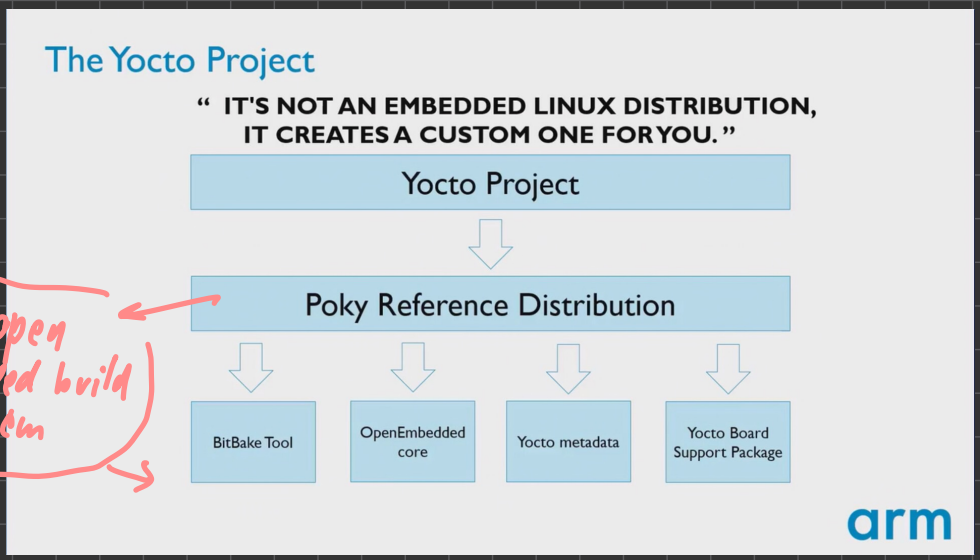
1. Bootloader → เก็บใน ROM ควบคุมสิ่งที่เกิดขึ้นใน system ตอนที่เปิดขึ้น
2. Device tree → script ของ physical hardware ที่ available ใน system
3. Linux kernel → software ที่ manage ระหว่าง computer และ hardware resource
4. System programs → set ของ utilities ที่ให้ app access resource ของ system
5. Application → software ที่ implement fn ของ system ให้ user ใช้
6. Root filesystem → ที่ๆ ไว้เก็บ (โปรแกรมพื้นฐาน/application configuration file

LAB

ใน lab จะ boot ด้วย linux system และ bootloader ด้วยวิธีการ flash ลงใน micro SD card

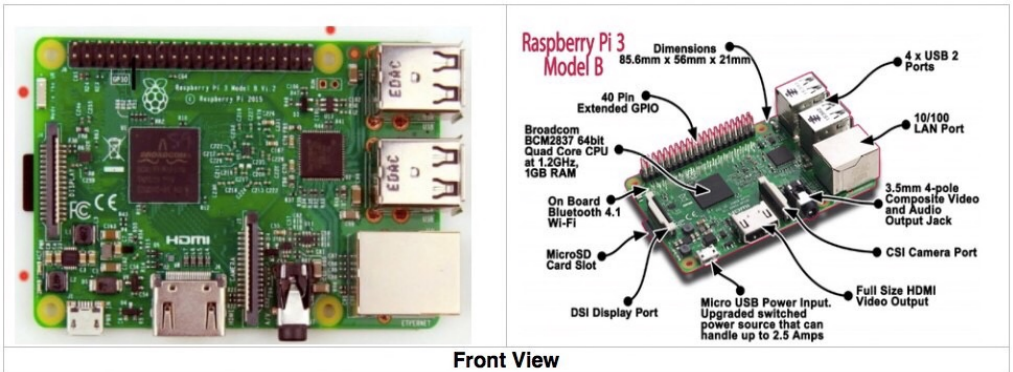
---

The Yocto Project provide platform to create custom Linux space system for any embedded linux hardware



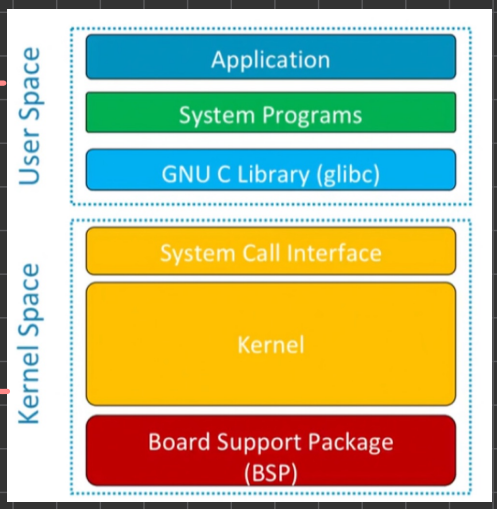
- Raspi 3 พัฒนามาจาก Raspi 2 โดย upgrade Arm cores เป็น Cortex-A53 และเพิ่ม onboard single-band 2.4 GHz-only wireless chipset

The Raspberry Pi 3 measures the same 85.60mm x 53.98mm x 17mm, with a little overlap for the SD card and connectors that project over the edges. The SoC is a Broadcom BCM2837. This contains a quad-core Cortex-A53 running at 1.2GHz and a Videocore 4 GPU.



