# Android Kernel Hacks

## 안드로이드를 위한 리눅스 커널 핵스

2013.7.16 ~ 2013.9.27

이 충 한(chunghan.yi@gmail.com, slowboot)

# 목차

## CHAPTER 4

# ARM 보드 초기화 과정 분석

본 장에서는 ARM SoC 및 보드 초기화 과정에 필요한 Device Tree, Clock, Pin Control, Platform Device Driver 등의 동작 원리를 살펴 본 후, 이를 토대로 실제 보드 초기화 과정이 어떠한 형태로 이루어지는지를 분석해 보고자 한다.

- ARM SoC와 보드(Board) 소개

- Device Tree
- Timer, IRQ controller, Serial port driver, Earlyprintk
- Common Clock Framework
- Pin Control Subsystem
- GPIO

- Platform Device 기반의 구(old) ARM 보드 초기화
- Device Tree 기반의 신(new) ARM 보드 초기화

# 1. ARM SoC와 보드 소개

ARM 아키텍쳐는 영국의 ARM 홀딩스(Holdings)가 설계한 것으로, 이 회사는 명령어 셋(instruction sets)을 정의(spec)하고, 명령어 셋, 메모리 관리 장치(memory management unit), 캐쉬(cache) 등을 구현한 ARM Core(예: ARM926EJ-S, Cortex-A8, Cortex-A9, Cortex-M3, etc.)를 전 세계의 실리콘 벤더(silicon vendor)에 판매한다. 각 실리콘 벤더(Broadcom, Ti, Samsung, Marvell, Qualcomm, etc)는 ARM Core에 다양한 주변 장치(peripherals)를 포함시켜 그들만의 독자적인 SoC(System-on-Chip)를 만들게 되고, 각 시스템 제작 업체(system maker)에서는 이를 구입한 후, 최종적으로 자신들이 설계한 PCB(Printed Circuit Board)에 여러 컴포넌트를 추가하여 원하는 보드를 생산해 내게 된다.

그림 4-1과 4-2는 지금까지 설명한 바와 같이 ARM 아키텍쳐(architecture)로부터 하나의 보드가 생산되기까지의 과정을 그림으로 표현한 것이며, 그림 4-3은 ARM Core와 SoC 및 보드 간의 상관 관계를 역시 그림으로 표현한 것이다.

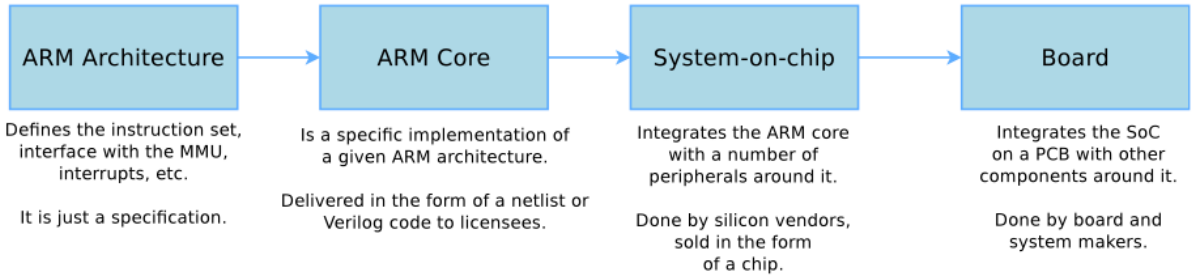


**그림 4-1 ARM 아키텍쳐에서 보드까지의 생성 과정(1) [출처 – 참고 문헌 1-6]**

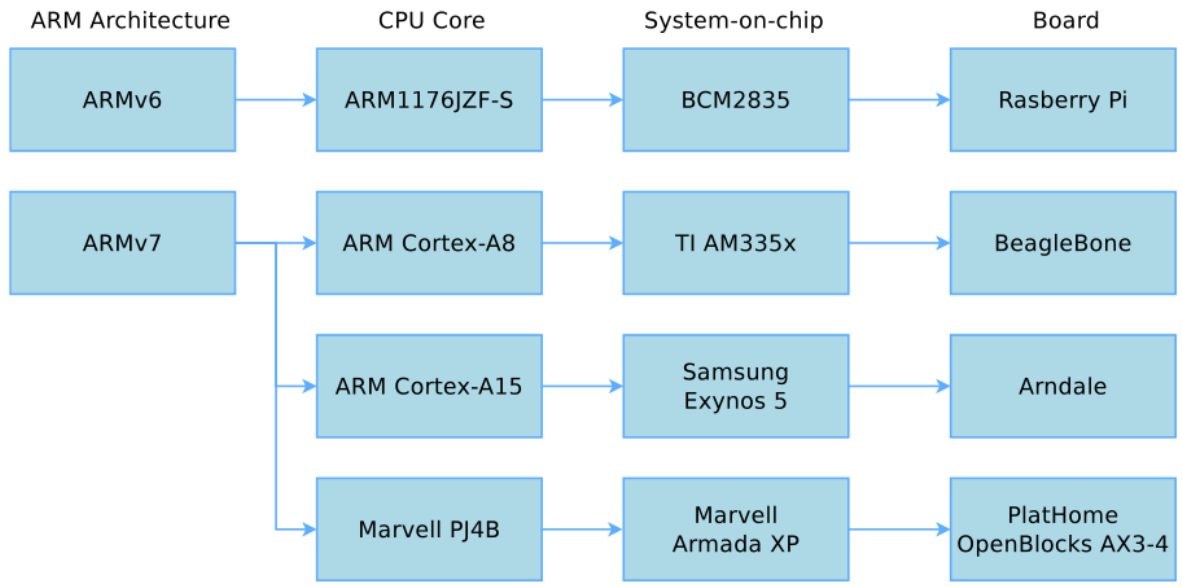


**그림 4-2 ARM 아키텍쳐에서 보드까지의 생성 과정(2) [출처 – 참고 문헌 1-6]**

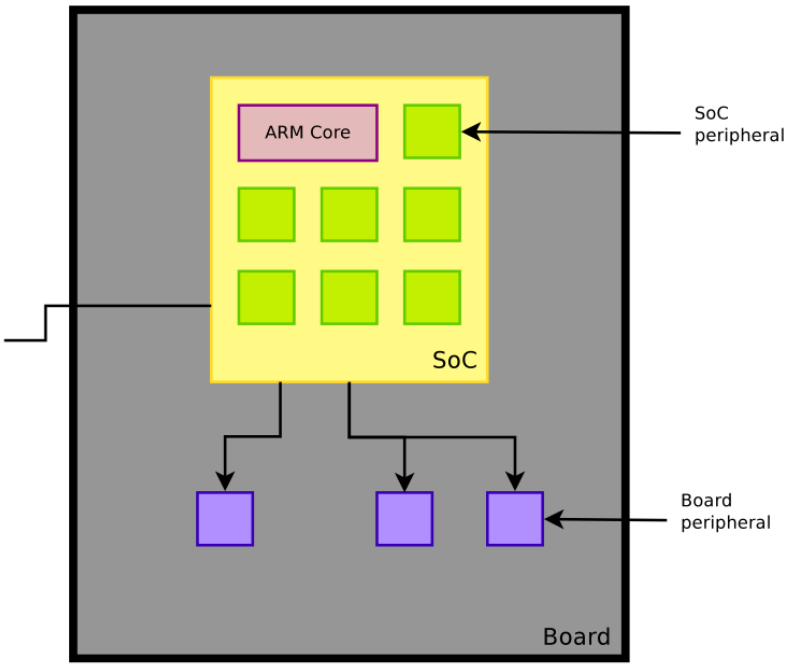


**그림 4-3 ARM Core, SoC 및 보드(board)의 상관 관계 [출처 – 참고 문헌 1-6]**

스마트 폰 덕분에 최근에 많이 접하는 Cortex-A8, Cortex-A9, Cortex-A15 등은 ARM Core에 해당하며, 이로부터 다양한 SoC(예: TI OMAP 시리즈, Samsung Exynos 시리즈, Qualcomm Snapdragon 시리즈 등)들이 파생되게 되는데, 이를 그림으로 표현하면 다음과 같다.

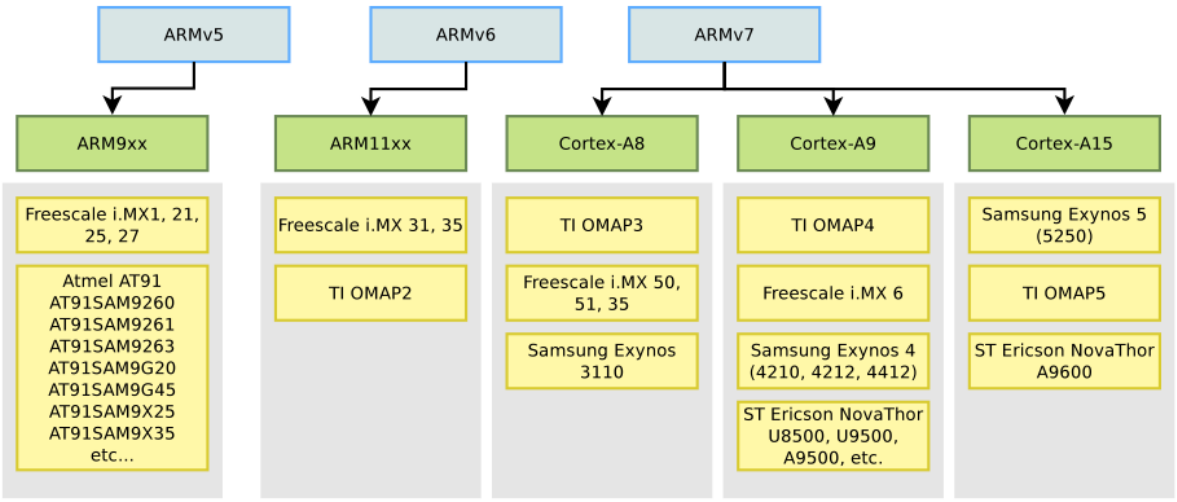


**그림 4-4 ARM 아키텍쳐의 발전 과정 [출처 – 참고 문헌 1-6]**

앞서서 이미 언급했다시피, SoC는 ARM Core에 다양한 주변 장치를 추가한 것인데, 그림 4-5는 SoC에 다양한 주변 장치를 추가시켜 스마트 폰 용 SoC로 변화시킨 예를 보여준다.

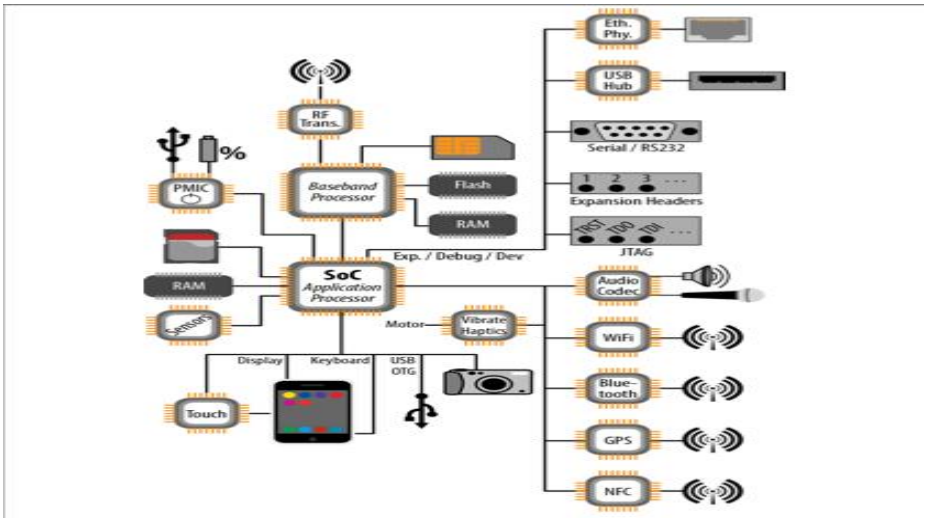


**그림 4-5 Smart Phone SoC 기본 구조 [ 출처 – 참고 문헌 ]**

# 2. 디바이스 트리(Device Tree)

arch/arm/mach-* 아래의 ARM 커널 코드를 살펴 보면서 느끼게 되는 점은, 매우 다양한 플랫폼이 존재한다는 사실과 함께, 각각의 플랫폼 별로 코드가 매우 복잡하고 방대하며, 플랫폼 마다 유사한 코드가 많이 존재한다는 점일 것이다. 이는 x86, mips, ppc 등 다른 CPU core와는 확연한 차이를 보이는 부분으로, 이와 관련하여 2011년 3월에 리누즈 토발즈(Linus Torvalds)는 아래와 같은 말을 남겼고, 이는 이후 ARM sub-architecture 팀이 결성되는 계기를 만들어 주었다.

*Gaah. Guys, this whole ARM thing is a f*cking pain in the ass.*
*Linus Torvalds - 17 March 2011*

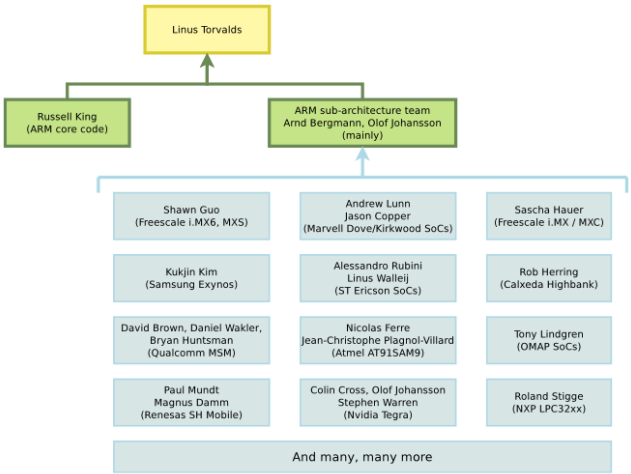


**그림 4-6 ARM sub-architecture team과 ARM code review 과정 [출처 – 참고 문헌 1-6]**

ARM sub-architecture 팀이 결성된 목적은 기존의 복잡하고 방대하며, 플랫폼 별로 중복되어 코드를 잘 다듬어, 하나의 통일된 ARM code로 발전시키기 위함인데, 이러한 내용의 중심에 있는 것이 앞으로 이번 장에서 설명할 Device Tree이다.

