# **I. Tổng quan về hệ điều hành Linux, Linux Embedded**

## **1. Linux OS**

Linux là một hệ điều hành máy tính mã nguồn mở và tự do tương tự như Unix, được xây dựng trên nền nhân Linux (Linux Kernel). Thực chất, các bản phân phối Linux hiện tại là việc kết hợp của GNU/Linux để phục vụ các nhu cầu của người dùng. Một số bản phân phối phổ biến là: Ubuntu, RedHat, CentOS,...

Linux có các ưu điểm:

* Open source
* Miễn phí
* Tính bảo mật cao
* Tính linh hoạt
* Hoạt động mượt trên các máy tính cấu hình yếu

Với các ưu điểm như trên, các bản phân phối của Linux rất thích hợp cộng với máy tính nhúng hiện nay.

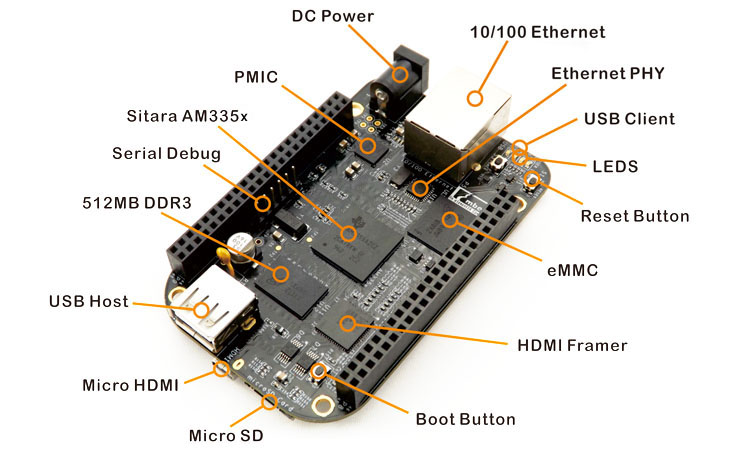
## **2. Linux Embedded**

Linux Embedded có thể được hiểu là các hệ thống embedded mà trên đó sử dụng hệ điều hành Linux. Nó có những ưu điểm mà các hệ thống nhúng sử dụng vi điều khiển không có. Tính năng của Linux Embedded cũng rất đa dạng, thời gian phát triển ứng dụng nhanh, hỗ trợ nhiều thư viên, API có sẵn. Bên cạnh đó là khả năng xử lý muilti thread mạnh,….

Linux Embedded có nhiều ứng dụng trong thiết bị ô tô, thiết bị mạng - viễn thông, thiết bị di động (Android). Về công việc có thể chia Linux Embedded làm 2 phần là tầng kernel và tầng application. Về tổng quan, để làm việc với nó cần những yêu cầu cơ bản:

* Có kinh nghiệm về ngôn ngữ C
* Nắm chắc kiến thức về vi điều khiển
* Có các kiến thức cơ bản về hệ điều hành như: file system, scheduler, virtual memory,…
* Cách thức kernel hoạt động.

Hiện nay, các thiết bị nhúng(máy tính nhúng) phát triển rất mạnh. Từ những mạch giá rẻ cho sinh viên như RaspBerry Pi, BeagleBone Black,.. thì cũng có những mạch cấu hình rất cao dành cho các công việc chuyên sâu như ZC706, ZCU,….



Hình 1. BeagleBone Black



Hình 2. ZC706

# **II. Build Linux OS trên BeagleBone Black**

Về cơ bản, Linux Embedded có 4 thành phần quan trọng nhất là: Toolchains, Bootloader, Kernel, Root FileSystem.

Việc build lại các thành phần này có thể giúp có cái nhìn tổng quan nhất về cách thức hoạt động của Linux OS trên một thiết bị nhúng. Bên cạnh đó, nó còn giúp có thể cấu hình các thành phần quan trong/ cần thiết cho người dùng ngay từ đầu.

## **1. Toolchains**

### **1.1. Tổng quan về Toolchains**

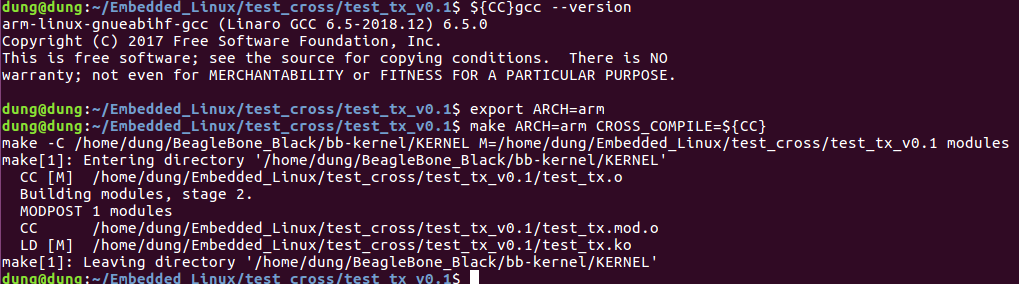
Toolchains là tập hợp các công cụ giúp biên dịch source code thành các mã thực thi có thể chạy trên thiết bị yêu cầu. Toolchain bao gồm các thành phần :

* Binutils (Binary utilities): GNU Assembler, Linker,…
* GCC : GNU C Compiler
* C library: Chứa các file header, file bianry cho phép giao tiếp với hệ điều hành
* GDB (GNU Debugger)

Toolchains được phân ra làm 2 loại là Native và Cross:

* Native compiler: Chỉ biên dịch cho máy nó đang chạy
* Cross compiler: Biên dịch cho 1 thiết bị trên 1 thiết bị khác. Linux Embedded sẽ sử dụng cross compiler để biên dịch code từ 1 máy tính (sử dụng Ubuntu, Kali,..) để biên dịch cho 1 thiết bị nhúng (RaspBerry Pi, Beagbone Black,…)

Nguyên nhân sử dụng Cross Compiler bởi vì cấu hình trên các thiết bị nhúng yếu hơn rất nhiều so với các PC hiện nay. Do vậy, nếu biên dịch trực tiếp trên thiết bị nhúng sẽ rất lâu, nhiều khi sẽ gặp lỗi.



Hình 1. Biên dịch file.ko sử dụng cross compile

### **1.2. Cách build toolchain sử dụng crosstool-NG**

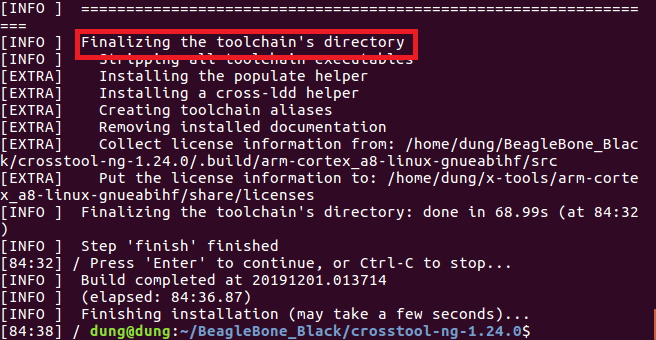
#### **1.2.1. Install crosstool-NG**

Trước khi bắt đầu, máy tính cần cài 1 số phần mềm yêu cầu

*$ sudo apt-get install automake bison chrpath flex g++ git gperf gawk libexpat1-dev libncurses5-dev libsdl1.2-dev libtool python2.7-dev texinfo*

Cài đặt crosstool-NG

*$ git clone https://github.com/crosstool-ng/crosstool-ng.git* *$ cd crosstool-ng* *$ git checkout crosstool-ng-1.22.0* *$ ./bootstrap* *$ ./configure --enable-local* *$ make* *$ make install*



Hình 1. Kết quả build toochain cho BeagleBone Black

## **2. Bootloader**

### **2.1. Tổng quan về Bootloader**

Trong hệ thống Linux Embedded, bootloader có 2 công việc chính là khởi tạo hệ thống từ mức cơ bản và load kernel. Thực chất công việc thứ nhất là 1 phần phụ của công việc thứ 2.