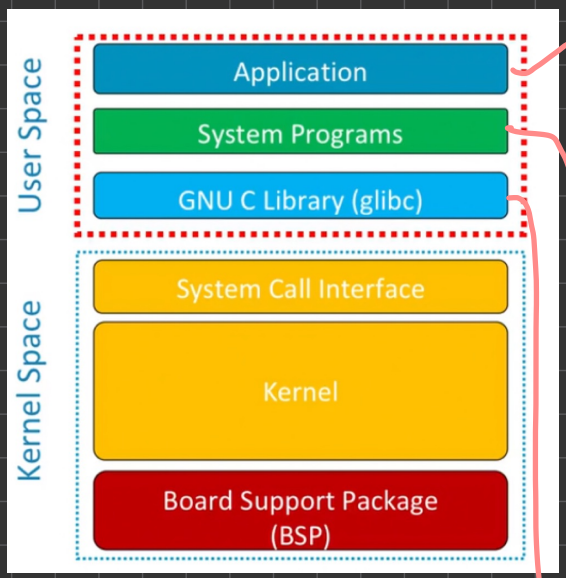
Front View

③ Anatomy of Linux-based system



ใช้ system calls ผ่าน

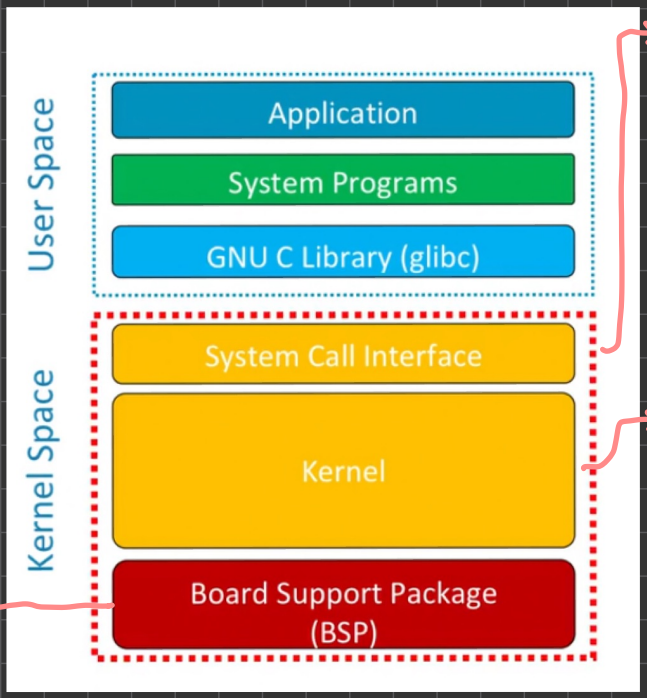
แบ่งเป็น 2 ส่วน คือ user space และ kernel space ซึ่ง independent และ isolate กัน



Appilation → implementing the functionalities ใน embedded system user

system programs → user-friendly utilities ในการเข้าถึง operating system service

GNU C library (glibc) interface ระหว่าง user space กับ kernel space



System call interface

เป็น entry point ในการเข้าถึง service ของ kernel

(process management, memory management)