이제부터는 새로운 보드(board 혹은 machine)를 위한 디바이스 트리(device tree)를 작성하는 방법을 소개하고자 한다. 디바이스 트리에 관한 자세한 설명을 위해서는 우선적으로 ePAPR spec 문서(https://www.power.org/documentation/epapr-version-1-1/)를 참조해야 하며, 이번 절에서는 이 문서를 기반으로 작성된 http://www.devicetree.org/Device_Tree_Usage 웹 사이트의 내용을 최대한 자세히 소개해 보고자 한다.

## 기본 데이터 형식

디바이스 트리(device tree)는 노드(node)와 속성(property)으로 구성된 트리 구조이다. 속성은 키(key)와 값(value)을 한 쌍으로 하며, 노드는 속성과 자식(child) 노드를 모두 포함할 수 있다. 예를 들어 아래의 내용은 .dts 형식으로 된 간단한 트리를 보여준다.

```
/ {
    node1 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        a-byte-data-property = [0x01 0x23 0x34 0x56];
        child-node1 {
            first-child-property;
            second-child-property = <1>;
            a-string-property = "Hello, world";
        };
        child-node2 {
        };
    };
    node2 {
        an-empty-property;
        a-cell-property = <1 2 3 4>; /* each number (cell) is a uint32 */
        child-node1 {
        };
    };
```

```
};
```

위의 예는 실제 예가 아니므로 별로 도움이 안될지 모르나, 노드와 속성에 관현 구조를 보여주기에는 충분하다. 즉,

- *단일 root 노드: "/"*
- *여러 개의 자식 노드: "node1"과 "node2"*
- *node1 에 대한 여러 자식 노드: "child-node1"과 "child-node2"*
- *트리 내에 퍼져 있는 여러 속성 정보*

속성은 키(key)와 값(value)의 단순 조합인데, 값(value)은 비워 있을 수도 있고, 임의의 바이트 스트림(stream)으로 채워질 수도 있다. 디바이스 소스 파일 안에는 각각의 데이터 형식(data type)을 표현하는 몇 가지 기본 규칙이 있다.

- *텍스트 문자열(null 로 끝남)은 큰 따옴표로 구분함.*
  - *string-property = "a string"*
- *셀은 '<' 및 '>'으로 구분되는 32bit 정수(unsigned integer)임.*
  - *cell-property = <0xbeef 123 0xabcd1234>*
- *바이너리 데이터는 '[' 및 ']'으로 구분됨.*
  - *binary-property = [0x01 0x23 0x45 0x67];*
- *서로 다른 데이터를 함께 묶어 표현할 때에는 콤마(,)를 사용함.*
  - *mixed-property = "a string", [0x01 0x23 0x45 0x67], <0x12345678>;*
- *콤마는 문자열 목록을 표현하는데도 사용함.*
  - *string-list = "red fish", "blue fish";*

## 기본 개념

디바이스 트리가 어떻게 사용되는지를 이해하기 위해, 간단한 보드(혹은 머신)를 소개하고, 디바이스 트리를 써서 하나씩 표현해 보도록 하겠다.

### 샘플 머신(보드)

다음과 같이, "Acme"라는 회사에서 만든 "Coyote's Revenge"라는 가상의 보드(ARM 기반)을 예로 들어 보자.

- *1 개짜리 32bit ARM CPU*

- *memory mapped 시리얼 포트, SPI 버스 컨트롤러, I2C 컨트롤러, 인터럽트 컨트롤러, 외부 버스(external bus) 브릿지(bridge)에 붙어 있는 프로세서 내부 버스*
- *0 번지에서 시작하는 256MB 크기의 SDRAM*
- *0x101F1000 와 0x101F2000 시작 번지를 갖는 2 개의 시리얼 포트*
- *0x101F3000 번지에서 시작하는 GPIO 컨트롤러*
- *0x10170000 번지에서 시작하며, 다음의 장치를 갖고 있는 SPI 컨트롤러*
  - *GPIO 1 번에 붙어 있는 SS 핀을 갖는 MMC 슬롯*
- *다음의 장치를 갖는 외부 버스 브릿지*
  - *0x10100000 번지에서 시작하는 외부 버스에 붙어 있는 SMC SMC91111 이더넷 장치*
  - *다음의 장치를 가지며, 0x10160000 번지에서 시작하는 i2c 컨트롤러*
    - *Maxim DS1338 리얼타임 클록(RTC). 슬레이브 주소(Slave address)가 1101000 (0x58) 임.*
  - *0x30000000 에서 시작하는 64MB NOR 플래시(flash) 메모리*

## 초기 구조(initial structure)

첫 번째로 해야 할 일은 머신에 대한 기본 골격을 정하는 일이다. 이는 유효한 디바이스 트리를 위한 최소한의 작업으로써 이 단계에서 머신의 고유한 이름을 명명하게 된다.

```
/ {
    compatible = "acme,coyotes-revenge";
};
```

**compatible** 은 시스템의 이름을 지정하는데 사용한다. 시스템의 이름은 "제조사,모델명" 형식의 문자열로 이루어져 있다. 장치 명간에 충돌을 피하기 위해서는 정확한 제조사 및 모델명을 입력하는 것이 중요하다. OS(linux)는 머신을 구동하는 방법에 대한 결정을 내리기 위해 이 **compatible** 값을 사용할 것이기 때문에, 이 속성 값에 정확한 값을 적는 것이 매우 중요함을 다시 한번 강조한다. 이론상 **compatible** 속성이 OS(linux)가 머신을 명명하기 위해 필요로 하는 내용의 전부라 할 수 있다. 머신에 대한 세부 사항이 하드 코딩되어 있다면, OS 는 맨 상위 레벨에 있는 **compatible** 속성 값인 "acme,coyotes-revenge"를 찾게 된다.

## CPU

다음으로 해야 할 작업은 각각의 CPU 를 기술하는 것이다. 다음의 예는 2 개의 자식 노드(dual core Cortex A9)를 갖는 "cpus"라는 이름의 노드(node)를 표현하고 있다.

```
/ {
```

```
    compatible = "acme,coyotes-revenge";

    cpus {
        cpu@0 {
            compatible = "arm,cortex-a9";
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
        };
    };
};
```

각 cpu 노드에 있는 **compatible** 속성은 "제조사,모델명" 형식으로 이루어져 있으며, 정확한 cpu 모델을 나타내는 문자열로 이해할 수 있다. 이는 앞서 살펴본 상위 레벨의 **compatible** 속성과 동일한 형식이다. 보다 많은 속성이 cpu 노드에 추가가 되어야 하지만, 우선은 기본 개념에 어울리는 부분을 먼저 집고 넘어갈 것이다.

## 노드명

이름을 정하는 방법을 잠시 소개해 보도록 하겠다. 각각의 노드는 **<name>[@<unit-address>** 형태의 명명 규칙을 따른다. <name>은 단순 아스키(ascii) 스트링이며, 최대 31 자까지 사용할 수 있다. 일반적으로 노드는 그 노드가 무엇을 의미하는지에 잘 부합되는 것으로 이름 지어져야 한다. 예를 들어, 3com 이더넷 어댑터의 경우는 3com509 라는 이름 보다는 ethernet 이라는 이름이 더 적합하다 할 것이다. 주소 값을 갖는 장치를 표현해야 한다면, <unit-address>가 노드 내에서 포함되어야 한다. <unit-address>는 장치에 접근하기 위해 사용되는 1 차 주소이고, 노드 내의 **reg** 속성에 나열되어 있는 정보이다. 이 부분은 **reg** 속성을 언급하는 부분에서 다시 살펴 볼 것이다. 자식 노드들은 유일한 이름을 부여해야 하지만, 주소 값이 다르다면 한 개 이상의 노드가 동일한 이름을 사용하는 것이 가능하다. 즉, serial@101f1000 과 serial@101f2000 처럼 말이다

## 장치들(Devices)

시스템 내의 모든 장치는 디바이스 트리 노드로 표현된다. 다음 단계는 디바이스 트리를 각각의 장치를 나타내는 노드로 표현하는 것이다. 주소 범위(address range)와 irq 가 어떻게 다루어지는지를 논의할 때까지 당분간, 새로운 노드들은 공백 상태로 남아 있게 될 것이다.

```
/ {
    compatible = "acme,coyotes-revenge";
```

```
cpus {
        cpu@0 {
                compatible = "arm,cortex-a9";
        };
        cpu@1 {
                compatible = "arm,cortex-a9";
        };
};

serial@101F0000 {
        compatible = "arm,pl011";
};

serial@101F2000 {
        compatible = "arm,pl011";
};

gpio@101F3000 {
        compatible = "arm,pl061";
};

interrupt-controller@10140000 {
        compatible = "arm,pl190";
};

spi@10115000 {
        compatible = "arm,pl022";
};

external-bus {
        ethernet@0,0 {
                compatible = "smc,smc91c111";
        };

        i2c@1,0 {
                compatible = "acme,a1234-i2c-bus";
                rtc@58 {
```

```
                compatible = "maxim,ds1338";
            };
        };

        flash@2,0 {
            compatible = "samsung,k8f1315ebm", "cfi-flash";
        };
    };
};
```

트리에서 하나의 노드는 시스템 내의 각 장치를 위해 추가되었으며, 계층 구조는 각 장치들이 어떻게 시스템에 연결되어 있는지를 보여주게 된다. 즉, 외부 버스에 붙어 있는 장치들은 외부 버스 노드의 자식들이며, i2c 장치들은 i2c 버스 컨트롤러 노드의 자식들이다. 일반적으로 이러한 계층 구조는 CPU 의 관점에서 전체 시스템을 보여준다. 위의 트리는 현재 시점에서 볼 때 유효하지 못하다. 이는 각 장치들간의 연결 정보가 없기 때문이다. 이 부분은 나중에 추가될 것이다.

이 예제에서 주목해야 할 만한 부분을 정리하면 다음과 같다:

- *모든 장치는 **compatible** 속성을 가지고 있다.*
- *플래시 노드는 **compatible** 속성으로 2 개의 문자열을 갖고 있다. 그 이유를 알고자 한다면 다은 절을 읽어 보기 바란다.*
- *앞서 언급한 것 처럼, 노드 명은 특정 모델을 지칭하기 보다는 장치의 타입을 나타낼 수 있어야 한다. 이와 관련하여 가능한 사용해야 하는 기 정의된 일반 노드 명칭을 알아보기 위해서는 ePAPR 문서의 2.2.2 절을 살펴보기 바란다.*

## compatible 속성 이해하기

장치를 표현하는 트리 내의 각 노드는 **compatible** 속성을 반드시 가져야 한다. **compatible** 속성은 OS 가 어느 디바이스 드라이버를 이 장치에 연결할지를 결정하는 중요한 핵심 역할을 한다.

**compatible** 은 문자열의 목록으로 이루어져 있다. 첫 번째 문자열("<manufacturer>,<model>")은 노드가 표현하고자 하는 정확한 장치를 나타내고, 이어지는 문자열은 그 장치와 연관되어 있는 다른 장치를 표시할 수 있어야 한다. 예를 들어, Freescale MPC8349 SoC 의 경우는 National Semiconductor ns16550 레지스터 인터페이스를 구현하고 있는 시리얼 정치를 포함하고 있다. 따라서 MPC8349 시리얼 장치의 **compatible** 속성은 compatible = "fsl,mpc8349-uart", "ns16550" 과 같이 기술해야 한다. 여기서 첫 번째 문자열인 "fsl,mpc8349-uart"은 정확한 장치를 표현해주고

있으며, 이어서 나오는 문자열인 "ns16550"은 National Semiconductor 16550 UART 와 레지스터 단계에서 상호 호환된다는 것을 보여주고 있다.

***참고 사항**: ns16550 는 제조사 이름이 앞에 붙어 있어야 함에도 불구하고 그렇지 않은데, 이는 예전부터 관행적으로 그렇게 사용해 왔기 때문이며, 다른 경우에는 반드시 제조사로 시작하도록 해야 한다.*

***주의 사항**: compatible 속성에 값을 줄 때 와일드카드(wildcard) 형식을 사용해서는 안된다. 즉, "fsl,mpc83**xx**-uart"과 같은 식으로 표현해서는 안 된다.*

## 주소 지정 방식 이해(How Addressing Works)

주소 지정이 가능한 장치들은 디바이스 트리 내에서 주소 정보를 표현하기 위하여 다음의 속성들을 사용한다.

- *reg*
- *#address-cells*
- *#size-cells*

주소 지정이 가능한 각각의 장치는 reg = <address1 length1 [address2 length2] [address3 length3] ... >의 형태를 갖는 **reg** 속성을 사용할 수 있는데, 이는 장치가 사용하는 주소 범위를 표현하는 역할을 한다. 각 주소(address) 값은 cell 이라고 부르는 1 개 이상의 32bit 정수 값으로 구성된 목록이다. 마찬가지로 길이(length) 값은 cell 들의 목록이거나, 빈 값일 수 있다. 주소와 길이 필드가 가변 크기를 갖는 변수이므로, 부모 노드에 있는 #address-cells 나 #size-cells 속성은 얼마나 많은 cell 들이 각각의 필드에 있는지를 보여준다. 혹은 다른 식으로 표현해서 **reg** 속성 값을 정확히 분석하기 위해서는 부모 노드의 #address-cells 와 #size-cells 값이 필요하다. 동작 과정을 이해하기 위해 지금까지 살펴본 샘플 디바이스 트리에 주소 관련 속성을 추가해 보자.

### CPU 주소 지정(CPU addressing)

주소 지정 방식을 이해할 때, CPU 노드는 가장 간단한 경우에 해당한다. 각각의 CPU 는 하나의 유일한 ID 값을 할당 받으며, CPU ID 와 연관된 size 값은 없다.

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

```
            reg = <0>;
        };
        cpu@1 {
            compatible = "arm,cortex-a9";
            reg = <1>;
        };
    };
```

cpu 노드의 경우에 #address-cells 값은 1 이며, #size-cells 값은 0 이다. 이는 자식 노드의 **reg** 값이 주소 값 만을 나타내는 1 개의 필드(uint32)로만 이루어져 있음을 뜻한다. 위의 예의 경우, 2 개의 cpu 노드가 각각 주소 0 과 1 을 할당 받고 있다. 주목해야 할 사항으로는, **reg** 속성에 있는 값이 노드 명을 구성하고 있다는 사실이다. 관행적으로 노드 내에 **reg** 속성이 있다면, 노드의 명칭은 **reg** 속성의 첫 번째 주소 값으로 이루어진 <unit-address>를 포함(@ 다음에 붙여 주어야 함)시켜야 한다.