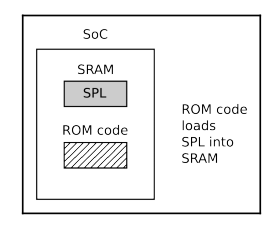
Về tổng quan, khi bật nguồn hoặc reset thiết bị, các dòng đầu tiên của bootloader code được thực thi. Khi đó, DRAM controller chưa được thiết lập dẫn đến việc không thể truy cập vào bộ nhớ chính. Tương tự, các giao diện, các bộ nhớ flash như NAND, MMC cũng không thể sử dụng. Do đó cần sử dụng bootloader để boot đừng thành phần của hệ điều hành lên bộ nhớ chính. Việc cuối cùng của bootloader là boot kernel vào RAM và tạo môi trường cho nó.

Hoạt động của bootloader được chia làm các bước (boot sequence) như sau:

* ***Phase 1 - ROM Code***

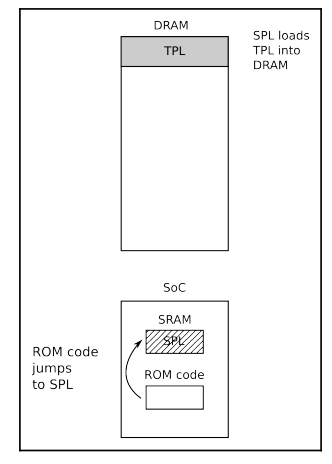
Để đảm bảo tính tin cậy, nhà sản xuất có tích hợp 1 vùng nhớ chứa code (ngay từ khi sản xuất và không thể xóa) để sử dụng khi vừa thiết lập lại hoặc vừa reset thiết bị. Code trong vùng nhớ đó được gọi là ROM code. ROM code có khả năng load 1 số đoạn code nhỏ từ vị trí được chỉ định vào SRAM.

Hầu hết các thiết bị đều được thiết kế 1 vùng nhỏ trên SRAM on-chip (4Kb đến vài trăm Kb). ROM code sẽ load SPL (Secondary program loader) vào vùng nhớ đó.



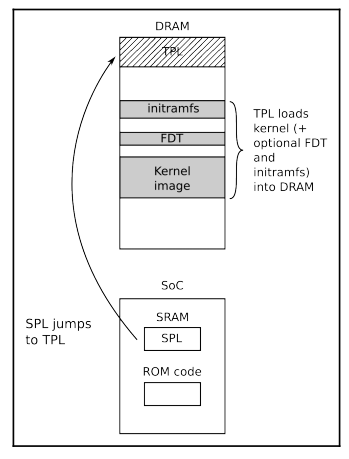
* ***Phase 2 - Secondary program loader***

SPL sẽ thiết lập memory controller và các thành phần thiết yếu để hệ thống chuẩn bị load TPL (Teriary Program Loader). Nếu SPL có tích hợp file system driver thì có thể load thêm u-boot.img từ phân vùng khác.

******

* ***Phase 3 - Teriary Program Loader***

Đến bước cuối này, hệ thống đã boot được bootloader đầy đủ (ví dụ: U-boot, BareBox,…). Hệ thống sẽ có giao diện command line để người dùng có thể load kernel image, can thiệp vào enviroment variable để thiết lập vị trí boot kernel, chọn boot kernel bằng các cách khác nhau như boot từ flash (QSPI, MMC), từ PC thông qua TFTP,…..



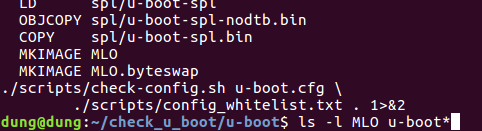
### **2.2. Build U-boot**

U-boot là open source nên có thể tải và build theo các bước sau:

***$ git clone git://git.denx.de/u-boot.git***

***$ cd u-boot***

***$ git checkout v2017.01***

******

Hình 1. Kết quả build U-boot

Sau khi build xong, hệ thống sẽ tạo ra 2 file là u-boot.img và MLO. Trước khi copy vào SD card thì cần phải format và chia lại SD card thành 2 phân vùng (1 phân vùng BOOT với định đạng FAT32, 1 phân vùng ROOTFS với định dạng ext4).

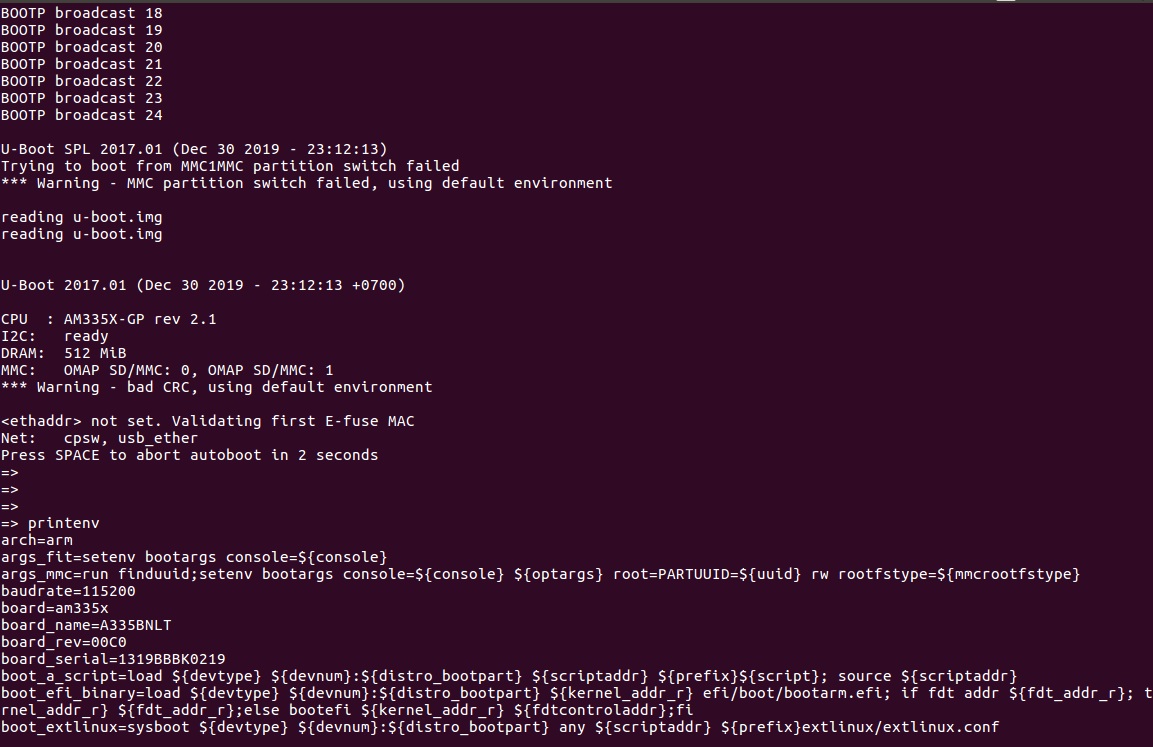
Copy 2 file u-boot.img và MLO vào phân vùng BOOT với lệnh

***$ cp u-boot.img MLO /media/BOOT***

### **2.3. Install U-boot**

U-boot cung cấp cho người dùng 1 khái niệm gọi là enviroment variable. Việc này giúp u-boot có thể cấu hình lại 1 số thành phần bằng command line ngay cả khi đã build xong.

Enviroment variable được viết dưới dạng *name=value.* Trong 1 số trường hợp muốn chọn cách boot kernel (boot từ QSPI, SD card,…) hoặc cấu hình truyền nhận TFTP thì việc viết lại enviroment variable là rất cần thiết.

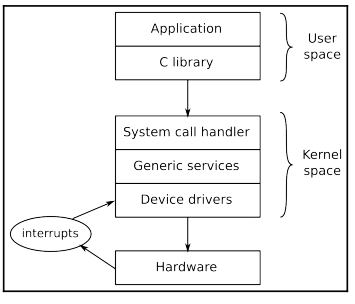


Hình 1. Kết quả install u-boot cho beaglebone black

## **3. Kernel Image**

### **3.1. Tổng quan về Kernel**

Kernel là 1 thành phần quan trọng của Linux. Kernel thực hiện 3 việc chính: quản lý tài nguyên, cung cấp giao diện với phần cứng và cung cấp API cho user space.