Kernel

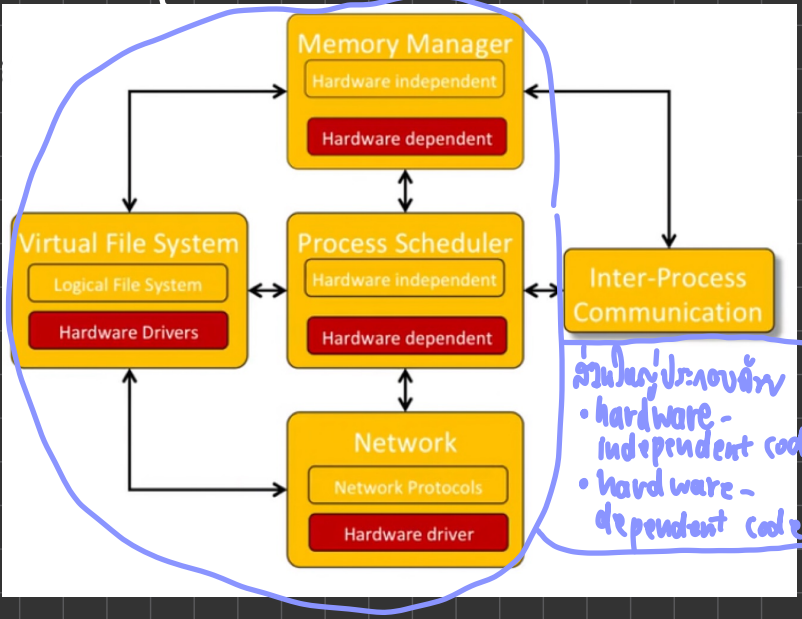
- architecture-independent operating system code
- it implements the hardware-agnostic ??? services of the operating system เช่น process scheduler

Board Support Package (BSP)

- architecture-dependant operating system code
- เป็นส่วนที่มี hardware specific services ของ operating system เช่น context switch

---

Conceptual view of the kernel

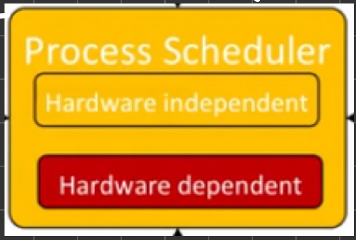


แบ่งเป็น 5 ส่วน

1. Process scheduler
2. memory manager
3. virtual file system
4. inter-process communication (IPC)
5. network

ดูแต่ละส่วน

Process Scheduler → ตัดสินว่า process ไหนจะรันตอนเวลาไหน



main fn

- ให้ process สร้าง new copy ของตัวเอง
- ทำ cpu scheduling policy และ context switch
- รับ → interrupt → พาไป appropriate kernel subsystem
- ส่ง signal ไปให้ user process
- จัดการ timer ของ hardware
- cleans up process resources ในตอนที่ process execute เสร็จ
- Provide support สำหรับ loadable kernel module

external interface

- system call ที่ใช้คู่กับ user space (เช่น fork())
- inter-kernel interface ที่ใช้คู่กับ kernel space (เช่น create_module())

scheduler จะถูกเรียกโดยตรงจะ system call เช่น sleep

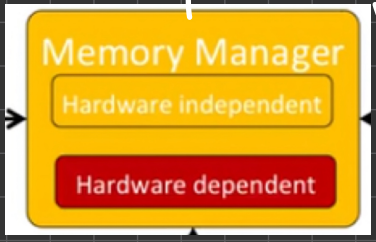
scheduler จะถูกเรียกทางอ้อมหลังจากทุกๆ system call

scheduler จะถูกเรียกหลังจากทุกๆ slow interrupt

interrupt มี 2 แบบ

- slow interrupt ทั่วไป เช่น coming from a disk driver
- fast interrupt ตอบสนอง operation ที่ไว เช่น processing a keyboard input

Memory manager



หน้าที่ handle

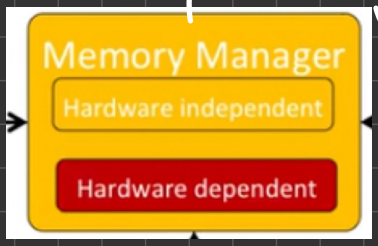
- large address space
  - ↳ user process สามารถอ้างอิง RAM memory ได้มากกว่าที่มีอยู่จริงๆ
- Protection
  - ↳ คือ memory ของ process ต้อง private แล: ไม่สามารถดูอ่านหรือแก้ไขข้อมูลprocess อื่นได้
  - ↳ ป้องกัน process จาก overwriting code แล: read-only-data
- memory mapping
  - ↳ ให้ process map File ใน virtual memory แล: เข้าถึง files as memory
- แบ่งทุก process เข้าถึง physical memory ได้อย่าง fair ๆ
- จัดการ shared memory · ให้ process shared บางส่วนของ memory ได้ (เช่น executable code มักใช้ share ร่วมกันใน process)

ใช้ Memory Management Unit (MMU) ในการ map virtual address กับ physical address เช่น processor running linux, MMU is mandatory

ประโยชน์คือ

- process สามารถเคลื่อนที่ภายใน physical memory ได้อย่างอิสระ virtual address ไม่เปลี่ยน
- physical memory เดียวกันสามารถ shared ร่วมกันได้ระหว่าง process ที่ต่างกัน

memory manager (ต่อ)