## Memory Mapped 장치들

cpu 노드에서 보았던 단일 주소(single address) 값 대신에, memory mapped 장치는 일련의 주소 범위를 값으로 부여 받는다. #size-cells 는 길이 필드(length field)가 각 자식 노드의 reg 속성 내에 얼마큼 있는지를 나타내는 용도로 사용된다. 다음의 예에서, 각각의 주소 값은 1 개의 cell(32 비트), 길이 값 역시 1 개의 cell(이 경우는 32bit 시스템의 경우에 해당함)로 이루어져 있음을 알 수 있다. 참고로 64bit 머신의 경우는 64bit 주소 계산을 위해 #address-cells 값과 #size-cells 값이 각각 2 일 것이다.

```
/ {
    #address-cells = <1>;
    #size-cells = <1>;

    ...

    serial@101f0000 {
        compatible = "arm,pl011";
        reg = <0x101f0000 0x1000 >;
    };

    serial@101f2000 {
        compatible = "arm,pl011";
        reg = <0x101f2000 0x1000 >;
```

```
        };

        gpio@101f3000 {
            compatible = "arm,pl061";
            reg = <0x101f3000 0x1000
                   0x101f4000 0x0010>;
        };

        interrupt-controller@10140000 {
            compatible = "arm,pl190";
            reg = <0x10140000 0x1000 >;
        };

        spi@10115000 {
            compatible = "arm,pl022";
            reg = <0x10115000 0x1000 >;
        };

        ...

};
```

각 장치는 시작 주소(base address)와 할당 받은 영역의 크기를 부여 받는다. 이번 예에서 GPIO 장치는 두 개의 주소 범위를 부여 받았다. 즉, 0x101f3000...0x101f3fff 와 0x101f4000..0x101f400f.

어떤 장치들은 서로 다른 주소 체계(scheme)을 갖는 버스에 붙어 있다. 예를 들어 한 장치가 구별 가능한 칩 선택 선(chip select line)을 갖고 있는 외부 버스(external bus)에 붙어 있다고 하자. 각각의 부모 노드(parent node)는 자식 노드의 주소 도메인(address domain)을 정의하고 있기 때문에, 주소 매핑이 시스템을 최적으로 기술하도록 선택될 수 있다. 아래 코드는 주소에 칩 선택 번호가 포함된 외부 버스에 붙어 있는 장치들의 주소 할당 방법을 보여 준다.

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

            ethernet@0,0 {
                compatible = "smc,smc91c111";
                reg = <0 0 0x1000>;
            };
```

```
        i2c@1,0 {
            compatible = "acme,a1234-i2c-bus";
            reg = <1 0 0x1000>;
            rtc@58 {
                compatible = "maxim,ds1338";
            };
        };

        flash@2,0 {
            compatible = "samsung,k8f1315ebm", "cfi-flash";
            reg = <2 0 0x4000000>;
        };
    };
```

외부 버스는 주소 값을 위해 2 개의 cell 을 사용한다. 그 중 하나는 칩 선택 번호이고, 다른 하나는 시작 번지로 부터의 옵셋(offset)이다. 길이(length) 필드는 offset 필드만이 주소 범위를 가질 수 있으므로 단일 cell 형태로 구성되어 있다.

주소 도메인 노드와 자식 노드에만 국한되어 있기 때문에, 부모 노드 입장에서는 어떠한 주소 기법을 사용할지를 정의함에 있어서 자유롭다. 부모 노드와 자식 노드 이외의 다른 노드들은 로컬 주소 도메인과는 전혀 관련이 없다. 따라서 주소 기법은 도메인 마다 별도로 매핑해 주어야 한다.

### 비 Memory Mapped 장치들

어떤 장치들은 CPU 버스로 직접 제어가 불가능한 형태를 띄기도 한다. 이러한 비 memory mapped 장치는 주소 범위(address ranges)는 가질 수 있으나, CPU 에 의해 직접 접근(제어)이 불가능하다. 그 대신에, 부모 디바이스 드라이버가 CPU 를 대신하여 간접적으로 접근을 대행하게 된다. 아래의 i2c 장치를 예로 들어 보면, 각각의 장치는 주소를 부여 받았으나, 그것과 연관된 길이나 주소 범위는 존재하지 않고 있다. 이는 마치 앞서 설명했던 CPU 주소 할당 방식과 유사해 보인다.

```
        i2c@1,0 {
            compatible = "acme,a1234-i2c-bus";
            #address-cells = <1>;
            #size-cells = <0>;
            reg = <1 0 0x1000>;
            rtc@58 {
                compatible = "maxim,ds1338";
```

```
                reg = <58>;
            };
        };
```

### 주소 범위(주소 변환)

지금까지는 장치들에 주소를 할당하는 방법에 대해 살펴 보았다. 그러나, 장치에 할당된 주소는 장치 자신에 국한된 로컬 주소였다. 아직까지는 이 로컬 주소를 CPU 가 사용 가능한 주소로 전환하는 방법을 소개하지는 않았다. root 노드는 이미 CPU 가 인식하는 주소 공간을 기술하고 있다. 따라서 root 노드의 자식 노드들은 이미 CPU 주소 도메인을 사용하고 있으므로, 별도의 주소 전환 과정이 필요치 않다. 예를 들어 serial@101f0000 장치는 주소 0x101f0000 로 직접 접근이 가능하다. root 노드의 직계 자식이 아닌 경우에는 CPU 주소 도메인을 이용할 수가 없다. 따라서 주소 전환 과정이 필요한데, 이를 위해 디바이스 트리는 **ranges** 라는 속성을 정의하여 사용하고 있다. 아래에 **ranges** 속성을 추가한 예를 제시하였다.

```
/ {
    compatible = "acme,coyotes-revenge";
    #address-cells = <1>;
    #size-cells = <1>;
    ...
    external-bus {
        #address-cells = <2>
        #size-cells = <1>;
        ranges = <0 0   0x10100000    0x10000       // Chipselect 1, Ethernet
                  1 0   0x10160000    0x10000       // Chipselect 2, i2c controller
                  2 0   0x30000000    0x1000000>; // Chipselect 3, NOR Flash

        ethernet@0,0 {
            compatible = "smc,smc91c111";
            reg = <0 0 0x1000>;
        };

        i2c@1,0 {
            compatible = "acme,a1234-i2c-bus";
            #address-cells = <1>;
            #size-cells = <0>;
            reg = <1 0 0x1000>;
            rtc@58 {
                compatible = "maxim,ds1338";
```

```
                reg = <58>;
            };
        };

        flash@2,0 {
            compatible = "samsung,k8f1315ebm", "cfi-flash";
            reg = <2 0 0x4000000>;
        };
    };
};
```

**ranges** 속성은 주소 변환을 위한 정보를 담고 있는데, 각각의 항목을 보면, 자식 주소, 부모 주소, 자식 노드의 주소 공간 영역의 크기가 그 것이다. 각 필드의 크기는 자식의 #address-cells 값, 부모의 #address-cells 값 및 자식의 #size-cells 값에 의해 결정된다. 위의 예에서 외부 버스의 경우에, 자식 주소는 2 개의 cell 로, 부모 주소는 1 개의 cell 로 이루어져 있으며, 자식 cell 의 크기는 1 임을 알 수 있다. 3 개의 **ranges** 값을 정리해 보면 다음과 같다.

- *칩 선택 번호가 0 이고, offset 이 0 인 경우는 0x10100000..0x1010ffff 주소 범위로 매핑됨.*
- *칩 선태 번호가 1 이고, offset 이 0 인 경우는 0x10160000..0x1016ffff 주소 범위로 매핑됨.*
- *칩 선태 번호가 2 이고, offset 이 0 인 경우는 0x30000000..0x10000000 주소 범위로 매핑됨.*

만일 부모와 자식 노드의 주소 공간이 같다면, 빈 정보를 갖는 **ranges** 속성을 추가할 수 있을 것이다. 이는 자식 노드의 주소 공간이 부모 노드의 그것과 1:1 로 매핑 됨을 뜻한다.

1:1 매핑의 경우에도 왜 주소 변환이 사용되는지 궁금할 수 있을 것이다. PCI 같은 버스는 전혀 다른 주소 공간(address spaces)을 갖고 있으며, 이에 대한 자세한 사항을 OS 에 알려줄 필요가 있다. 또 어떤 것들은 버스상의 실제 주소를 알아야만 하는 DMA 엔진을 가지기도 한다. 때때로 장치들은 동일 그룹으로 묶이기도 하는데, 이는 각 장치들이 동일한 소프트웨어 프로그래밍 가능하는 물리 주소 매핑(software programmable physical address mapping) 방식을 공유하기 때문이다. 1:1 매핑을 사용할지 말지는 OS 가 필요로 하는 정보나 하드웨어 설계 방식에 전적으로 달려있다 할 것이다.

i2c@1,0 노드에는 **ranges** 속성이 없음을 알 수 있다. 그 이유는 외부 버스 노드와는 달리, i2c bus 에 장착된 장치들은 CPU 주소 도메인에 매핑되지 않기 때문이다. 그 대신에 CPU 는 the i2c@1,0 장치를 경유하여 rtc@58 장치를 간접적으로 접근하게 된다. **ranges** 속성이 없다는 것은 부모 노드 이외의 다른 다른 장치에서는 그 장치를 직접적으로 접근할 수 없음을 뜻한다.

## 인터럽트 동작 방식 이해(How Interrupts Work)

트리의 기본 구조를 따르는 주소 범위 변환과는 달리, 인터럽트 신호는 한 노드에서 시작해서 다른 장치 노드에서 끝나는 방식이다. 따라서 다른 장치 표현 방식과는 달리, 인터럽트 신호를 표현하기 위해서는 노드 간의 링크 개념이 사용된다. 아래 4 가지의 속성이 인터럽트 연결을 묘사하기 위해 사용된다.

- ***interrupt-controller*** *- 인터럽트 신호를 받는 장치를 노드로 표현. 빈 속성 값을 가짐.*
- ***#interrupt-cells*** *- 인터럽트 컨트롤러 노드의 속성이다. 얼마나 많은 cell 이 인터럽트 콘트롤러를 위한 인터럽트 지정자(interrupt specifier) 내에 있는지를 기술한다(마치 #address-cells 나 #size-cells 과 유사함)*
- ***interrupt-parent*** *- 인터럽트 컨트롤러에의 phandle 를 포함하고 있는 장치 노드에 대한 속성. 이 속성을 가지고 있지 않는 노드는 부모 노드로부터 속성을 상속받을 수 있다.*
- ***interrupts*** *- 인터럽트 지정자(interrupt specifier)의 목록을 포함하는 장치 노드의 속성. 각 인터럽트 출력 신호당 1 개씩.*

인터럽트 지정자(specifier)는 장치가 어느 인터럽트 입력에 연결되어 있는지를 나타내는 한 개 이상의 cell 이다(#interrupt-cells 로 표시함). 대부분의 장치는 아래 예에서 볼 수 있는 것처럼, 한 개의 인터럽트 출력을 가지고 있으나, 여러 개의 인터럽트 출력을 갖는 것들도 있다. 인터럽트 지정자의 의미는 전적으로 인터럽트 컨트롤러 장치에 대한 바인딩(binding)과 연관이 있다. 각각의 인터럽트 컨트롤러는 인터럽트 입력을 정의하기 위하여 얼마나 많은 cell 이 필요한지를 결정할 수 있다. 다음 코드는 Coyote's Revenge 샘플 머신에 인터럽트 연결 부분을 추가한 예이다.

```
/ {
    compatible = "acme,coyotes-revenge";
    #address-cells = <1>;
    #size-cells = <1>;
    interrupt-parent = <&intc>;

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

```
serial@101f0000 {
        compatible = "arm,pl011";
        reg = <0x101f0000 0x1000 >;
        interrupts = < 1 0 >;
};

serial@101f2000 {
        compatible = "arm,pl011";
        reg = <0x101f2000 0x1000 >;
        interrupts = < 2 0 >;
};

gpio@101f3000 {
        compatible = "arm,pl061";
        reg = <0x101f3000 0x1000
               0x101f4000 0x0010>;
        interrupts = < 3 0 >;
};

intc: interrupt-controller@10140000 {
        compatible = "arm,pl190";
        reg = <0x10140000 0x1000 >;
        interrupt-controller;
        #interrupt-cells = <2>;
};

spi@10115000 {
        compatible = "arm,pl022";
        reg = <0x10115000 0x1000 >;
        interrupts = < 4 0 >;
};

external-bus {
        #address-cells = <2>
        #size-cells = <1>;
        ranges = <0 0   0x10100000    0x10000        // Chipselect 1, Ethernet
                  1 0   0x10160000    0x10000        // Chipselect 2, i2c controller
```

```
                2 0  0x30000000  0x1000000>; // Chipselect 3, NOR Flash

        ethernet@0,0 {
            compatible = "smc,smc91c111";
            reg = <0 0 0x1000>;
            interrupts = < 5 2 >;
        };

        i2c@1,0 {
            compatible = "acme,a1234-i2c-bus";
            #address-cells = <1>;
            #size-cells = <0>;
            reg = <1 0 0x1000>;
            interrupts = < 6 2 >;
            rtc@58 {
                compatible = "maxim,ds1338";
                reg = <58>;
                interrupts = < 7 3 >;
            };
        };

        flash@2,0 {
            compatible = "samsung,k8f1315ebm", "cfi-flash";
            reg = <2 0 0x4000000>;
        };
    };
};
```

위의 내용 중 주목할 만한 사항을 요약하면 다음과 같다:

- *위의 머신에는 하나의 인터럽트 컨트롤러가 있음. 즉, interrupt-controller@10140000.*
- *'intc:' 라는 이름의 레이블이 인터럽트 컨트롤러 노드에 붙어 있음. 이 레이블은 root 노드에 있는 interrupt-parent 속성에 phandle 을 부여하는 용도로 사용되었음. interrupt-parent 값은 모든 자식 노드가 다시 interrupt-parent 속성을 재 정의하여 사용하지 않는 한, 자동으로 상속되므로, 시스템 전체의 기본 값이 된다.*
- *각 장치는 서로 다른 인터럽트 입력 선을 표시하기 위해 interrupt 속성을 사용한다.*
- *#interrupt-cells 는 2 임. 따라서 각각의 인터럽트 지정자는 2 개의 cell 을 가지고 있음. 위의 예는 첫 번째 cell 은 interrupt 라인 번호를 표시하기 위하여, 두 번째 cell 은 인터럽트 관련*

*플래그(active high, active low, edge, level 트리거 등)를 표시하기 위해 사용되고 있음. 지정자(Specifier)가 어떻게 표시되는지를 알아보기 위해서는 인터럽트 컨트롤로 관련 바인딩(binding) 문서를 참조하기 바란다.*

## 디바이스 전용 데이터(Device Specific Data)

일반 속성 이외에도 임의의 속성이나 자식 노드가 추가될 수 있다. 아래 규칙만 잘 지킨다면, OS가 필요로 하는 여러 데이터에 대한 추가가 가능하다.