Hình 1. Kernel trong hệ thống

Các ứng dụng chạy trên user space với mức độ ưu tiên CPU thấp. Giao diện chính giữa kernel space và user space là C library (sử dụng để thực hiện các yêu cầu từ user space tới kernel space thông qua system call). System call cho phép user chuyển từ chế độ user ưu tiên thấp sang chế độ user ưu tiên cao, cho phép truy cập tất cả địa chỉ ô nhớ và thanh ghi của CPU.

**3.2. Build Kernel**

Tải kernel về máy:

***$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git***

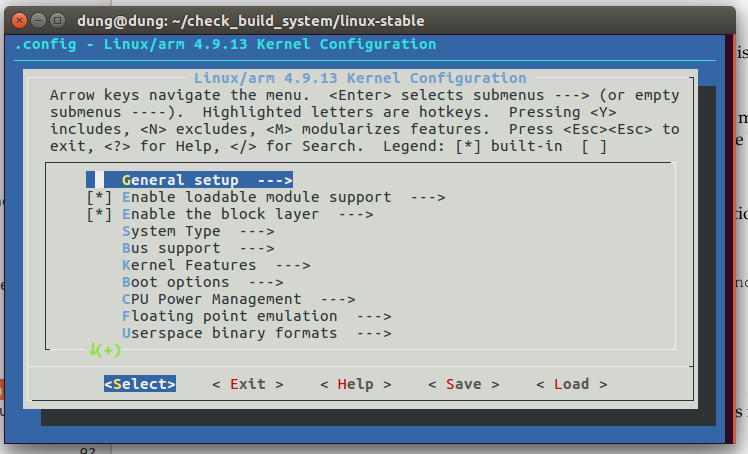
***$ git checkout linux-4.9.y***

*$ cd linux-stable**$ make ARCH=arm CROSS\_COMPILE=arm-cortex\_a8-linux-gnueabihf- mrproper**$ make ARCH=arm multi\_v7\_defconfig**$ make -j4 ARCH=arm CROSS\_COMPILE=arm-cortex\_a8-linux-gnueabihf- zImage**$ make -j4 ARCH=arm CROSS\_COMPILE=arm-cortex\_a8-linux-gnueabihf- modules**$ make ARCH=arm CROSS\_COMPILE=arm-cortex\_a8-linux-gnueabihf- dtbs*

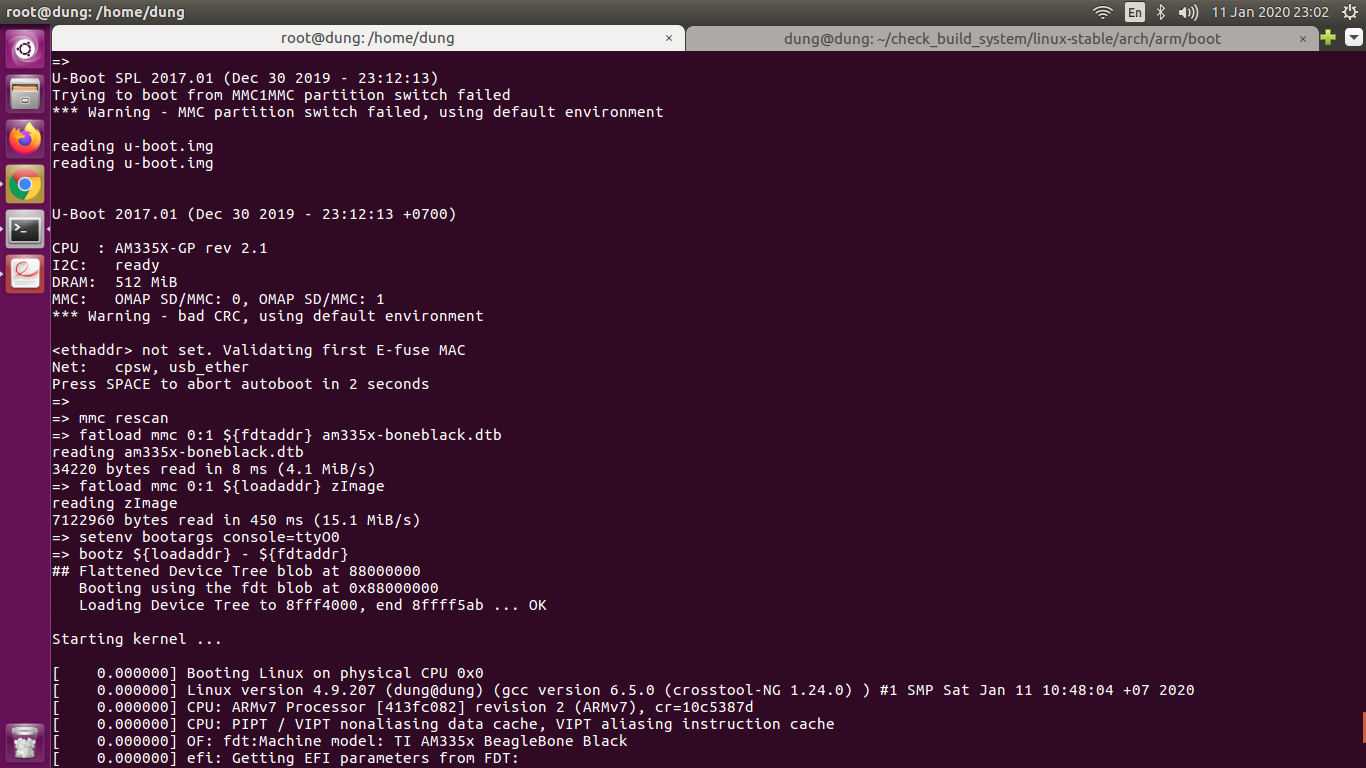
Trong file source code kernel, các file được chứa trong các folder sau:

* **arch:** Chứa các file chỉ định cho từng loại kiến trúc (architecture-specific files)
* **Documentation :** Chứa các tài liệu về kernel. Nếu muốn tìm hiểu thông tin nào về kernel thì vào tìm trong phần này.
* **drivers:** Chứa rất nhiều device driver.
* **fs:** Chứa filesystem code.
* **include:** Chứa kernel header file.
* **init:** Chứa kernel start-up code.
* **kernel:** Chứa các hàm chức năng, bao gồm scheduler, locking, timers, quản lý năng lượng, debug/trace code.
* **mm:** Chứa bộ quản lý bộ nhớ
* **net:** Chứa các giao thức mạng
* **script:** Chứa nhiều script tiện dụng, bao gồm cả device tree compiler, DTC.
* **tools:** Chứa nhiều công cụ hỗ trợ tiện dụng cho Linux.

Ngoài việc build 1 kernel mặc định, source kernel còn hỗ trợ các file Kconfig để người dùng có thể tự cấu hình bật/tắt, thêm các chức năng cho phù hợp với nhu cầu công việc.



Hình 1. Cấu hình cho kernel



Hình 2. Kết quả install kernel trên BeagleBone Black

## **4. Root Filesystem**

### **4.1. Tổng quan Rootfs**

Root Filesystem là thành phần cuối cùng cần được boot lên board. Nó có chứa các thư mục gốc cơ bản của 1 hệ điều hành Linux.