หน้าที่ handle

- swap process memory ออกไปที่ paging file ตอนที่ไม่ได้ใช้
  - ถ้าพื้น process ที่ใช้ memory มากกว่า physical memory มีอาจทำให้ execute ได้
- kswapd kernel-space process (หรือ daemon) จะใช้ทำถ้าพื้
  - เลือกว่าจะ physical memory pages ที่ไม่ได้ถูก referenced เมื่อเร็วๆนี้
  - pages พื้นๆ จะถูกเขียนออกจาก physical memory และ stored ใน paging file
- MMU ตอนที่ user process เข้าถึง memory address ที่ไม่ได้ map ไว้ตรงไปที่ physical memory location
- MMU จะแจ้งไปที่ linux kernel ว่า page fault และ memory manager subsystem จะพยายามแก้ page fault นี้
  - ถ้า page นั้นถูก swap ไปที่ paging file มันจะถูก swap กลับมา
  - ถ้า memory manager ตรวจเข้าถึง memory ที่ไม่ถูกต้อง มันจะแจ้งเตือน event ไปที่ user process ด้วย signal และถ้า process ไม่ handle signal นี้ มันจะ terminate

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

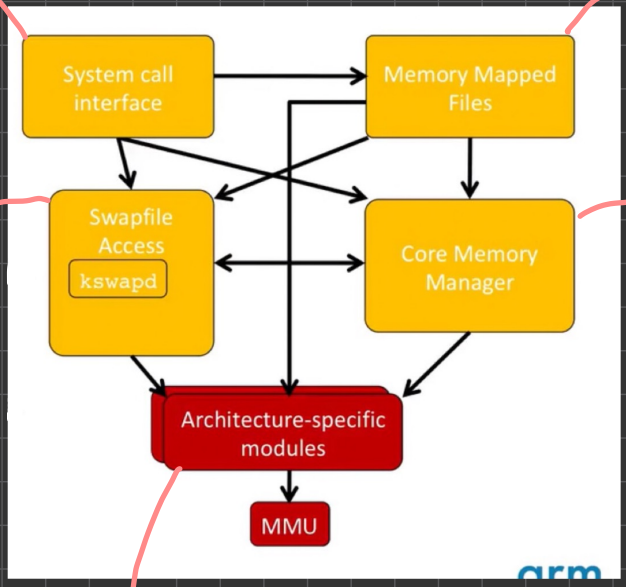
Memory Management Architecture

เป็นจุดที่ memory manager service รู้จัก user

เป็นแหล่ง memory file mapping algorithm

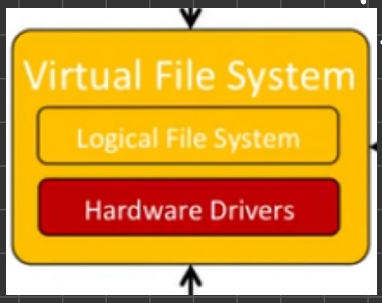
ควบคุมการเข้าถึง paging file

เป็นจุดที่ memory allocation algorithm



เป็นจุดจัดการ hardware-specific operations ที่ต้องเกี่ยวข้องกับ memory management เช่น การเข้าถึง MMU

Virtual file system



external interface:
- system-call interface: Posix standard ที่ใช้ (เช่น open / close / read / write)
- Inter-kernel interface: i-node interface ของ file interface

หน้าที่ handle
- จัดการ multiple hardware device: เป็นแบบ uniform access? กับ hardware device
- multiple logical file systems: supports many different logical organizations ของ information ใน storage media
- multiple executable format: supports different executable file format (เช่น a.out, ELF)
- homogeneity (เชื่อเดียวกัน): present interface เดียว สำหรับทุก logical file systems เช่น hardware device
- Performance: provide high speed access to file
- safety: enforce policy เพื่อให้มั่นใจว่าจะไม่ lose หรือ corrupt data
- security: enforce policy ต้องใช้ access file กับข้างของ user เท่านั้น เช่น จำกัด total file size ด้วย quotas สำหรับ user