첫째, 새로운 장치 관련 속성 명은 제조사명으로 시작해야 한다. 이는 기존 표준 속성 명칭과의 충돌을 피하기 위함이다.

둘째로, 속성과 자식 노드의 의미는 바인딩(binding) 문서로 남겨져야 한다. 그래야 디바이스 드라이버를 작성하는 사람이 그 의미를 해석할 수 있을 터이니 말이다. 바인딩 문서에 포함될 사항으로는 새로운 정한 compatible 값의 의미, 어떠한 속성을 갖는지, 어떠한 자식 노드가 필요한지, 어떠한 장치를 표현하고자 하는 지 등이다. 고유한 compatible 값을 역시 고유의 binding을 갖게 된다.

셋째로, 새로운 바인딩 문서를 devicetree-discuss@lists.ozlabs.org 메일링 리스트(mailing list)로 보내어, 다른 사람들이 검토할 수 있도록 해야 한다. 이 과정을 통해 흔히 할 수 있는 여러 실수를 찾아낼 수 있게 된다.

## 특수 노드(Special Nodes)

### 별명 노드(aliases Node)

어떤 노드는 /external-bus/ethernet@0,0 처럼 전체 경로로 표현해야 할 때가 있다. 그러나, 이는 사용자가 정말로 알고 싶은 내용이 "어느 장치가 eth0 이지 ?"인 순간에 문제가 될 수 있다. aliases 노드는 전체 경로에 대한 줄임 표현을 하고 싶을 때 사용한다. 예를 들어:

```
aliases {
        ethernet0 = &eth0;
        serial0 = &serial0;
    };
```

OS 입장에서는 장치에 식별자(identifier)를 할당하려 할 때 별명(aliases) 속성을 사용하는 것이 바람직하다. 아래와 같은 새로운 문법(syntax)이 사용되고 있음을 알 수 있는데, 이는 문자열 속성으로써 label 이 참조하는 전체 노드 경로를 할당하는 것을 의미한다.

*property* = &*label*;

그러나, 이것은 앞서 보았던 cell 에 phandle 값을 추가하는 *phandle* = < &*label* >; 형식 과는 차이가 있음을 알아야 한다.

### 선택 노드(chosen Node)

**chosen** 노드는 실제 장치를 표현하는 것은 아니고, 펌웨어(firmware or bootloader)와 OS 간에 데이터(예를 들어, boot argument)를 전달하는 공간으로 사용된다. **chosen** 노드 내에 있는 데이터는 하드웨어를 표현하지는 않는다. 대개 .dts 소스 파일 내의 **chosen** 노드는 빈 상태로 남아 있으며, 부팅 시점에 채워지게(populate) 된다.

지금까지 설명한 예제에서는 chosen 노드에 아래 내용이 추가되었음을 알 수 있다:

```
chosen {
    bootargs = "root=/dev/nfs rw nfsroot=192.168.1.1 console=ttyS0,115200";
};
```

# 3. Common Clock Framework

대부분의 칩(electronic chips)은 clock에 의해 동작한다. SoC 혹은 보드 내의 주변 장치를 위한 clock은 전체적으로 보면 하나의 트리 형태로 구성되어 있다. Clock을 제어하는 목적은 파워 관리(power management)와 기준 시간(time reference) 관리를 위해서이다.

- ***파워 관리**: clock의 주파수는 동적 파워 소비와 연관이 있다. 즉, clock 주파수를 크게 할수록 전력 소비가 커지게 되고, 작게 할수록 전력 소비도 그 만큼 줄어든다.*
- ***기준 시간 관리**: clock은 baud-rate이나 pixel clock 등을 계산하는 용도로 사용될 수 있다.*

Linux에서 clock 프레임워크는 아주 오랫동안 사용되어 왔는데, 디바이스 드라이버에서 사용하는 주요 clock API로는 다음과 같은 것들이 있다.

*clk_get(), clk_enable(), clk_get_rate(), clk_set_rate(), clk_disable(), clk_put(),...*

전통적인(legacy) clock 프레임워크의 몇 가지 단점 혹은 한계점을 요약해 보면 다음과 같다.

- *칩 제조사는 자체적으로 clock API 내부를 구현하여 사용하고 있음.*
- *코드 공유나 일반화된 메커니즘을 허용하지 않고 있음.*
- *ARM multiplatform 커널(하나의 커널 코드로 여러 플랫폼 지원 가능) 개념에도 맞지*

*않음.*

Legacy clock 프레임워크의 문제점 및 한계를 극복하기 위하여 2010년 초 Jeremy Kerr에 의하여 새로운 common clk 프레임워크(CCF: Common Clock Framework)가 소개되었으며, Mike Turquette의 노력으로 2012년 5월 kernel 3.4에 최종적으로 통합되기에 이르렀다. 이들이 작업한 내용을 살펴 보면, 크게 4가지로 요약해 볼 수 있다.

- *clock 프레임워크 API 및 몇 가지 기본 clock 드라이버를 구현하였으며, custom clock 드라이버 구현도 가능하도록 하였다.*
- *또한, clock을 device tree와도 연계시켜, device tree 내에서 clock을 선언하고 이를 각종 장치와 연결시킬 수 있도록 하였다. 물론 예전 방식대로 device tree를 사용하지 않고, source code내에서 직접 사용하는 것도 가능하도록 하였다.*
- *debugfs에도 clock tree를 표시하는 기능을 제공했으며,*
- *drivers/clk 디렉토리에 code를 구현하였다.*

Common Clock 프레임워크를 그림으로 표현하면 다음과 같다.

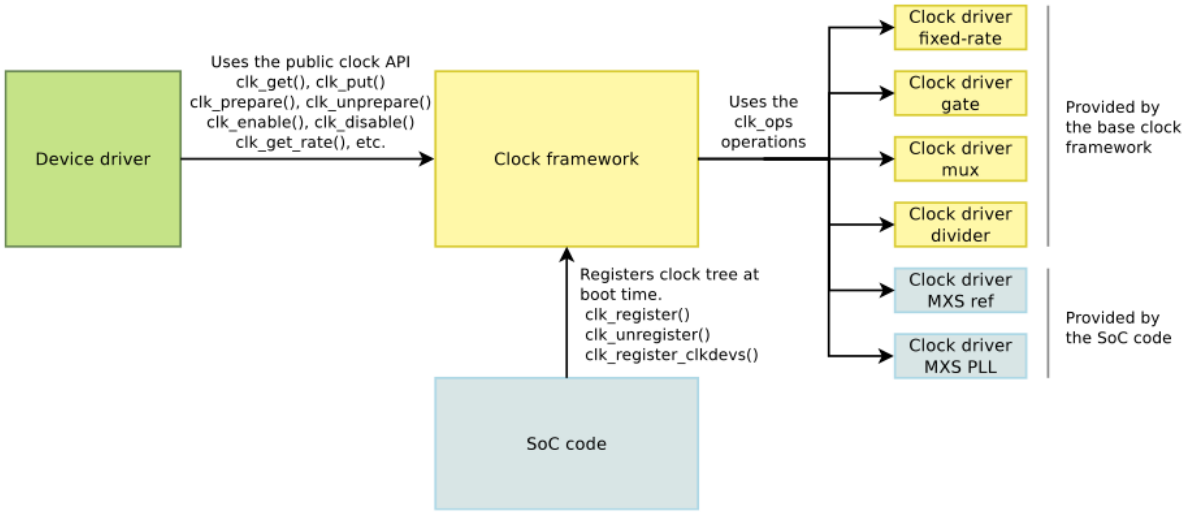


**그림 4-7 신규 Clock Framework 구조 [출처 – 참고 문헌 1-6]**

# 3.1 CCF 인터페이스

CCF 인터페이스는 크게 CCF 코어(Common Clock Framework core)와 하드웨어 종속 파트(Hardware-specific) 두 부분으로 구분이 가능하다.

먼저 CCF 코어에서는

- *struct clk를 정의하고 있으며,*
- *clk.h(defined in drivers/clk/clk.c) 파일에 clk API를 정의하고 있다.*
- *또한, clk API 내부에서 호출하는 여러 operation은 struct clk_ops로 구현하였으며,*

- *드라이버를 새로 추가한다 하더라도 CCF 코어는 수정이 안되어야 함을 가정하였다.*

다음으로 하드웨어 종속 파트에서는

- *struct clk_ops로 등록한 callback 함수 및 하드웨어 종속 데이터 구조(예를 들어 struct clk_foo라고 해보자)를 구현해야 하며,*
- *이는 모든 신규 하드웨어 clock에 대해 적용되어야 한다.*

이 두 파트는 struct clk_hw에 의해 서로 연결된다. 가령 struct clk_hw는 struct clk_foo 내에 정의되게 되며, 이는 다시 struct clk을 참조하게 된다.

```
struct clk_foo {
        struct clk_hw hw;
        [...]   /* hw specific data */
}

struct clk_hw {
        struct clk *clk;
        [...]
}

struct clk {
        [...]
}
```

# 3.2 CCF 코어 구현

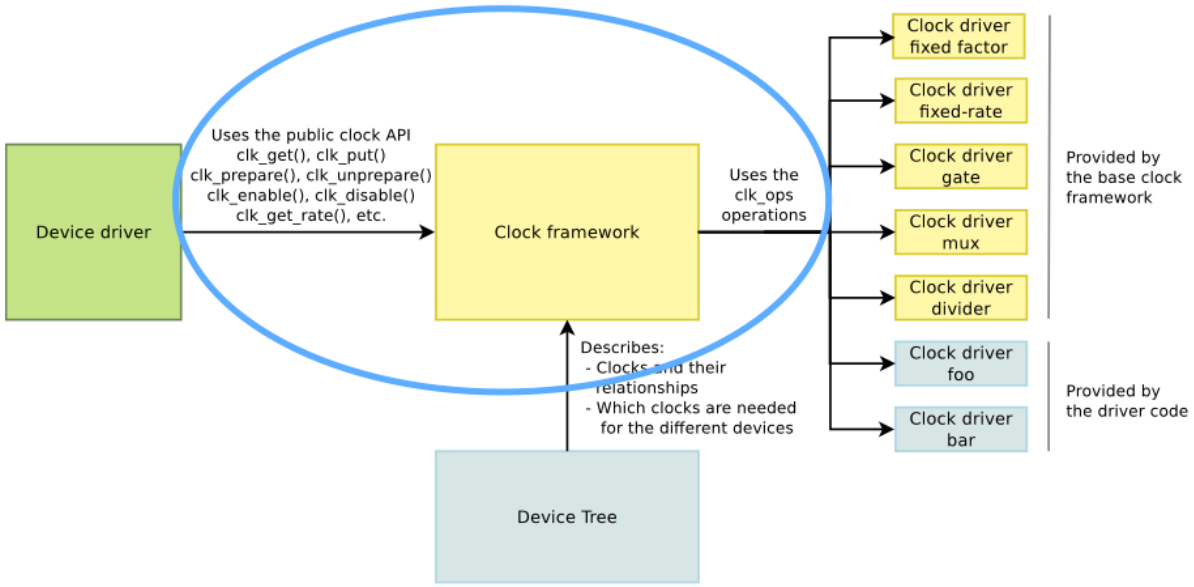


**그림 4-8 신규 Clock Framework 구조 해부(1) – CCF 코어 구현 [출처 – 참고 문헌 1-6]**

drivers/clk/clk.c 파일에 있는 CCF 코어를 요약하면, 다음과 같다.