Các thành phần để tạo 1 rootfs tối thiểu:

* **init:** Đây là chương trình để bắt đầu mọi thứ, thường chạy dưới dạng các script
* **Shell:** Cung cấp môi trường để viết command line, chạy các script của init
* **Daemon:** Là 1 background program sử dụng để cung cấp dịch vụ cho các thành phần khác. Ví dụ như system log daemon (syslogd) và secure shell daemon (sshd)
* **Shared Libraries:** Cung cấp liên kết cho hầu hết các chương trình
* **Configuration files:** Cấu hình cho init và các daemon, thường nằm trong đường dẫn /etc
* **Device Nodes:** Là các file đặc biệt cho phép truy cập vào các driver
* **/proc và /sys:** Biểu diễn cấu trúc dữ liệu của Linux dưới dạng hệ thống phân cấp thư mục (Filesystem Hierarchy Standard)
* **Kernel modules:** Nếu có cấu hình/ build các kernel module khi build cùng hệ thống thì chúng được lưu trữ trong /lib/modules/[kernel version]

### **4.2. Build Rootfs**

Tải rootfs về máy:

***$ git clone git://git.buildroot.net/buildroot***

Cấu hình roofs cho BeagleBone Black:

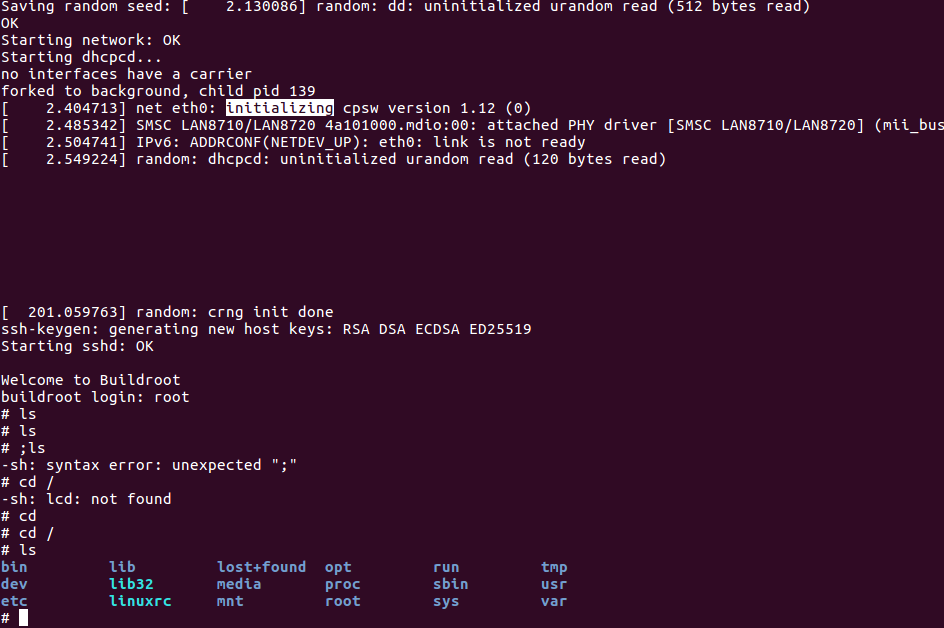
***$ make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- distclean***

***$ make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- beaglebone\_defconfig***

***$ make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf-***

Sau khi build được rootfs.ext4, việc tiếp theo cần quan tâm là boot rootfs lên board như thế nào. Linux hỗ trợ 3 phương thức boot:

* **initramfs:** Còn được gọi là ramdisk, filesystem img sẽ được load vào RAM. Cần lưu ý RAM là bộ nhớ sẽ bị mất hết dữ liệu khi ngắt nguồn khỏi board. Do đó nếu dùng initramfs thì mọi dữ liệu sau khi boot lên sẽ bị mất nếu ngắt nguồn.
* **Disk image:** Đây là 1 bản sao định dạng của rootfs và được load lên thiết bị lưu trữ của board. Disk image có thể là định dạng ext4 nếu muốn copy vào SD card, có thể là định dạng jffs2 nếu muốn copy vào bộ nhớ flash.
* **Network Filesystem:** Rootfs có thể được boot từ máy tính tới board thông qua TFTP, NFS. Cách này thường chỉ sử dụng trong giai đoạn phát triển.



Hình 1. Install Root File System cho BeagbleBone Black

# **III. Linux driver**

## **1. Tổng quan về Linux kernel**

Về khái niệm, hệ điều hành có thể hiểu theo 2 nghĩa :

* Với nghĩa hẹp, hệ điều hành là một phần mềm quản lý tài nguyên phần cứng. Phần mềm này được gọi là Kernel. Với Linux thì còn gọi là ***Linux kernel.***
* Với nghĩa rộng, hệ điều hành là một gói bao gồm phần mềm quản lý tài nguyên phần cứng, giao diện dòng lệnh (Command line Interface - CLI), giao diện đồ họa cho người dùng (Graphic User Interface - GUI),…



Hình 1. Sơ đồ khối chức năng của hệ điều hành

Dựa trên UNIX kernel, Linus Tovarlds đã tạo ra Linux kernel. Dựa vào các chức năng, Linux kernel được chia làm 6 phần chính:

***Process manager*** -có nhiệm vụ quản lý các tiến trình, bao gồm :

* Tạo / hủy các tiến trình
* Lập lịch cho các tiến trình
* Hỗ trợ các tiến trình giao tiếp với nhau
* Đồng bộ hoạt động của các tiến trình tránh tình trạng tranh chấp tài nguyên

***Memory Manager*** - có nhiệm vụ quản lý bộ nhớ, bao gồm :

* Cấp phát bộ nhớ trước khi chạy chương trình, thu hồi bộ nhớ khi kết thúc chương trình
* Đảm bảo các chương trình được đưa vào bộ nhớ
* Bảo vệ vùng nhớ của mỗi tiến trình

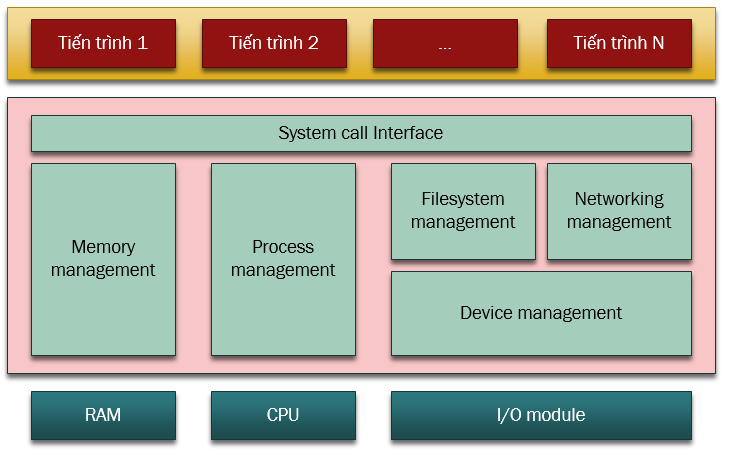
***Device Management*** - có nhiệm vụ quản lý thiết bị, bao gồm:

* Điều khiển hoạt động của các thiết bị
* Giám sát trạng thái của các thiết bị
* Trao đổi dữ liệu với các thiết bị
* Lập lịch sử dụng các thiết bị

***File System Management*** - có nhiệm vụ quản lý dữ liệu trên các thiết bị lưu trữ (ổ cứng, thẻ nhớ,…). bao gồm các công việc: thêm, tìm kiếm, sửa, xóa dữ liệu

***Networking Management*** - có nhiệm vụ quản lý các gói tin theo mô hình TCP/IP

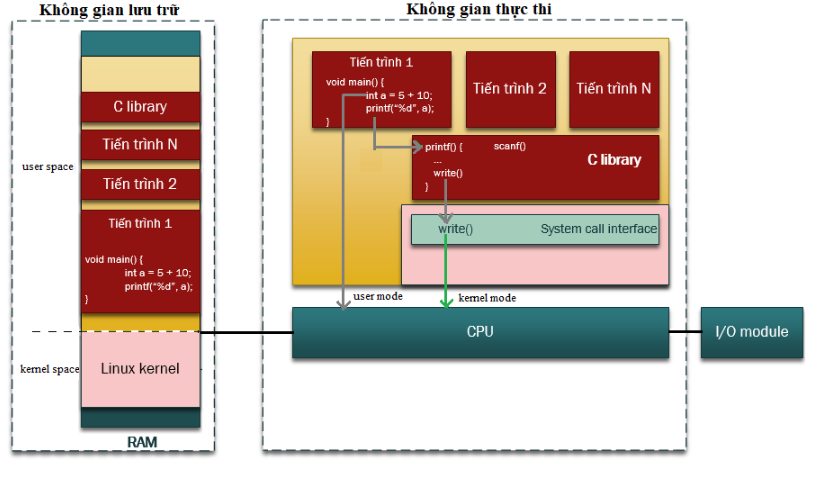
***System Call Interface*** - có nhiệm vụ cung cấp các dịch vụ sử dụng phần cứng cho các tiến trình