i-node

- จะเก็บ information ทั้งหมดของ file ยกเว้นสิ่งเดียว คือ data ของ file นั้น
- ในตอนที่ file ถูกสร้างขึ้นจะถูก assigned ชื่อ เป็น unique i-node number
  a unique integer number
- ในตอนที่ file ถูก access
  - ↳ each file is associated with unique i-node number
  - ↳ i-node number จะถูกใช้ในการเข้าถึง data structure ที่เก็บ information ของ file นั้น ๆ access

```
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

arm

File 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

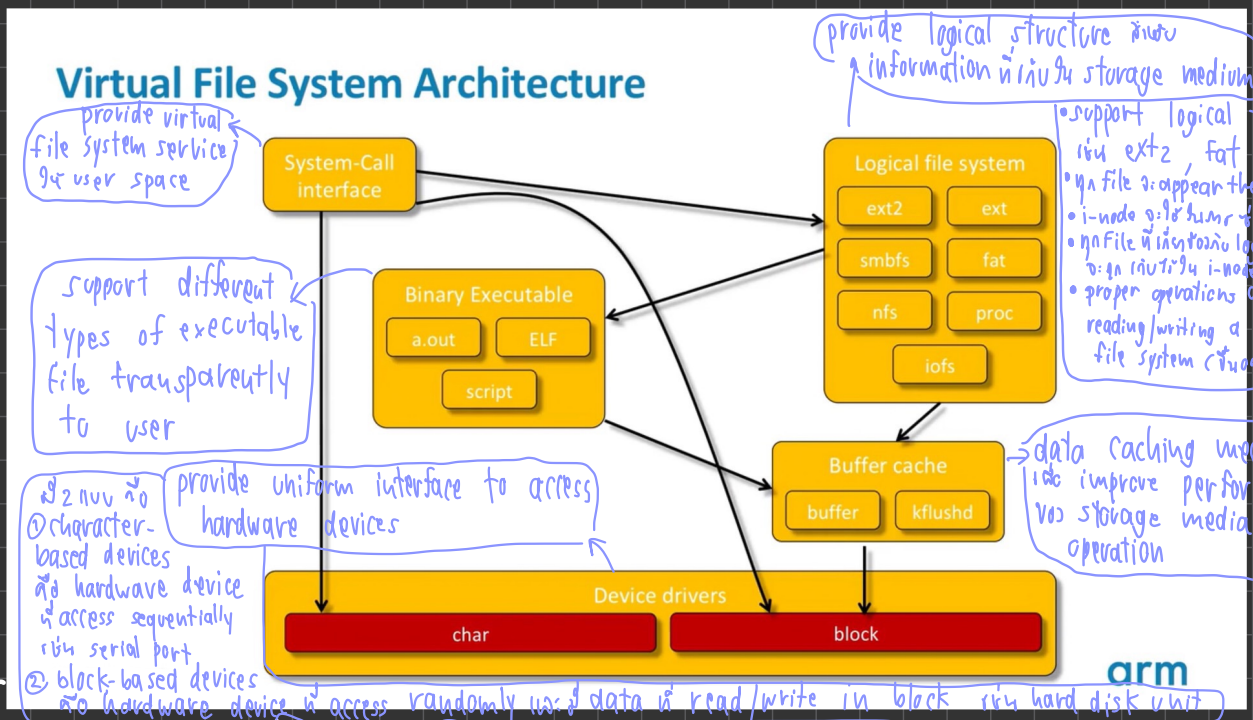
`readdir()`: reads the files that are pointed to by a directory file

`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

arm



provide virtual file system service ผ่าน user space

provide logical structure ของ information ที่เก็บอยู่ storage medium

- support logical file system หลาย type เช่น ext2, fat
- ทำให้ file ต่างๆ appear the same to the user
- i-node เก็บข้อมูล logical file system details
- ทำให้ file ที่เก็บอยู่บน logical file system type ต่างกัน ใช้วิธีผ่าน i-node
- proper operations are activated when reading/writing a file in a given logical file system (ขึ้นอยู่กับ information ใน i-node)

support different types of executable file transparently to user

data caching mechanism เพื่อ improve performance ของ storage media access operation

provide uniform interface to access hardware devices

มี 2 แบบ คือ
① character-based devices คือ hardware device ที่ access sequentially เช่น serial port
② block-based devices คือ hardware device ที่ access randomly และ data ที่ read/write in block เช่น hard disk unit

→ diagram ที่แสดง flow ของ virtual file system architecture

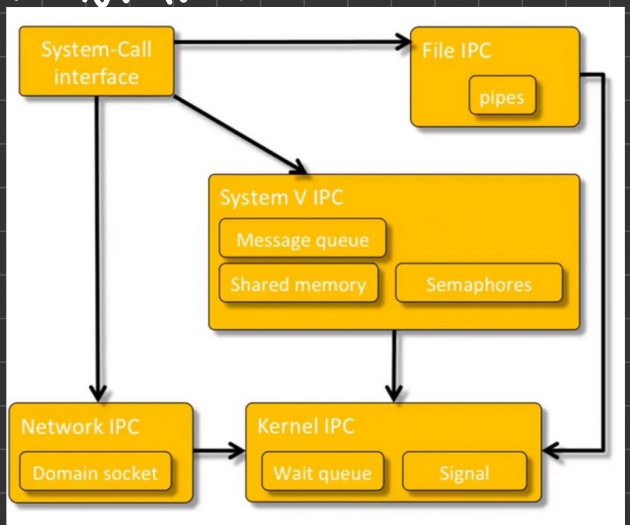
Device drivers use the file interface abstraction:

- 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

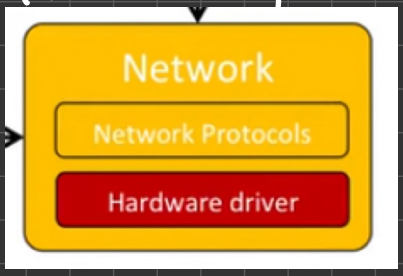
- provides mechanisms to processes for allowing
  - Resource sharing
  - synchronization
  - data exchange

diagram



- IPC provide to user-space as a system call interface
- IPC supports หลาย เช่น pipes, message queues, shared mem, semaphores, domain socket, wait queues, signal

Network



- provide support for network connectivity
  - → implement network protocol เช่น TCP/IP through hardware-independent code
  - → implement network card drivers through hardware-specific code

---

Device Tree

- manage hardware resources
- kernel ต้องรู้ว่า resources ไหน available ใน embedded system (เช่น hardware description : I/O devices, memory, etc)

ต้องรู้ยังไง provide this information to kernel

① hardcode ไว้ใน kernel binary code แล้วเวลาที่มีการ hardware definition ต้อง recompile source code

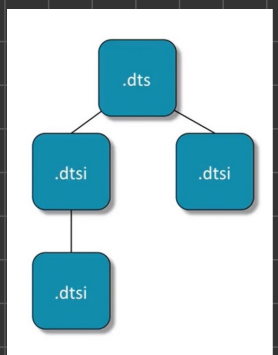
② provide it to the kernel when the bootloader uses a binary file, the device tree blob

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.

ใน Arm architecture, device tree source file จะอยู่ที่ arch/arm/boot/dts

- → .dts files จะเก็บ board-level definitions
- → .dtsi files จะเก็บ include file

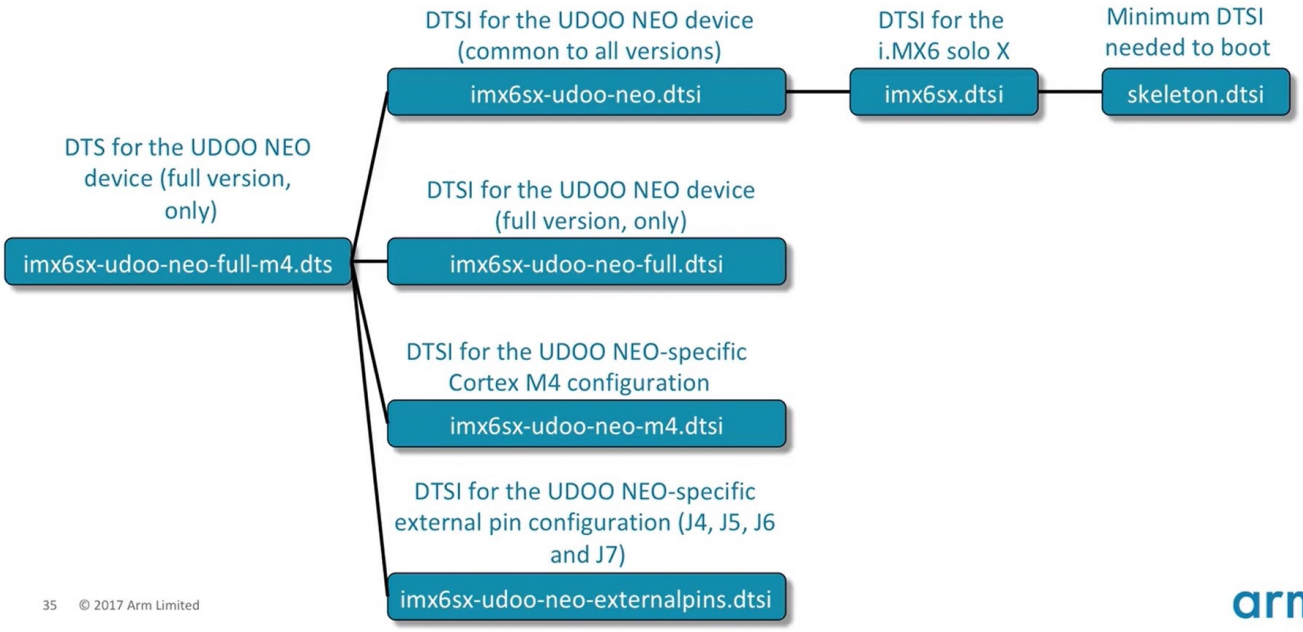


A tool, the device tree compiler, compiles the source into a binary form: the device tree blob (DTB).

- 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



arm

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 label

Node name

Property value as string

Property name

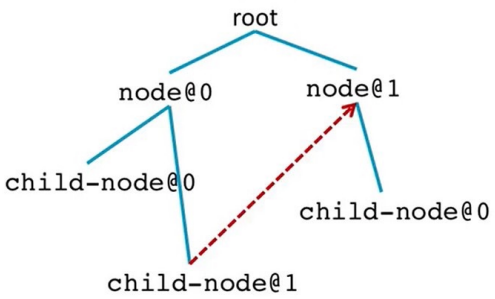
Property value as sequence of bytes

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

ไฟล์ root จะมี node หลายตัวเสมอ