- *clock tree를 유지하며,*
- *clk_enable()/clk_disable()에 대해 global spinlock과 모든 다른 operation에 대해 global mutex를 사용하여 concurrency 문제를 해결하고 있다.*
- *clock tree를 통해 동작(operations)을 처리하고 있으며(???),*
- *clock에 변화(rate change)가 발생할 경우, 기 등록 callback이 호출되도록 하고 있다.*

include/linux/clk-private.h 파일에 있는 struct clk 데이터 구조를 살펴보면 다음과 같다:

```
struct clk {
        const char *name;
        const struct clk_ops *ops;
        struct clk_hw *hw;
        char **parent_names;
        struct clk **parents;
        struct clk *parent;
        struct hlist_head children;
        struct hlist_node child_node;
        [...]
};
```

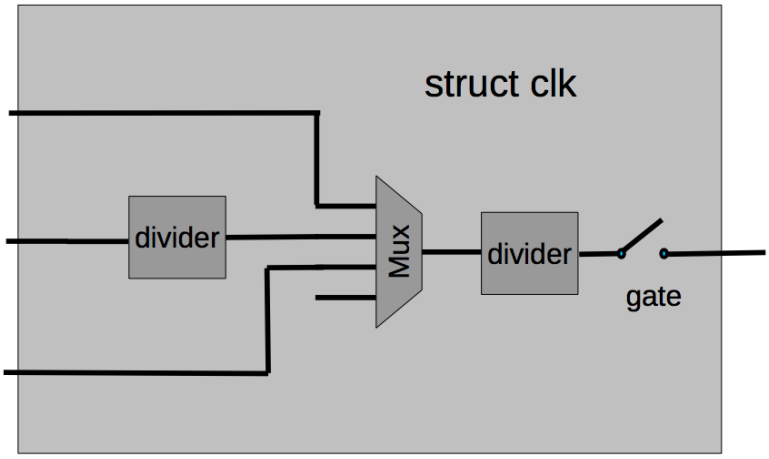


**그림 4-9 struct clk**

clk_set_rate() 함수를 예로 들어 보자:

```
int clk_set_rate(struct clk *clk, unsigned long rate)
{
    struct clk *top, *fail_clk;
    int ret = 0;
    /* prevent racing with updates to the clock topology */
    mutex_lock(&prepare_lock);
    /* bail early if nothing to do */
    if (rate == clk->rate)
        goto out;
    if ((clk->flags & CLK_SET_RATE_GATE) && clk->prepare_count) {
        /* clock rate 설정은 clock이 아직 준비되지 않았을 경우
        (ungated or not yet prepared)에만 가능하다 */
        ret = -EBUSY;
        goto out;
    }
    /* calculate new rates and get the topmost changed clock */
    top = clk_calc_new_rates(clk, rate);
    [...] /* Exit with error if clk_ calc_ new_ rates( ) failed */

    /* notify that we are about to change rates */
    fail_clk = clk_propagate_rate_change(top, PRE_RATE_CHANGE);
    if (fail_clk) {
        pr_warn("%s: failed to set %s rate₩n", __func__, fail_clk->name);
        clk_propagate_rate_change(top, ABORT_RATE_CHANGE);
        ret = -EBUSY;
```

```
            goto out;
        }
        /* change the rates */
        clk_change_rate(top);
        /* Actually set the rate using the hardware operation */
        out:
        mutex_unlock(&prepare_lock);
        return ret;
}
```

# 3.3 하드웨어 Clock의 구현

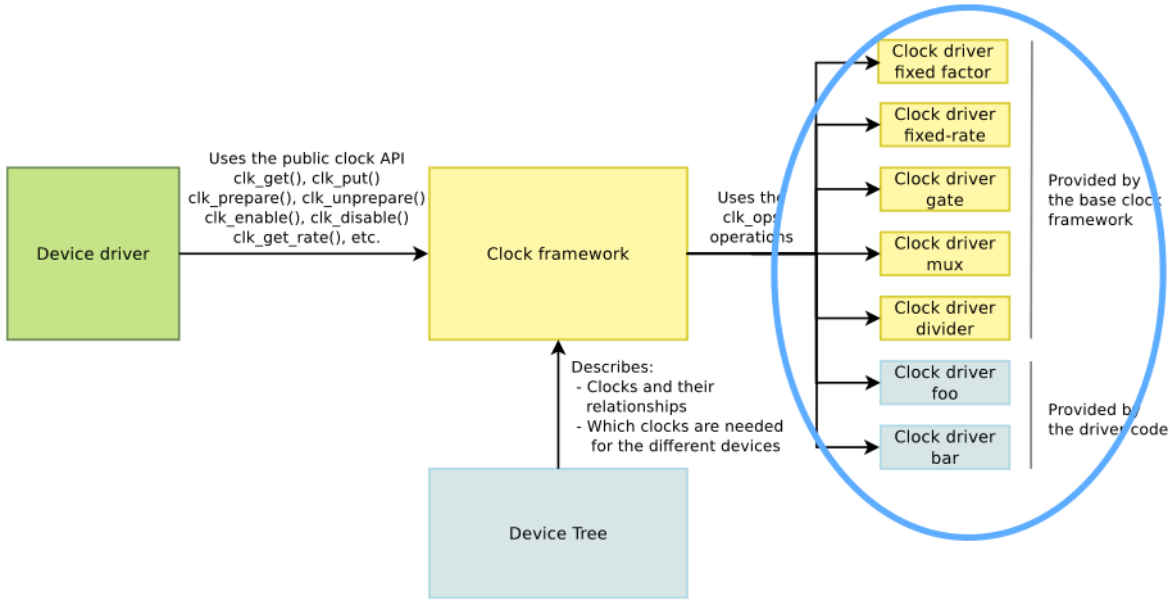


**그림 4-10 신규 Clock Framework 구조 해부(2) – 하드웨어 종속 파트 [출처 – 참고 문헌 1-6]**

하드웨어 clock을 구현하기 위해서 알아야 할 사항을 정리해 보면 다음과 같다.

- *.ops and .hw 포인터와 연관이 있다.*
- *하드웨어 종속 데이터로부터 struct clk의 세부 사항을 구현해야 한다(Abstracts the details of struct clk from the hardware-specific bits)*
- *모든 operation을 구현할 필요는 없으며, clock의 유형에 따라서 몇몇 강제 사항만 구현하면 된다.*
- *clk_register()* 함수를 써서 operation set을 등록해 주면 clock이 생성되게 된다.

include/linux/clk-provider.h 파일에 정의되어 있는 하드웨어 clock operation 데이터 구조를 살펴보면 다음과 같다.

```
struct clk_ops {
        int (*prepare)(struct clk_hw *hw);
        void (*unprepare)(struct clk_hw *hw);
        int (*enable)(struct clk_hw *hw);
        void (*disable)(struct clk_hw *hw);
        int (*is_enabled)(struct clk_hw *hw);
        unsigned long (*recalc_rate)(struct clk_hw *hw, unsigned long parent_rate);
        long (*round_rate)(struct clk_hw *hw, unsigned long, unsigned long *);
        int (*set_parent)(struct clk_hw *hw, u8 index);
        u8 (*get_parent)(struct clk_hw *hw);
        int (*set_rate)(struct clk_hw *hw, unsigned long);
        void (*init)(struct clk_hw *hw);
};
```

위의 clock operation callback에 대하여 clock이 가진 능력(capability)에 따라 구현해야 할 내용을 표로 정리해 보면 다음과 같다.

**표 4-1 clock 능력과 clock operation과의 관계**

| | gate | change rage | single parent | multiplexer | root |
|---|---|---|---|---|---|
| **.prepare**<br>**.unprepare** | | | | | |
| **.enable**<br>**.disable**<br>**.is_enabled** | y<br>y<br>y | | | | |
| **.recalc_rate**<br>**.round_rate**<br>**.set_rate** | | y<br>y<br>y | | | |
| **.set_parent**<br>**.get_parent** | | | n<br>n | y<br>y | n<br>n |
| **.init** | | | | | |

(참고: y = 강제 사항, n = 불필요하거나, 무의미한 사항)

지금부터는 clk_ops 데이터 구조를 구성하는 주요 callback 함수의 의미를 자세히 설명해 보기로 한다.

## 1) Clock을 사용 가능하게 만들기

Clock을 사용 가능하게 하는 API는 크게 두 가지로 분류가 가능하다:

1) **.prepare(/.unprepare):**
   - Clock을 ungate하기 전에 clock을 준비시키기 위해 호출함.
   - 어떤 경우(가령, I2C를 통해 접근)에는 enable() 함수 대신에 호출되기도 함.
   - Sleep 할 수도 있음(process context 임)
   - 따라서 atomic context(interrupt context)에서는 호출 불가함
2) **.enable(/.disable):**
   - 일단 clock이 준비(prepared)되었으면, clock을 ungate 시키기 위해 호출함.
   - 어떤 경우(예를 들어 SoC 내의 단일 레지스터를 통해 접근하는 경우)에는 prepare() 함수를 대신하여 호출되기도 함.
   - Sleep 할 수 없음(atomic context에서 동작함)
   - Atomic context(interrupt context)에서 호출됨.
   - .is_enabled: enable 횟수를 검사하는 대신에, clock이 enable되었는지를 결정하기 위하여 하드웨어에게 물어 보는 함수.

| | Fast Clock | Slow Clock |
|---|---|---|
| clk_prepare() | | enable() |
| clk_enable() | enable() | |
| clk_disable() | disable() | |
| clk_unprepare | | disable() |

**그림 4-11 fast clock과 slow clock 비교**

## 2) Rate 관리

1) **.round_rate**:
   - clock이 지원하는 가장 근접한 rate 값을 리턴해 줌. Propagation 중에 clk_route_rate() 이나 clk_set_rate() 함수에 의해 호출됨.
2) **.set_rate**:
   - clock의 rate를 바꿔 줌. Propagation 중 혹은 clk_set_rate() 함수에 의해 호출됨.
3) **.recalc_rate**:
   - 하드웨어에 확인 후, clock의 rate 값을 재 계산해 줌. 내부적으로 clock tree를 갱신하기 위해 사용됨.

### 3) Parent 관리

위의 표 4-1에서 본 바와 같이, multiplexer를 위해서만 사용된다.

1) **.get_parent:**

- Clock의 parent를 결정하기 위하여 하드웨어게 요청(확인함)
- 현재는 clock이 정적 초기화되었을 경우에만 사용됨.
- clk_get_parent() 함수는 이를 사용하지 않으며, 단순히 clk->parent 내부 데이터 구조체만 리턴함.

2) **.set_parent**:

- Clock의 입력 단(input source)을 바꿔 줌.
- .parent_names 이나 .parents 배열 중에 인덱스 값을 받아들임.
- clk_set_parent() 함수는 인덱스 내의 clk을 바꿔줌(clk_set_parent() translate clk in index)

### 4) 기본 Clock

CCF는 다음과 같은 5개의 기본 clock을 제공한다:

**1) fixed-rate**:

- 항상 동작하며, 항상 동일한 rate를 제공함.

**2) gate**:

- parent clock과 동일한 rate를 가지며, gate되거나 ungate될 수 있음.

**3) mux**:

- 여러 clock 중 parent를 선택하도록 허용하며, 선택된 parent로부터 rate 값을 얻어옴. Gate/ungate 불가 함.

**4) fixed-factor**:

- parent rate 값을 상수 값으로 나누고, 곱해 줌. Gate/ungate 불가 함.

**5) divider**:

- parent rate 값을 나눔. 나누는 값은 등록 시 제공된 배열에서 선택할 수 있음. Gate/ungate 불가 함.

대 부분의 clock 들은 이 5개의 기본 clock 중의 하나를 사용하여 등록할 수 있다. 복잡한 하드웨어 clock은 기본 clock으로 나누어져야 한다. 예를 들어, fixed rate을 가진 gate clock은 parent clock으로 fixed rate clock이 위치해야 할 것이다. 최근에 clk-composite 라는 새로운 clock이 추가되었는데, 이 clock의 역할은 기본 clock의 기능을 하나의 clock으로 통합하도록 하는 것이다. 하지만, 아직 받아들여지지는 않고 있다(계속 검토 중임).

### 5) 정적 초기화

기존의 복잡한 SoC code(수백 개의 clock을 정적으로 정의하여 사용함)를 CCF로 쉽게 migration 시킬 목적으로 만들어 두었다. 기존 코드를 재사용하기 위해 include/linux/clk-private.h 파일과 __clk_init() 함수를 어쩔 수 없이 포함시키게 되었다. 그러나, 이렇게 동적으로 초기화하여 사용할 수는 있으나, 권장하지는 않는 방법이다. 당연한 얘기지만, 신규 플랫폼을 구현할 경우에는 절대로 clk-private.h를 사용해서는 안 된다(강제 조항). Clock은 오직 clk_init_data를 사용하는

clk_register() 함수 호출을 통해서만 초기화 되어야 한다(참고로, clk_init_data는 clk_hw과 연결되어 있다).

## 3.4 Clock과 디바이스 트리(device tree)

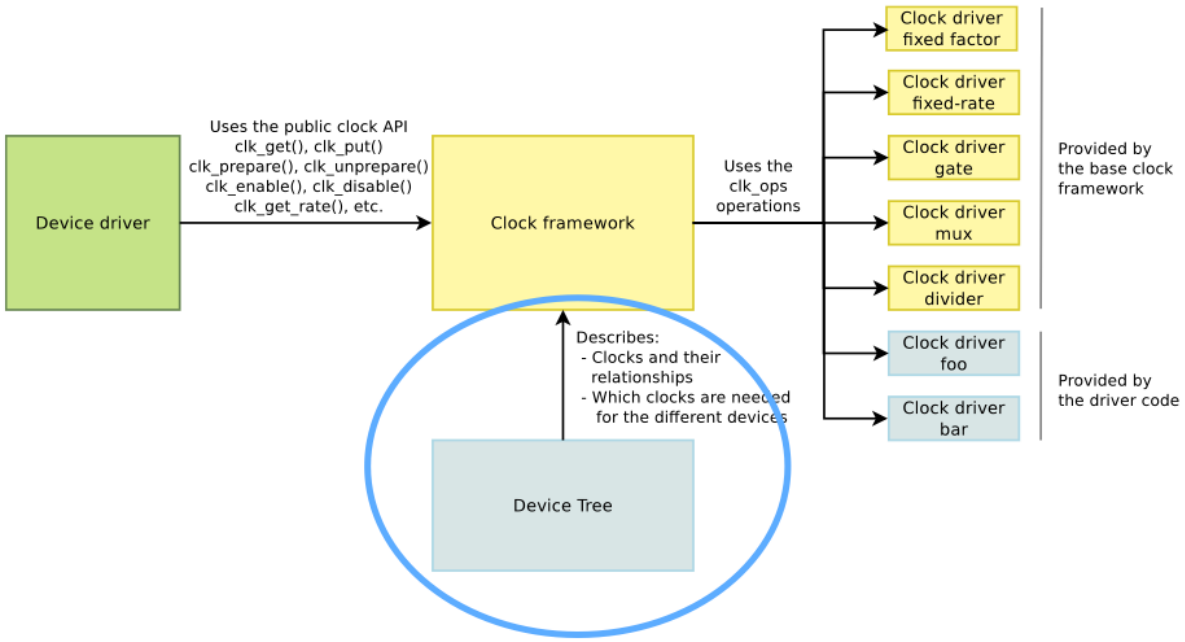


**그림 4-12 신규 Clock Framework 구조 해부(3) [출처 – 참고 문헌 1-6]**

Clock을 정의하고 clock resource를 얻어 오려면 디바이스 트리를 사용해야 한다(최신 kernel에서 선호하는 방법임). 이 과정을 간략히 요약하면, 먼저 Clock을 설정하기 위해 device tree 구문을 파싱하여, 관련 속성 정보를 추출한 후, 다음으로 Compatible clock과 매칭시키기 위해 struct of_device_id 배열을 만들고, 마지막으로 각각의 노드에 대해 data와 setup 함수를 결합시켜는 과정을 거쳐야 한다.