Hình 2. Kiến trúc của Linux kernel

Bộ nhớ RAM chứa các lệnh nhị phân của Linux kernel và các tiến trình. RAM được chia làm 2 phần:

* ***Kernel space***: Vùng không gian chứa các lệnh và dữ liệu của kernel
* ***User space***: Vùng không gian chứa các lệnh và dữ liệu của tiến trình



Hình 3. Kiến trúc của Linux kernel (góc độ của CPU)

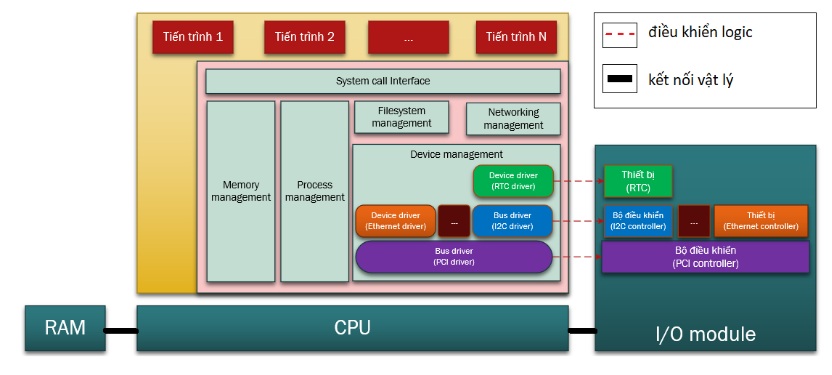
## **2. Linux driver**

Driver là một phần mềm, gồm các lệnh, hướng dẫn CPU tương tác với các thiết bị (màn hình, chuột, ổ cứng,…). Các thiết bị này không được nối trực tiếp tới CPU vì :

* Số lượng chân CPU nhỏ hơn số lượng thiết bị
* Tốc độ làm việc của CPU lớn hơn nhiều so với các thiết bị

Do đó, các thiết bị kết nối với CPU thông qua device controller. Linux driver được chia làm 2 loại là bus driver và device driver. CPU sẽ làm việc với device controller thông qua bus driver. Device driver hướng dẫn CPU làm việc với các thiết bị.

Ví dụ để đọc được giá trị từ RTC DS1307 sẽ cần 2 driver. Bus driver là I2C driver sẽ hướng dẫn cho CPU xử lý các frame truyền theo chuẩn I2C. Device driver là DS1307 driver sẽ hướng dẫn cho CPU làm như thế nào để đọc/ghi các thanh ghi ngày, giờ, tháng, năm,….



Hình 1. Quan hệ giữa driver, controller và device

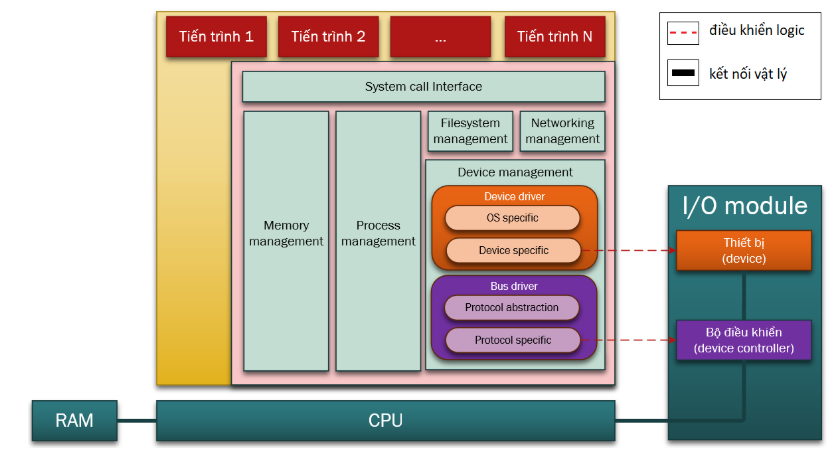
Về chi tiết, bus driver và device driver được chia làm các thành phần cụ thể để dễ làm việc

***Bus driver***

* Protocol abstraction: Cung cấp các dịch vụ cho device driver sử dụng. Ví dụ: đọc ghi thanh ghi của thiết bị
* Protocol specific: Cung cấp các lệnh hướng dẫn CPU làm việc với controller, giúp đọc/ghi dữ liệu trên bus.

***Device driver***

* OS Specific: Cung cấp các dịch vụ giúp CPU đọc/ghi dữ liệu của thiết bị. Ví dụ:cung cấp các hàm entry point như read, write, ioctl,…
* Device Specific: Hướng dẫn CPU điều khiển thiết bị, giám sát thiết bị, trao đổi dữ liệu với thiết bị. Phần này được xây dựng dựa trên datasheet.



Hình 2. Các thành phần của driver

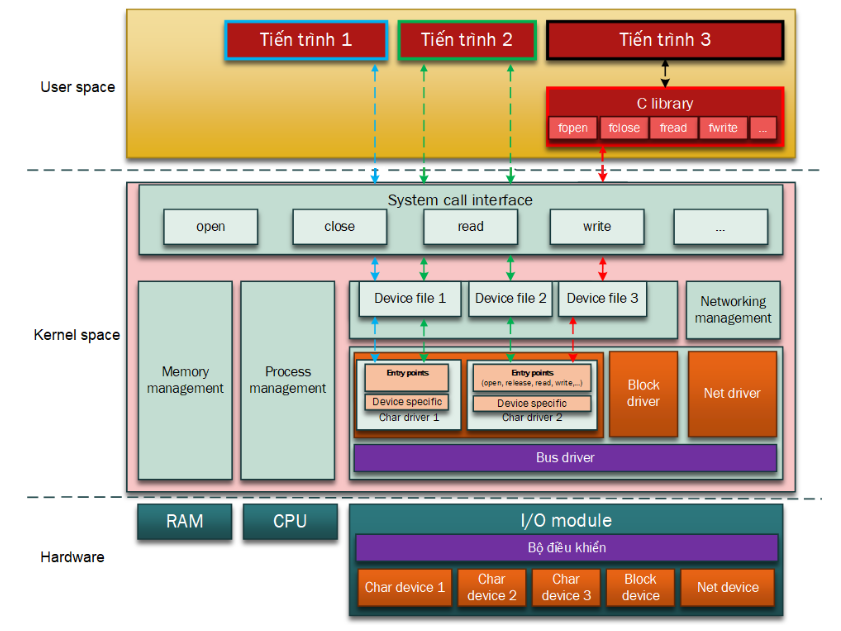
Dựa vào lượng dữ liệu mà mỗi thiết bị trao đổi với CPU, có thể chia thiết bị làm 3 loại:

* **Character device:** Lượng dữ liệu trao đổi min giữa CPU và thiết bị là 1 byte. Ví dụ: chuột, loa,…
* **Block device:** Lượng dữ liệu trao đổi min giữa CPU và thiết bị là 1 khối. Ví dụ: các thiết bị lưu trữ
* **Network device:** Lượng dữ liệu trao đổi min giữa CPU và thiết bị là 1 gói tin (nhiều byte). Ví dụ: các thiết bị mạng (NIC card, Wifi card,…)

### **2.1. Character driver**

Character driver được sử dụng để hướng dẫn CPU làm việc với character device.

Các tiến trình ở user space không giao tiếp trực tiếp với character driver mà thông qua device file (device node). Bằng cách này, Linux kernel đánh lừa các tiến trình là các character device cũng chỉ là 1 file, việc đọc/ghi cũng tương tự với file thông thường.