```
/ {
    alias {};

    cpus {};

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

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

define shortcuts to contain node

เป็น sub nodes ที่ describe แต่ละ CPU ใน system

memory node, define location กับ size ของ RAM

chosen node, ใช้ pass parameters ให้ kernel (kernel command line) at boot time

เพื่อมีอีก 1 เพื่อแทน 1 node define buses in the SOC

เพื่อมีอีก 1 เพื่อแทน 1 node define on-board devices

Device Tree Addressing

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

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

- ทุก node ใน tree ที่ represents a device จะต้องมี required ชื่อว่า compatible property
- compatible คือ key ที่ linux ใช้ในการหา device driver ในแง่ของ bind กับ device

```
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>;
  };
};
```

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>

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>

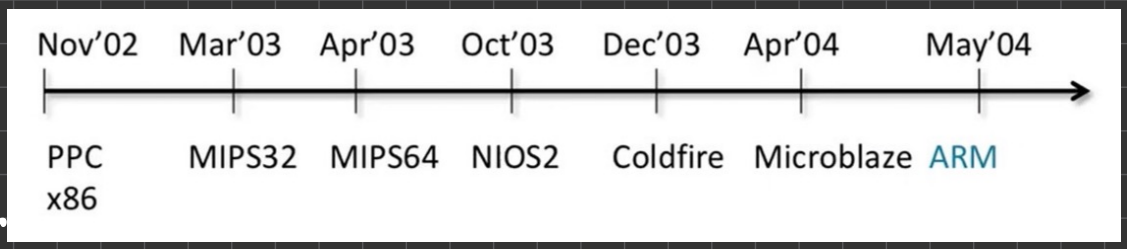
```
cpus {
  #address-cells = <1>;
  #size-cells = <0>;
  cpu@0 {
    compatible = "arm,cortex-a9";
    reg = <0>;
  };
};

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


arm

U-Boot Bootloader

เป็น bootloader ที่นิยมในหมู่ embedded system developers



↳ Historic perspective

ทำให้มันเป็น "de-facto standard" ใน embedded system

U-Boot architecture มี 2 ส่วนคือ

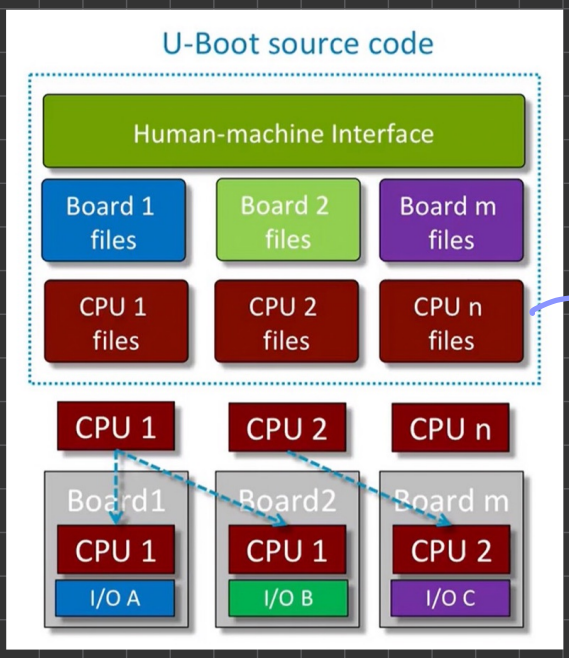
1st half