**Device Tree로 clock 선언: 간단한 예제(1)**

```
From arch/arm/boot/dts/ecx-common.dtsi
[...]
        osc: oscillator {
                #clock-cells = <0>;
                compatible = "fixed-clock";
                clock-frequency = <33333000>;
        };
        ddrpll: ddrpll {
```

```
            #clock-cells = <0>;
            compatible = "calxeda,hb-pll-clock";
            clocks = <&osc>;
            reg = <0x108>;
        };
[...]
```

**Device Tree 처리하기: 간단한 예제(1)**

```
From drivers/clk/clk-highbank.c
static const __initconst struct of_device_id clk_match[] = {
        { .compatible = "fixed-clock", .data = of_fixed_clk_setup, },
        [...]
};
void __init highbank_clocks_init(void)
{
        of_clk_init(clk_match);
}
```

```
From drivers/clk/clk.c
void __init of_clk_init(const struct of_device_id *matches)
{
        struct device_node *np;
        for_each_matching_node(np, matches) {
                const struct of_device_id *match = of_match_node(matches, np);
                of_clk_init_cb_t clk_init_cb = match->data;
                clk_init_cb(np);
        }
}
```

**Device Tree 처리하기: 간단한 예제(2)**

```
From drivers/clk/clk-fixed-rate.c
void __init of_fixed_clk_setup(struct device_node *node)
{
        struct clk *clk;
        const char *clk_name = node->name;
        u32 rate;
```

```
    if (of_property_read_u32(node, "clock-frequency", &rate))
        return;
    of_property_read_string(node, "clock-output-names", &clk_name);
    clk = clk_register_fixed_rate(NULL, clk_name, NULL,
    CLK_IS_ROOT, rate);
    if (!IS_ERR(clk))
        of_clk_add_provider(node, of_clk_src_simple_get, clk);
}
```

**Device Tree로 clock 선언: 고급 예제(1)**

```
From arch/arm/boot/dts/armada-xp.dtsi
    [...]
    coreclk: mvebu-sar@d0018230 {
        compatible = "marvell,armada-xp-core-clock";
        reg = <0xd0018230 0x08>;
        #clock-cells = <1>;
    };
    cpuclk: clock-complex@d0018700 {
        #clock-cells = <1>;
        compatible = "marvell,armada-xp-cpu-clock";
        reg = <0xd0018700 0xA0>;
        clocks = <&coreclk 1>;
    };
    [...]
```

**Device Tree 처리하기: 고급 예제(1)**

```
From drivers/clk/mvebu/clk-core.c (some parts removed)
static const struct core_clocks armada_370_core_clocks = {
    .get_tclk_freq = armada_370_get_tclk_freq,
    .num_ratios = ARRAY_SIZE(armada_370_xp_core_ratios),
};
static const __initdata struct of_device_id clk_core_match[] = {
    [...]
    {
        .compatible = "marvell,armada-xp-core-clock",
        .data = &armada_xp_core_clocks,
    },
```

```
	[...]
};
void __init mvebu_core_clk_init(void)
{
	struct device_node *np;
	for_each_matching_node(np, clk_core_match) {
		const struct of_device_id *match =
		of_match_node(clk_core_match, np);
		mvebu_clk_core_setup(np, (struct core_clocks *)match->data);
	}
}
```

**Device Tree 처리하기: 고급 예제(2)**

```
From drivers/clk/mvebu/clk-core.c (some parts removed)
static void __init mvebu_clk_core_setup(struct device_node *np, struct core_clocks *coreclk)
{
	const char *tclk_name = "tclk";
	void __iomem *base;
	base = of_iomap(np, 0);
	/* Allocate struct for TCLK, cpu clk, and core ratio clocks */
	clk_data.clk_num = 2 + coreclk->num_ratios;
	clk_data.clks = kzalloc(clk_data.clk_num * sizeof(struct clk *), GFP_KERNEL);
	/* Register TCLK */
	of_property_read_string_index(np, "clock-output-names", 0, &tclk_name);
	rate = coreclk->get_tclk_freq(base);
	clk_data.clks[0] = clk_register_fixed_rate(NULL, tclk_name, NULL, CLK_IS_ROOT, rate);
	[...]
}
```

# 3.5 Clock과 디바이스 드라이버

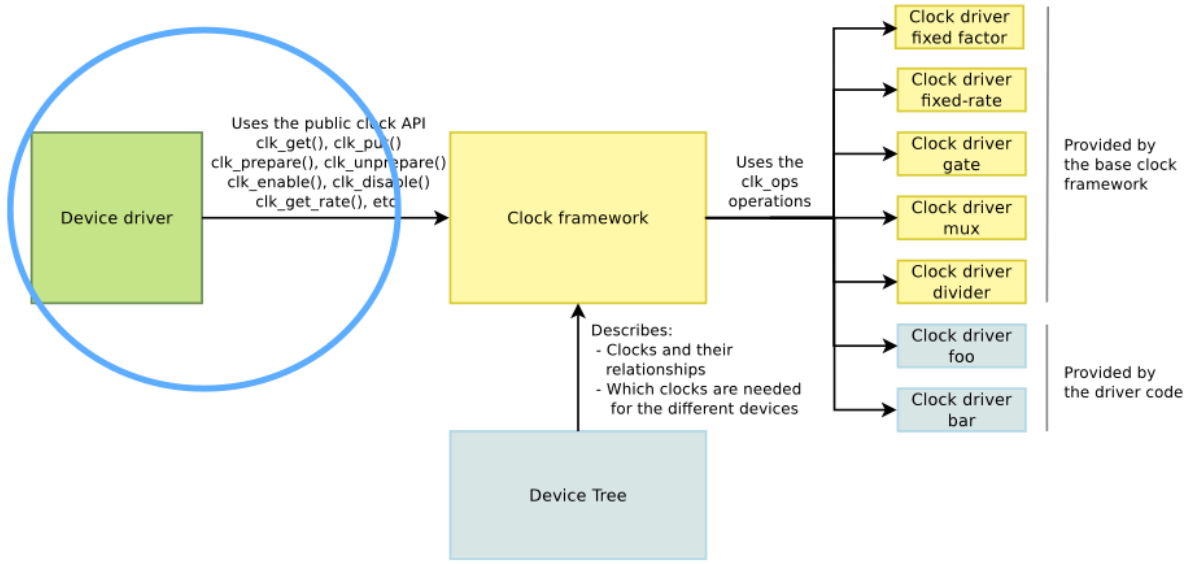


**그림 4-13 신규 Clock Framework 구조 해부(4) [출처 – 참고 문헌 1-6]**

디바이스 드라이버에서 clock을 다루는 절차를 소개하면 다음과 같다.

*1) 디바이스 드라이버스에서 clock을 얻어 사용하기 위해서는 clk_get() 함수를 사용한다.*
*2) Clock driver와 device를 연결하는 방법으로는 platform data를 활용하는 방법(예전 기법)과 device tree를 이용한 방법(신규 기법)이 존재하는데, 둘 중의 하나의 방법을 써서 driver와 device를 연결해 준다.*
*3) 다음으로, clk_enable()이나 clk_prepare() 함수를 사용하여, clock을 활성화 시킨다. 실제로 둘 중 어떤 함수를 사용해야 하는지는 context 상황(interrupt context 혹은 process context)에 따라 달라 지게 된다.*
*4) 마지막으로 나머지 clock API를 사용하여 clock을 조작한다.*

**Device Tree에서 clock을 참조하는 장치 예**

```
From arch/arm/boot/dts/armada-xp.dtsi
ethernet@d0030000 {
        compatible = "marvell,armada-370-neta";
        reg = <0xd0030000 0x2500>;
        interrupts = <12>;
        clocks = <&gateclk 2>;
        status = "disabled";
};

From arch/arm/boot/dts/highbank.dts
watchdog@fff10620 {
```

```
        compatible = "arm,cortex-a9-twd-wdt";
        reg = <0xfff10620 0x20>;
        interrupts = <1 14 0xf01>;
        clocks = <&a9periphclk>;
};
```

**드라이버에서 clock을 사용하는 예제(Device Tree가 아닌데 ???)**

```
From drivers/net/ethernet/marvell/mvneta.c
static void mvneta_rx_time_coal_set(struct mvneta_port *pp, struct mvneta_rx_queue *rxq, u32
value)
{
        [...]
        clk_rate = clk_get_rate(pp->clk);
        val = (clk_rate / 1000000) * value;
        mvreg_write(pp, MVNETA_RXQ_TIME_COAL_REG(rxq->id), val);
}

static int mvneta_probe(struct platform_device *pdev)
{
        [...]
        pp->clk = devm_clk_get(&pdev->dev, NULL);
        clk_prepare_enable(pp->clk);
        [...]
}

static int mvneta_remove(struct platform_device *pdev)
{
        [...]
        clk_disable_unprepare(pp->clk);
        [...]
}
```

# 4. Pin Control Subsystem

SoC를 설계하다 보면, 실제로 외부로 나오는 핀(pin)의 개수 보다 더 많은 수의 주변 장치를 달아

야 하는 경우가 생긴다. 따라서 이 경우, 몇 개의 핀을 모아 다용도(가령, function A, function B, function C, 혹은 GPIO로 사용)로 활용(multiplexing. 혹은 줄여서 먹스라고 부르겠음)하게 되는데, 이렇게 사용되는 것들로는 다음과 같은 것들을 생각해 볼 수 있다.

- *parallel LCD lines*
- *I2C bus: SDA/SCL lines*
- *SPI: MISO/MOSI/CLK lines*
- *UART: RX/TX/CTS/DTS lines*

이러한 먹스 기능(muxing)은 소프트웨어에 의해 조정 가능한 형태이며, 보드에 따라서 활용 방식이 서로 다를 수 있다(즉, 정의해서 사용하기 나름임). 그림 4-14는 이러한 핀먹스 기능을 그림으로 표현한 것이다.

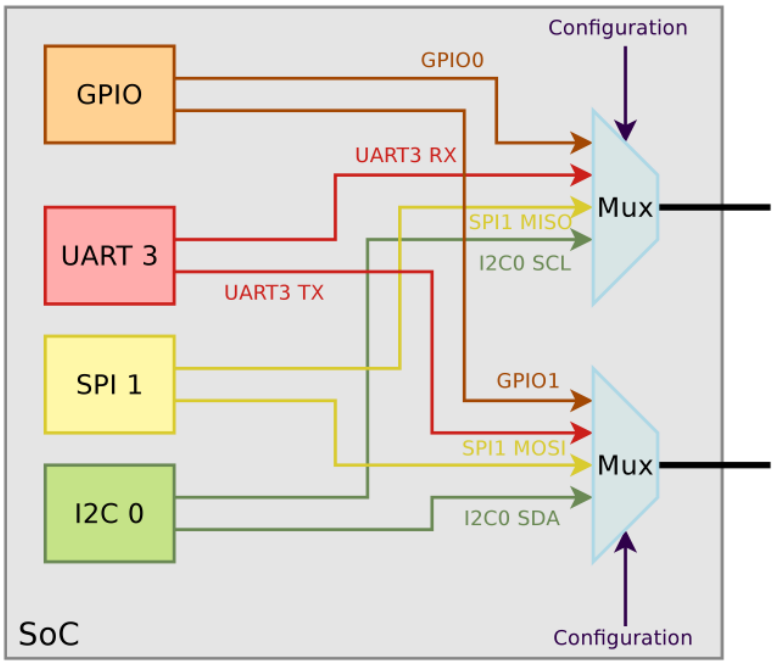


**그림 4-14 Pin Muxing의 개념 [출처 – 참고 문헌 1-6]**

앞으로 소개할 새로운 핀먹스(pin muxing) 기능과 비교해서, 지금까지 사용해 온 기존의 핀먹스 코드의 특징을 살펴 보면, 우선 각 칩 제조사 별로 독자적인 핀먹스 코드를 가지고 있었으며, API 역시 자신의 칩 구조에 맞게 특화되어 있어서, 여러 칩들 간에 같은 기능을 다른 방식으로 구현한 형태였다. 또한 핀먹스는 SoC 레벨에서만 가능했으며, 디바이스 드라이버에서 사용할 수 있는 방법이 없었다.

반면에 본 절에서 소개할 새로운 핀먹스 서브 시스템(일명 pinctrl subsystem 이라고 함)은 아래와 같은 특징을 갖고 있다.

*1) 주로 Linus Walleij(Linaro/ST-Ericsson)에 의해 개발되고 관리되고 있음.*

*2) drivers/pinctrl 디렉토리에 별도의 코드를 구현함.*

*3) 핀의 목록, 기능, 제어 방법 등으로 이루어진 pinctrl 드라이버를 등록해 주는 API를 제공함.*

*4) 디바이스 드라이버에서 특정 핀 집합에 대해 먹싱을 요청(request pin muxing)할 수 있도록 하는 API를 제공함*
*5) GPIO 프레임워크와의 상호 작용 관련 기능을 제공함.*

그림 4-15는 pinctrl subsystem의 전체 구조를 그림으로 도해한 것으로, device tree와 GPIO/IRQ sub system 등과의 관계도 함께 표현해 보았다.

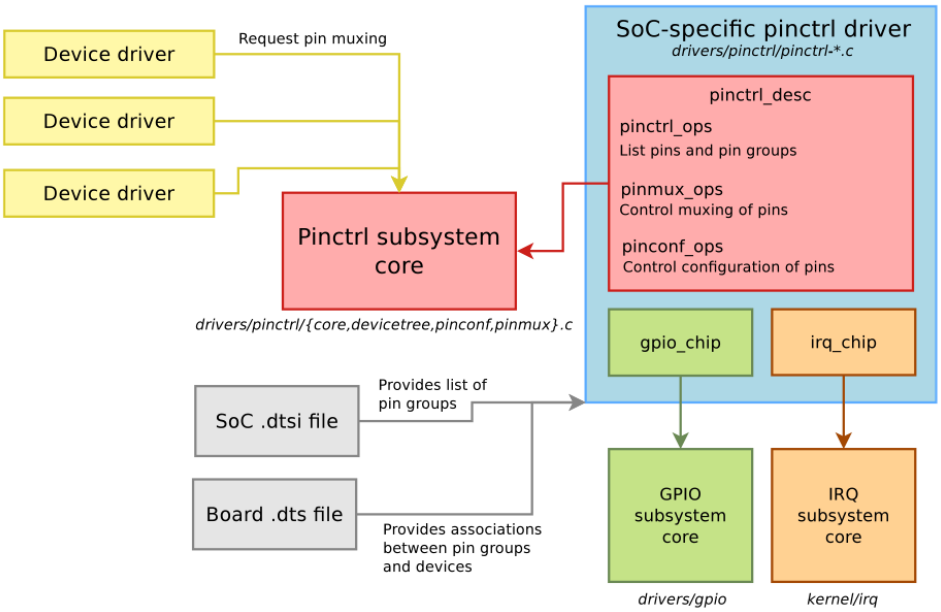


**그림 4-15 신규 Pin Muxing 서브시스템 구조 [출처 – 참고 문헌 1-6]**

Device tree를 사용하여 pinctrl subsystem을 표현하는 방법을 예로 들어 보기로 하자.