Hình 1. Kết nối giữa các tiến trình và character device

Cấu trúc của 1 character driver bao gồm :

**OS Specific**

* Hàm khởi tạo:
* Yêu cầu kernel cấp phát device number
* Yêu cầu kernel tạo device file
* Yêu cầu kernel cấp phát bộ nhớ cho các cấu trúc dữ liệu của driver và khởi tạo chúng
* Yêu cầu khởi tạo thiết bị vật lý
* Đăng ký các hàm entry point với kernel
* Đăng ký hàm xử lý ngắt
* Hàm kết thúc : Ngược lại hàm khởi tạo

**Device Specific**

* Hàm khởi tạo/giải phóng thiết bị
* Đọc/ghi các thanh ghi của thiết bị
* Hàm xử lý ngắt

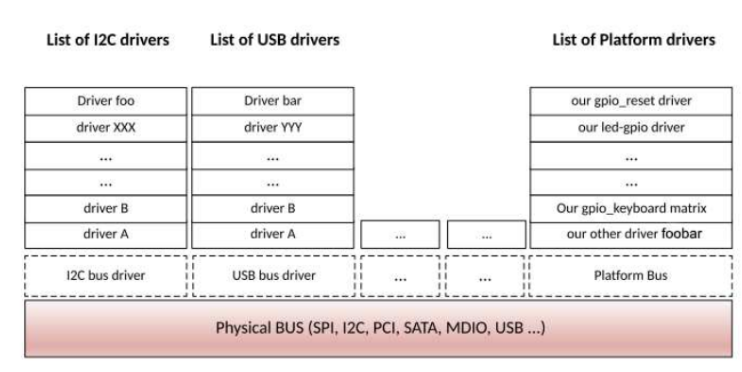
### **2.2. Platform driver**

Platform device được hiểu là các thiết bị khi cắm vào sẽ không chạy luôn (khác với USB, chuột, bàn phím,…). Các thiết bị này cần được khai báo với kernel để có thể sử dụng. Ví dụ các thiết bị sử dụng I2C, SPI,…là platform device vì không thể thấy nếu không khai báo.

Pseudo platform bus (platform bus) là 1 loại bus ảo của kernel, sử dụng cho những device mà không nằm trên 1 bus vật lý nào của kernel. Ví dụ led không nằm trên bus I2C, SPI,…

Platform driver sử dụng để điều khiển platform device. Nhưng platform driver chỉ điều khiển các thiết bị nằm trên platform bus. Ví dụ không thể điều khiển I2C device mặc dù nó là platform device, bởi nó nằm trên bus vật lý I2C có sẵn.

Để có thể match platform driver và platform device ta sử dụng device tree. Platform driver sẽ kiểm tra trường ***compatible*** (trong ***struct platform\_device\_id)*** xem có trùng với trường compatible (trong device tree). Nếu trùng thì có thể match paltform driver với device tree.



Hình 1. Các thiết bị được đăng ký trên bus

## **3. Ứng dụng - UART Software Controller (UART Bit Banging)**

### **3.1.Giới thiệu**

Kiểu truyền dữ liệu nối tiếp bằng phần mềm thay thế cho các phần cứng chuyên dụng. Phần mềm sẽ trực tiếp thiết lập, lấy mẫu trạng thái các chân trên vi điều khiển, chịu trách nhiệm xử lý các tham số của tín hiệu như: thời gian, lấy mẫu, đồng bộ hóa tín hiệu,….Việc thực hiện như trên còn gọi là Bit Banging.

Có thể sử dụng cho một số trường hợp cụ thể. Ví dụ cổng UART bị hỏng, muốn tăng số lượng cổng UART nhưng lại không muốn/không thể sử dụng thêm phần cứng hỗ trợ truyền nhận UART, vi điều khiển không hỗ trợ UART thì có thể sử dụng software UART (UART Bit Banging).

Ưu điểm của phương pháp này là chi phí thấp(hầu như không có), không cần thay đổi phần cứng,…

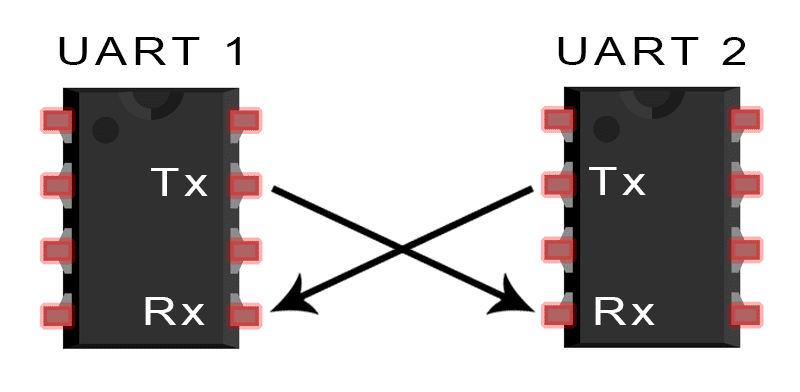
Nhược điểm ở đây là vi điều khiển sẽ tốn thêm tài nguyên để thực hiện các tác vụ của Bit Banging. Do đó có thể làm các tác vụ khác bị chậm đi. Bên cạnh đó, nếu truyền dẫn dữ liệu yêu cầu tốc độ quá nhanh thì có thể các chân GPIO không thể đáp ứng kịp.

### **3.2. UART Bit Banging**

#### **3.2.1. UART Protocol**

****

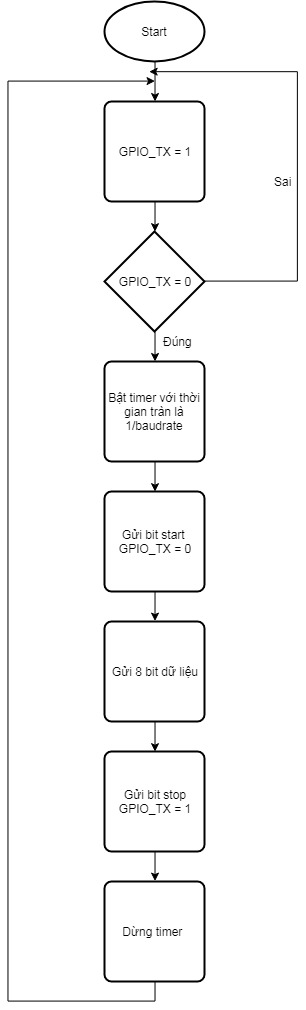
**Hình 1. UART Protocol**



**Hình 2. Kết nối phần cứng**

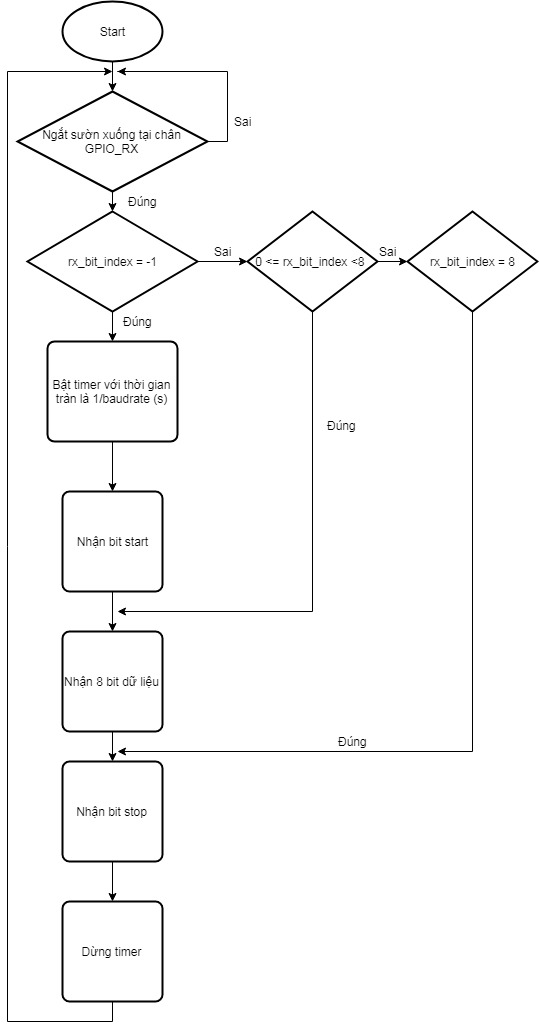
#### **3.2.2. Truyền dữ liệu**

Việc truyền nhận dữ liệu sẽ được 2 thiết bị sử dụng cùng 1 baudrate (tốc độ bit/s). Giả sử, nếu sử dụng baudrate = 9600 thì tương đương với việc thời gian gửi 1 bit mất (s). Nếu sử dụng software UART thì có thể sử dụng timer để định thời gian truyền/ nhận mỗi bit.