- ส่วนใหญ่จะเขียนด้วย Assembly code
- ทำงานกับ CPU on-chip memory เช่น on-chip static RAM
- มัน initializes the CPU RAM memory controller
- และ relocates ตัวมันเองใน off-chip RAM memory

2nd half

- ส่วนใหญ่จะเขียนด้วย 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.



แยก group of file ของ boot loader ได้เป็น

① processor dependent file

เช่น: CPU ที่จะ run U-Boot

เช่น CPU 1, CPU 2, CPU n

② board-dependent file

เช่น: boards hosting the above CPU ที่อาจมี set ของ I/O ที่ต่างกัน

เช่น board 1 → hosting CPU1, I/O A

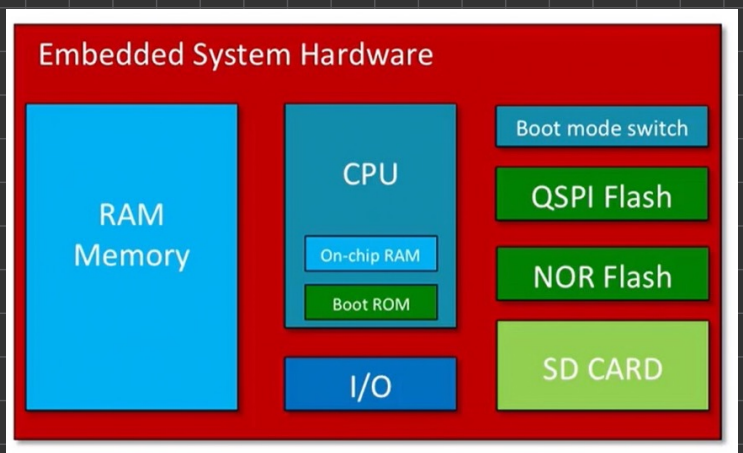
board2 → hosting CPU1, I/O A

③ general-purpose file

- เหมือนกันทุกบอร์ด/CPU
- implement the human-machine interface and the scripting feature of U-Boot

---

UDOO NEO Boot Process



on-chip firmware (เก็บใน on-chip Boot ROM) เช่น I/O configuration (boot mode switch) จะเป็นตัวบอก CPU ว่า boot มาจากไหน

CPU ปัจจุบันสามารถ boot ได้จาก หลาย sources:

- ROM memory
- Parallel I/O flash mem เช่น NOR Flash
- Serial I/O flash mem เช่น QSPI Flash
- SD card

firmware → 1st stage bootloader

bootloader เช่น U-Boot → 2nd stage bootloader

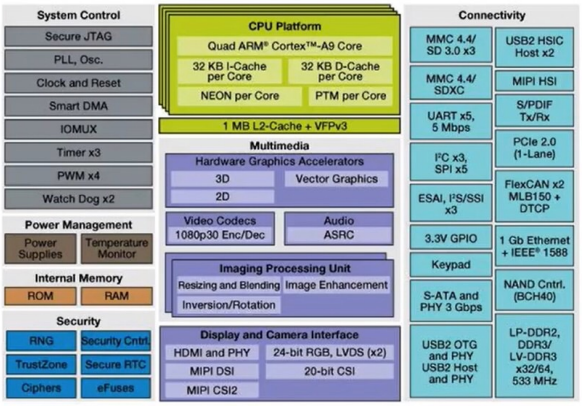
An Example: NXP i.MX6 System-on-Chip

The Internal Memory is composed of:

- 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
- 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

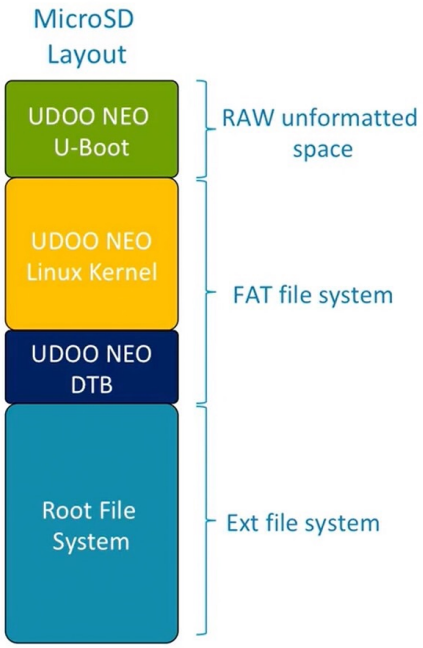
UDOO NEO Boot Process

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