**SoC dtsi 파일 내에서 핀 그룹(pin group) 선언하기**

```
From arch/arm/boot/dts/imx28.dtsi
Declares the pinctrl device and various pin groups

pinctrl@80018000 {
    #address-cells = <1>;
    #size-cells = <0>;
    compatible = "fsl,imx28-pinctrl", "simple-bus";
    reg = <0x80018000 2000>;
    duart_pins_a: duart@0 {
        reg = <0>;
        fsl,pinmux-ids = <0x3102 0x3112>;
        fsl,drive-strength = <0>;
        fsl,voltage = <1>;
        fsl,pull-up = <0>;
```

```
        };
        duart_pins_b: duart@1 {
                reg = <1>;
                fsl,pinmux-ids = <0x3022 0x3032>;
                fsl,drive-strength = <0>;
                fsl,voltage = <1>;
                fsl,pull-up = <0>;
        };
        mmc0_8bit_pins_a: mmc0-8bit@0 {
                reg = <0>;
                fsl,pinmux-ids = <0x2000 0x2010 0x2020
                        0x2030 0x2040 0x2050 0x2060
                        0x2070 0x2080 0x2090 0x20a0>;
                fsl,drive-strength = <1>;
                fsl,voltage = <1>;
                fsl,pull-up = <1>;
        };
        mmc0_4bit_pins_a: mmc0-4bit@0 {
                reg = <0>;
                fsl,pinmux-ids = <0x2000 0x2010 0x2020
                        0x2030 0x2080 0x2090 0x20a0>;
                fsl,drive-strength = <1>;
                fsl,voltage = <1>;
                fsl,pull-up = <1>;
        };
        mmc0_cd_cfg: mmc0-cd-cfg {
                fsl,pinmux-ids = <0x2090>;
                fsl,pull-up = <0>;
        };
        mmc0_sck_cfg: mmc0-sck-cfg {
                fsl,pinmux-ids = <0x20a0>;
                fsl,drive-strength = <2>;
                fsl,pull-up = <0>;
        };
};
```

**보드 dts 파일 내에서 핀 그룹(pin group)과 장치를 결합시키기**

From arch/arm/boot/dts/cfa10036.dts

```
apb@80000000 {
    apbh@80000000 {
        ssp0: ssp@80010000 {
            compatible = "fsl,imx28-mmc";
            pinctrl-names = "default";
            pinctrl-0 = <&mmc0_4bit_pins_a
            &mmc0_cd_cfg &mmc0_sck_cfg>;
            bus-width = <4>;
            status = "okay";
        };
    };
    apbx@80040000 {
        duart: serial@80074000 {
            pinctrl-names = "default";
            pinctrl-0 = <&duart_pins_b>;
            status = "okay";
        };
    };
};
```

**디바이스 드라이버 코드에서 핀먹스(pin muxing) 요청하기**

From drivers/mmc/host/mxs-mmc.c

```
static int mxs_mmc_probe(struct platform_device *pdev)
{
    [...]
    pinctrl = devm_pinctrl_get_select_default(&pdev->dev);
    if (IS_ERR(pinctrl)) {
        ret = PTR_ERR(pinctrl);
        goto out_mmc_free;
    }
    [...]
}
```

# 5. Platform device 기반의 구(old) ARM 보드 초기화

보드나 SoC 관련 세부 사항을 구현하기 위해서는 C code를 사용하며, 다양한 주변 장치와 보드 및 SoC 환경을 기술하기 위해, 보드마다 비슷하지만 서로 다른 방식의 코딩이 필요하다. 그림 4-16은 ARM Architecture -> ARM SoC -> ARM Board에 이르는 계층 구조를 표현한 것으로, 맨 하단의 보드 코드는 매우 복잡하고 보드마다 서로 (중첩되지만)다른 코드로 구성되어 있다.

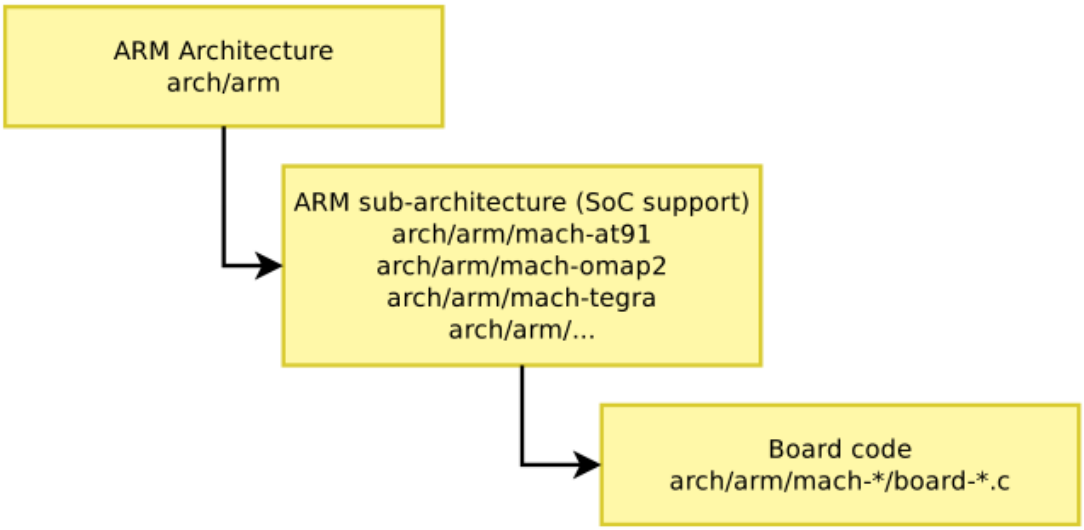


**그림 4-16 기존 보드 초기화 과정(1) [출처 – 참고 문헌 1-6]**

전통적인 방식을 따르는 보드 초기화 과정을 요약하면 다음과 같다.

*1) 부트로더는 커널을 로딩하면서 machine ID를 r1 register에 담아 전달한다.*

*2) 커널은 부팅의 마지막 단계에서 init_machine() 함수를 호출하여 보드 초기화 작업을 시작한다.*

*3) 보드 파일(arch/arm/mach-*/board-*.c 파일)은 SoC에 특화된 API를 사용하여 장치(platform device)와 보드에 특화된 내용을 등록(registration)한다.*

*4) SoC 파일(arch/arm/mach-*/devices*.c 혹은 *-devices.c 파일)은 base address/IRQ, pin muxing 정보 등을 알고 있으며, 이를 토대로 platform bus에 장치를 등록해 준다.*

*5) 각각의 디바이스 드라이버는 probe() 함수 내에서 platform bus를 통해 하드웨어와 직접 통신한다.*

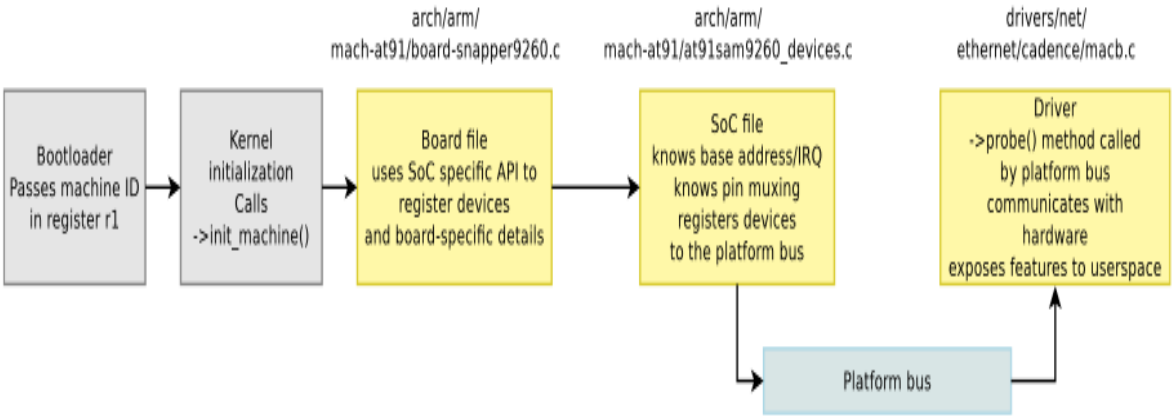


**그림 4-17 기존 보드 초기화 과정(2) [출처 – 참고 문헌 1-6]**

## 플랫폼 디바이스 & 드라이버

앞서 보드 초기화 과정에서 살펴 보았듯이, SoC와 드라이버 간에는 통신을 위해 플랫폼 버스(platform bus)를 경유하게 되는데, 이 속에 내포되어 있는 플랫폼 디바이스의 개념을 정리해 보면 다음과 같다.

**<플랫폼 디바이스>**

1) Bus는 확장성(enumeration), hot-plugging, unique identifier를 허용함에도 불구하고, Embedded system의 시스템의 경우, bus를 통해 device를 연결하지 않는 경우가 있음.

2) platform driver/platform device infrastructure를 사용하여 이를 해결할 수 있는데, platform device란, 별도의 bus를 거치지 않고, CPU에 직접 연결되는 장치를 일컬음.

플랫폼 디바이스는 동적으로 감지할 수가 없기 때문에 정적으로 정의해 주어야 하는데, 이를 위해서는 struct platform_device structure를 이용하여 아래와 같이 직접 선언해 주면 된다.

```
From arch/arm/mach-imx/mx1ads.c
static struct platform_device imx_uart1_device = {
    .name = "imx-uart",
    .id = 0,
    .num_resources = ARRAY_SIZE(imx_uart1_resources),
    .resource = imx_uart1_resources,
    .dev = {
        .platform_data = &uart_pdata,
    }
};
```

이렇게 선언한 플랫폼 디바이스는 devices 배열에 추가되며, 보드 초기화 과정을 진행하는 동안에 시스템에 추가된다.

```
static struct platform_device *devices[] __initdata = {
        &cs89x0_device,
        &imx_uart1_device,
        &imx_uart2_device,
};

static void __init mx1ads_init(void)
{
        [...]
        platform_add_devices(devices, ARRAY_SIZE(devices));
}
MACHINE_START(MX1ADS, "Freescale MX1ADS")
        [...]
        .init_machine = mx1ads_init,
MACHINE_END
```

각각의 장치는 대개 다른 장치와는 다른 고유의 하드웨어 리소스를 가지게 된다. 즉, I/O 레지스터 주소, DMA 채널, IRQ 라인 등이 서로 다르며, 이를 하드웨어 리소스라 부른다. 이러한 장치 별 고유의 정보는 struct resource를 사용하여 표현할 수 있으며, struct resource를 묶어 놓은 배열은 platform_device와 연결되게 된다. 아래 코드는 struct resource 배열을 사용하여 서로 다른 레지스터 주소, IRQ 등을 하나로 묶어 놓은 예이다. 한가지 알아두어야 할 사실은, 이러한 resource를 사용하는 장치는 서로 다른 resource 여러 개를 갖지만 수행하는 기능은 동일하다는 점이다.

```
static struct resource imx_uart1_resources[] = {
    [0] = {
        .start = 0x00206000,
        .end = 0x002060FF,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = (UART1_MINT_RX),
        .end = (UART1_MINT_RX),
        .flags = IORESOURCE_IRQ,
    },
};
```

platform_add_device() 함수를 사용하여 시스템에 platform_device를 추가하게 되면, platform driver의 probe() 함수가 호출되게 된다. 이 probe() 함수는 장치(하드웨어)를 초기화하는 역할을 하는데, 이때 적합한 서브시스템 프레임워크 내에 추가되게 된다(예: serial driver framework). 플랫폼 드라이버에서 I/O 리소스를 사용하는 방법은 다음과 같다.

```
res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
base = ioremap(res->start, PAGE_SIZE);
sport->rxirq = platform_get_irq(pdev, 0);
```

디바이스 공통으로 정해져 있는 리소스 정보 이외에도, 많은 디바이스 드라이버는 디바이스에 특화된 정보를 필요로 하게 된다. 이러한 정보는 struct device(struct platform_device는 이 것을 상속함)의 platform_data 필드를 사용하여 플랫폼 드라이버로 전달되게 된다. platform_data를 전달할 때에는 void * 포인터 형식으로 전달되기 때문에, 임의의 포맷의 데이터 전달이 가능하다. 아래에 platform_data를 통해 전달할 data를 예로 들어 보았다.

```
struct imxuart_platform_data {
    int (*init)(struct platform_device *pdev);
    void (*exit)(struct platform_device *pdev);
    unsigned int flags;
    void (*irda_enable)(int enable);
    unsigned int irda_inv_rx:1;
    unsigned int irda_inv_tx:1;
    unsigned short transceiver_delay;
};

static struct imxuart_platform_data uart1_pdata = {
    .flags = IMXUART_HAVE_RTSCTS,
};
```

아래 코드를 보면, uart_pdata structure는 platform_device와 연결되어 있음을 알 수 있다. 또한, 플랫폼 드라이버는 probe() 함수에서 platform data를 사용함을 알 수 있다.

```
struct platform_device mx1ads_uart1 = {
        .name = "imx-uart",
        .dev {
                .platform_data = &uart1_pdata,
        },
        .resource = imx_uart1_resources,
```

```
        [...]
};

static int serial_imx_probe(struct platform_device *pdev)
{
        struct imxuart_platform_data *pdata;
        pdata = pdev->dev.platform_data;
        if (pdata && (pdata->flags & IMXUART_HAVE_RTSCTS))
                sport->have_rtscts = 1;
        [...]
}
```

각각의 서브 시스템 프레임워크는 디바이스 드라이버가 이 프레임워크에 소속된 장치로 인식되도록 하기 위해서 등록해야 하는 structure를 정의해서 사용한다. 예를 들자면, serial port의 경우 uart_port structure, network devices의 경우 netdev structure, framebuffers의 경우는 fb_info structure 등을 생각해 볼 수 있다. 이러한 structure 이외에도, 드라이버는 자신의 디바이스를 위한 추가 정보를 저장하고 있어야 하는데, 이는 대개, 프레임워크 structure를 세분화(subclassing)해서 표현해 주거나, 적합한 프레임워크 structure에의 포인터를 저장하는 방식을 활용한다. 아래에 두 가지 경우에 해당하는 예를 들어 각각 들어 보았다.

```
[1] i.MX serial driver: imx_port is a subclass of uart_port
struct imx_port {
        struct uart_port port;
        struct timer_list timer;
        unsigned int old_status;
        int txirq, rxirq, rtsirq;
        unsigned int have_rtscts:1;
        [...]
};

[2] rtl8150 network driver: rtl8150 has a reference to net_device
struct rtl8150 {
        unsigned long flags;
        struct usb_device *udev;
        struct tasklet_struct tl;
        struct net_device *netdev;
        [...]
};
```

서브시스템 프레임워크는 struct device * 포인터를 갖고 있는데, 드라이버는 이것을 사용하여 struct device를 가리킬 수 있어야 한다. 이는 논리 장치(예: network interface 장치)와 물리 장치(예: USB network adaptor)와의 연결 고리이다. device structure 내에는 void * 포인터가 포함되어 있어서, 디바이스 드라이버가 이를 자유롭게 사용(임의의 format을 가진 data 전달 가능)할 수 있는 구조이다. 프레임워크로부터 플랫폼 디바이스를 역 참조하는 것이 가능하며, 또한, platform_device structure로부터 논리 장치를 표현하는 structure를 찾는 것도 가능하다.