#### **3.2.3. Nhận dữ liệu**

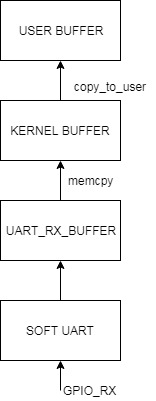
Việc nhận dữ liệu cần được xác nhận bằng cách sử dụng 1 ngắt sườn xuống để kiểm tra khi nào có bit start. Sau khi xác nhận sự xuất hiện của bit start thì việc nhận dữ liệu tương tự việc truyền dữ liệu.



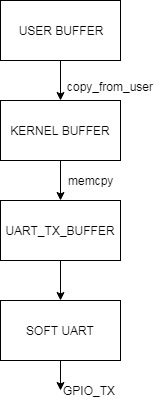
### **3.3. Truyền nhận dữ liệu sử dụng Character Driver**

Có thể tạo device file và sử dụng system call để thực hiện việc truyền nhận UART dễ dàng hơn. Khi đó dữ liệu gửi/nhận để được đẩy vào trong 1 buffer, soft UART sẽ lấy dữ liệu trong buffer đó để xử lý.

Lưu dữ liệu vào user buffer : Dữ liệu sẽ được lưu vào trong 1 buffer theo các bước của việc nhận dữ liệu. Thông qua việc copy dữ liệu vào kernel buffer, dữ liệu sẽ được lưu lên user buffer



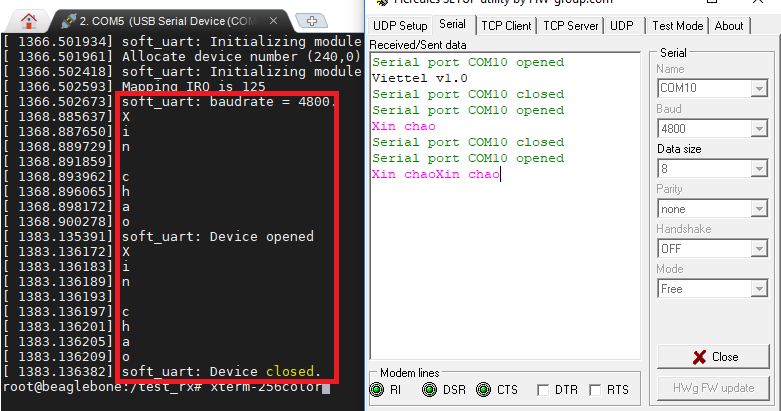
Truyền dữ liệu từ user buffer vào uart\_tx\_buffer: Dữ liệu từ user buffer sẽ được copy vào kernel buffer. Thông qua kernel buffer, dữ liệu sẽ được gửi xuống uart\_tx\_buffer. Việc truyền dữ liệu sẽ theo các bước truyền dữ liệu đã nêu trên.



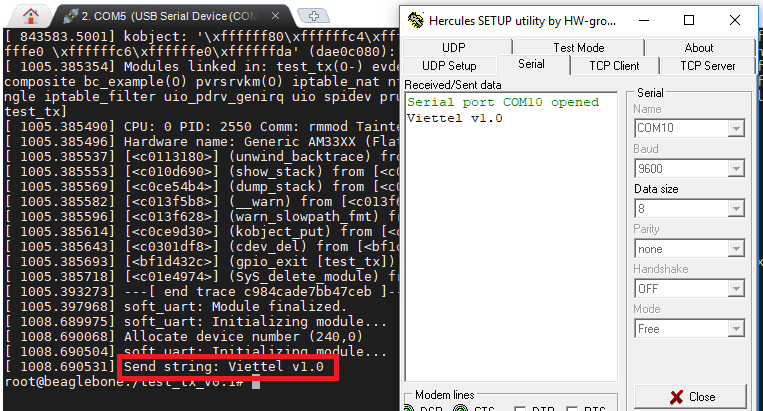
### **3.4. Kết quả sử dụng Soft UART**

Sau khi thử nghiệm với các tốc độ khác nhau như 4800, 9600, 14400, 19200 thì nhận thấy:

* RX nhận ổn định nhất ở tốc độ 4800. Khi tăng lên tốc độ 9600 thì sẽ mất 1 bit đầu tiên. Nếu bỏ qua bit đầu tiên thì vẫn có thể sử dụng nhận với tốc độ 9600.
* TX nhận ổn định nhất ở tốc độ 9600. Khi giảm hoặc tăng các giá trị baudrate khác thì có hiện tượng bị mất bit khi truyền đi với sai số tầm 5%.



**Hình 3. Nhận chuỗi với baudrate 4800**



**Hình 4. Truyền chuỗi với baudrate 9600**

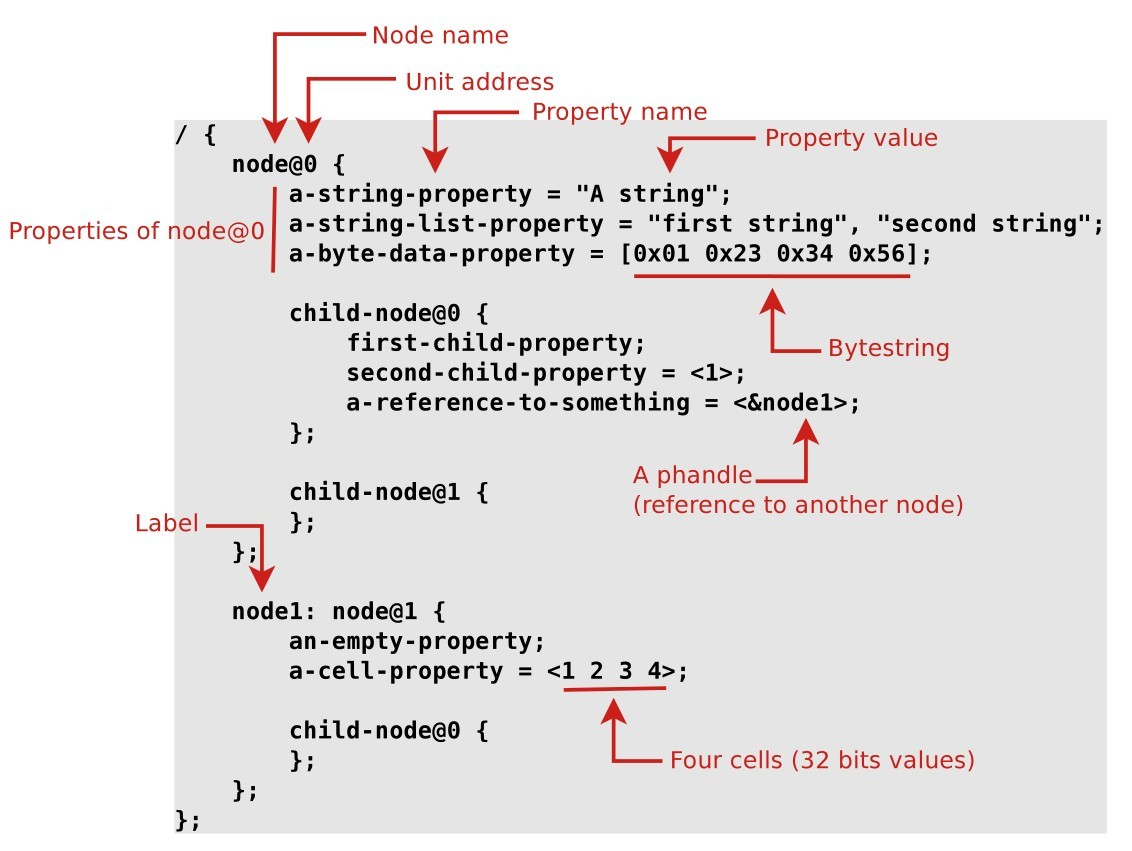
# **IV. Device tree**

Device tree là một file mô tả phần cứng, kiểu định dạng giống JSON. Cấu trúc được mô tả dưới dạng cây với các node đại diện cho device. Mỗi node có các thuộc tính, các thuộc tính có thể được gán giá trị hoặc để trống

Device Tree cũng là một chuẩn được sử dụng với mục đích định nghĩa giao tiếp(I2C, SPI,…) cho các hệ firmware trên máy tính.

Các phần cơ bản của Device Tree gồm:

* Quy ước đặt tên, alias, gán nhãn
* Mô tả dữ liệu và các API
* Quản lý địa chỉ và truy cập tài nguyên của device



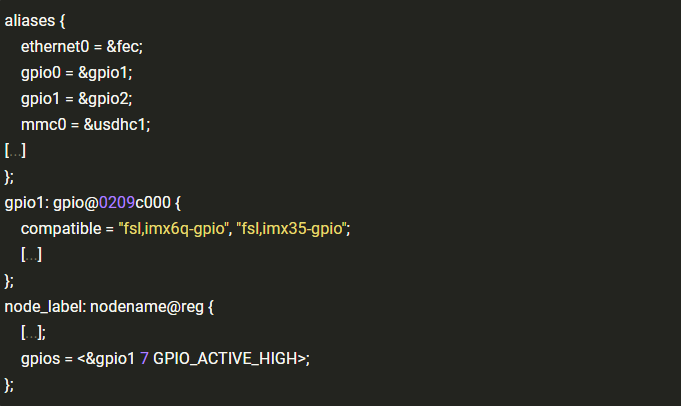
**Hình 1.Mô tả 1 node**

## **1.Quy ước đặt tên**

Node được đặt theo dạng name[@address] với name tối đa 31 ký tự, [@address] là địa chỉ để truy cập vào node. [@address] có thể có hoặc không có.



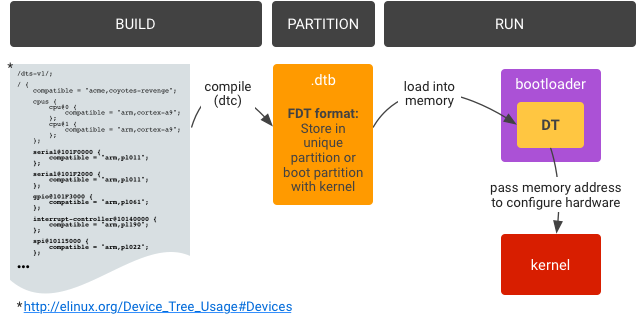
## **2.Alias, label và phandle**

**

Phân tích các thành phần trong hình trên:

* **Lable:** định danh node bằng tên duy nhất (1 lable không thể dùng cho nhiều node). Lable sẽ được DTC chuyển thành giá trị 32 bit duy nhất.Ví dụ: gpio1, node\_lable là các lable.
* **Phandle(pointer handle):** là giá trị 32 bit sử dụng để tham chiếu 1 node với 1 node khác.&gpio là 1 phandle trỏ tới node gpio1
* **Alias:** Giúp kernel có thể rà soát hết cây để tìm node. Alias không được DT dùng mà được kernel dùng.

## **3.Device Tree Compiler (DTC)**

Device Tree Compiler là công cụ biên dịch file .dts sang file .dtb. Device tree được sử dụng dưới các dạng chính:

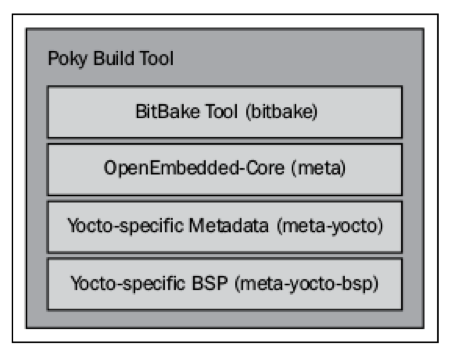
* Dạng source code (Device Tree Source – DTS) có định dạng .dts
* Dạng source code ở mức SoC có định dạng .dtsi, .dtsi sẽ được include vào .dts
* Dạng đã được biên dịch (Device Tree Blod –DTB) có định dạng .dtb

# **Phụ lục. Yocto Project**

## **1. Giới thiệu**

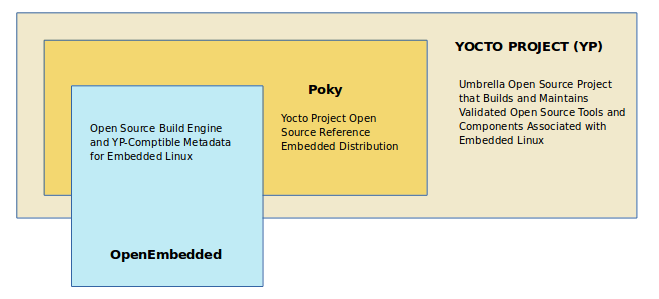
Yocto Project là 1 dự án mã nguồn mở cung cấp các công cụ giúp xây dựng các thành phần của Embedded Linux. Yocto Project bao gồm nhiều thành phần nhiều dự án mã nguồn mở như Poky, BitBake và OpenEmbedded-Core

Yocto Project sử dụng Poky làm reference distribution. Poky bao gồm OpenEmbedded Build System (BitBake + OpenEmbedded-Core) và 1 bộ các metadata mặc định có sẵn (các recipes, file cấu hình,…). Ngoài việc có thể build bootloader, kernel, rootfs thì Yocto project có thể build cả SDK (Software Development Kit)



Hình 1. Poky build system

Việc sử dụng Buildroot để build từng thành phần riêng lẻ có thể gây 1 số lỗi khi kết hợp các phần với nhau (bootloader, kernel, rootfs). Bởi vậy, sử dụng Yocto Project để build tất cả các thành phần là 1 giải pháp hiệu quả



Hình 2. Các thành phần của Yocto Project

Một số lưu ý:

* Yocto không phải 1 bản phân phối của Linux
* Yocto không phải build system
* Yocto không phải framework
* Yocto không phải tool

## **2. So sánh Yocto Project và Buildroot**

|  |  |  |
| --- | --- | --- |
|  | **Yocto Project** | **Buildroot** |
| Tư tưởng | * Hỗ trợ nhiều kiến trúc * Chỉ cung cấp Core recipes, sử dụng layer để mở rộng và thêm gói hỗ trợ * Việc custom xảy ra ở các layer khác nhau | * Tập trung vào sự đơn giản * Dễ sử dụng, dễ hiểu, dễ mở rộng * Sử dụng các công nghệ hiện tại: makefile, kconfig * Kích thước càng nhỏ càng tốt |
| Output | * Tạo bản phân phối, ouput là 1 loạt các package * Có thể tạo ra roofs img * Có thể tạo 1 disk image * Có thể tạo ra SDK kèm theo image | * Output chính là roofs img * Không có gói binary, không có hệ thống quản lý gói * Cần phải update toàn bộ hệ thống |
| Layer | * Cơ chế layer cho phép sửa/ thêm package mới * Bên thứ 3 có thể cung cấp layer với BSP | * Không có khái niệm này * Tất cả package đều được lấy từ repository |
| Mức độ phức tạp | * Cần tìm hiểu kỹ * Core BitBake là 1 dự án tách riêng * Có tài liệu hướng dẫn chi tiết | * Thiết kế đơn giản, dễ hiểu * Core logic được viết hoàn toàn bằng make * Tài liệu chi tiết, resource dễ dàng tìm kiếm và sử dụng |