아래에 서브시스템 프레임워크(예: uart_port)와 플랫폼 디바이스를 연결하는 예를 들어 보았다. 또한 그림 4-18에 이들의 관계를 그림으로 표현해 보았다.

```
static int serial_imx_probe(struct platform_device *pdev)
{
        struct imx_port *sport;
        [...]
        /* setup the link between uart_port and the struct device inside the platform_device */
        sport->port.dev = &pdev->dev;
        [...]
        /* setup the link between the struct device inside the platform device to the imx_port
        structure */
        platform_set_drvdata(pdev, &sport->port);
        [...]
        uart_add_one_port(&imx_reg, &sport->port);
}

static int serial_imx_remove(struct platform_device *pdev)
{
        /* retrieve the imx_port from the platform_device */
        struct imx_port *sport = platform_get_drvdata(pdev);
        [...]
        uart_remove_one_port(&imx_reg, &sport->port);
        [...]
}
```

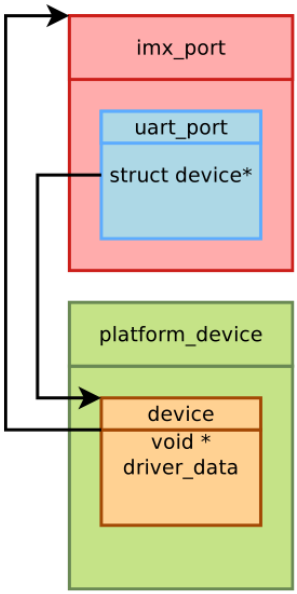


**그림 4-18 서브시스템 프레임워크와 Platform Device와의 관계 [출처 – 참고 문헌 1-6]**

지금까지 설명한 플랫폼 디바이스를 기반으로 보드 초기화하는 예를 소개해 보고자 한다.

**<Board File 예>**

```
static struct macb_platform_data snapper9260_macb_data = {
	.phy_irq_pin = -EINVAL,
	.is_rmii = 1,
};

static struct i2c_board_info __initdata snapper9260_i2c_devices[] = {
	{ I2C_BOARD_INFO("max7312", 0x28),
	.platform_data = &snapper9260_io_expander_data, },
	{ I2C_BOARD_INFO("tlv320aic23", 0x1a), },
};

static void __init snapper9260_board_init(void)
{
	at91_add_device_i2c(snapper9260_i2c_devices,
	ARRAY_SIZE(snapper9260_i2c_devices));
	at91_register_uart(0, 0, 0);
	at91_register_uart(AT91SAM9260_ID_US0, 1,
	ATMEL_UART_CTS | ATMEL_UART_RTS);
	at91_add_device_eth(&snapper9260_macb_data);
	[...]
```

```
}

MACHINE_START(SNAPPER_9260, "Bluewater Systems Snapper 9260/9G20 module")
    [...]
    .init_machine = snapper9260_board_init,
MACHINE_END

The board file registers devices (from arch/arm/mach-at91/board-snapper9260.c)
```

**<Device Definition(SoC 파일) 예>**

```
static struct macb_platform_data eth_data;
static struct resource eth_resources[] = {
    [0] = {
        .start = AT91SAM9260_BASE_EMAC,
        .end = AT91SAM9260_BASE_EMAC + SZ_16K - 1,
        .flags = IORESOURCE_MEM,
    },
    [1] = {
        .start = AT91SAM9260_ID_EMAC,
        .end = AT91SAM9260_ID_EMAC,
        .flags = IORESOURCE_IRQ,
    },
};

static struct platform_device at91sam9260_eth_device = {
    .name = "macb",
    .id = -1,
    .dev = {
        .dma_mask = &eth_dmamask,
        .coherent_dma_mask = DMA_BIT_MASK(32),
        .platform_data = &eth_data,
    },
    .resource = eth_resources,
    .num_resources = ARRAY_SIZE(eth_resources),
};
```

**<Device Registration 예>**

```
void __init at91_add_device_eth(struct macb_platform_data *data)
```

```
{
   [...]
   if (gpio_is_valid(data->phy_irq_pin)) {
      at91_set_gpio_input(data->phy_irq_pin, 0);
      at91_set_deglitch(data->phy_irq_pin, 1);
   }
   /* Pins used for MII and RMII */
   at91_set_A_periph(AT91_PIN_PA19, 0); /* ETXCK_EREFCK */
   at91_set_A_periph(AT91_PIN_PA17, 0); /* ERXDV */
   at91_set_A_periph(AT91_PIN_PA14, 0); /* ERX0 */
   at91_set_A_periph(AT91_PIN_PA15, 0); /* ERX1 */
   at91_set_A_periph(AT91_PIN_PA18, 0); /* ERXER */
   at91_set_A_periph(AT91_PIN_PA16, 0); /* ETXEN */
   at91_set_A_periph(AT91_PIN_PA12, 0); /* ETX0 */
   at91_set_A_periph(AT91_PIN_PA13, 0); /* ETX1 */
   at91_set_A_periph(AT91_PIN_PA21, 0); /* EMDIO */
   at91_set_A_periph(AT91_PIN_PA20, 0); /* EMDC */
   if (!data->is_rmii) {
      [...]
   }
   eth_data = *data;
   platform_device_register(&at91sam9260_eth_device);
}
```

**<device driver code 예>**

```
static int __init macb_probe(struct platform_device *pdev)
{
      [...]
}
static int __exit macb_remove(struct platform_device *pdev)
{
      [...]
}
static struct platform_driver macb_driver = {
      .remove = __exit_p(macb_remove),
      .driver = {
            .name = "macb",
            .owner = THIS_MODULE,
```

```
	},
};
static int __init macb_init(void)
{
	return platform_driver_probe(&macb_driver, macb_probe);
}
static void __exit macb_exit(void)
{
	platform_driver_unregister(&macb_driver);
}
module_init(macb_init);
module_exit(macb_exit);
```

# 6. Device Tree 기반의 신(new) ARM 보드 초기화

Device Tree를 기반으로 ARM 보드를 초기화하는 과정을 정리해 보면 다음과 같다.

*1) .dtsi(SoC 용 dts 파일)와 .dts(보드용 dts 파일) 파일을 컴파일하여 .dtb(device tree blob) 파일을 생성한다.*

*2) 부트로더는 .dtb 파일을 메모리(DRAM)으로 로드(적재)한 후, 이 주소를 r2 register에 담아 커널로 전달한다.*

*3) .dtb 파일은 board-*.c 파일 내의 대부분의 코드를 대체하는 역할을 하며, .dtb 파일을 사용할 경우, 이제는 더 이상 platform_device를 등록(registration)하는 과정이 필요 없게 된다. 또한 각각의 보드 별로 설정을 필요로 했던 Kconfig option도 더 이상 필요 없어지게 된다.*

*4) .dt_compat으로 명시되는 보드 파일(device tree)은 of_platform_populate() 함수를 호출해 주어야 각각의 장치로 등록이 된다(node -> device).*

*5) 이상의 과정에서 보드 초기화가 마무리되었으므로, 각각의 장치 드라이버는 probe() 함수 내에서 자신(node)에 맞는 정보를 얻어(of_ 시작하는 함수 사용)와 초기화 작업에 사용한다.*

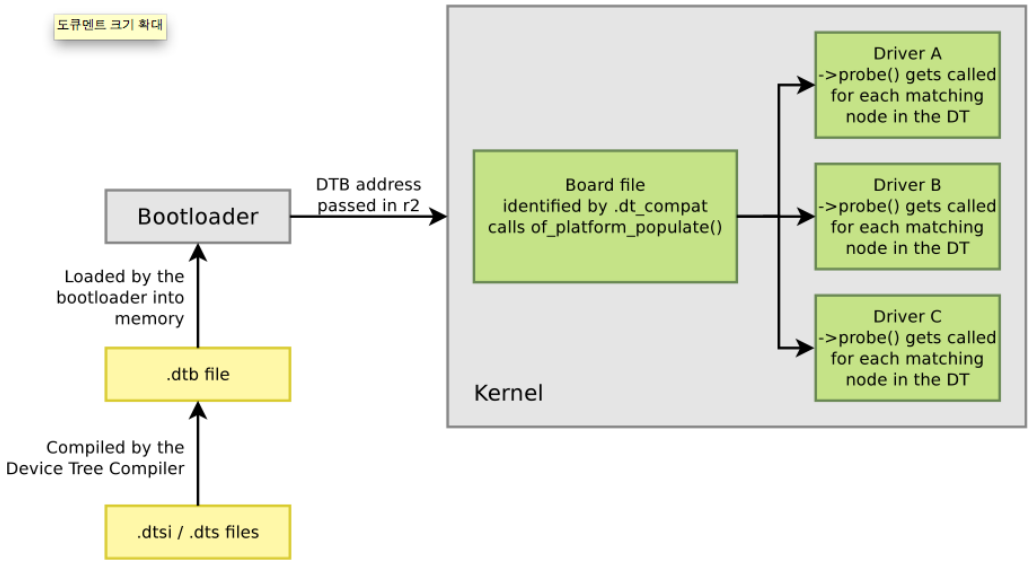


**그림 4-19 디바이스 트리를 이용한 보드 초기화 과정 [출처 – 참고 문헌 1-6]**

그림 4-20은 기존 부팅 방식과 dtb 파일이 추가된 상태에서의 부팅 방식을 그림으로 표현한 것이다. 앞서도 이미 설명했다시피, DTB(Device Tree Blob)를 지원하는 부트로더를 사용 중인 경우라면, 부트로더가 알아서 dtb 파일을 메모리에 적재한 후, r2 register를 통해 이의 주소를 커널에 넘겨주는 형태로 부팅이 진행되지만, 그렇지 못한 경우에는 커널 이미지(uImage) 끝에 dtb image를 함께 붙이는 방식으로 부팅이 진행되어야 한다. 이 경우 부트로더는 dtb가 붙어 있는 커널을 구동(메모리 적재 후, 시작 번지로 jump) 시키게 되고, 이후 커널이 알아서 dtb 정보를 가져오게 된다.

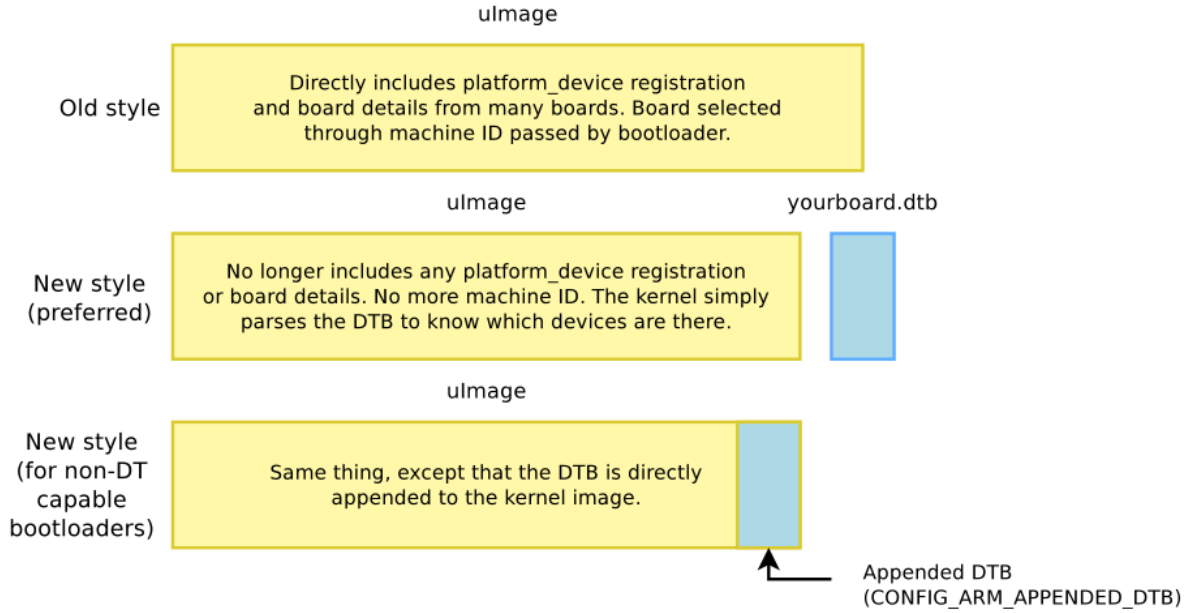


**그림 4-20 디바이스 트리를 이용한 부팅 [출처 – 참고 문헌 1-6]**

| 참고 사항: dtc(Device Tree Compiler)는 linux kernel tree내에 있으며, 아래 예 처럼, dts와 dtb 파 |
|---|

일을 상호 전환해 주는 역할을 한다.

1) dts 파일로 부터 dtb 파일 생성하기

***$ scripts/dtc/dtc -I dts -O dtb -o /path/to/my-tree.dtb /path/to/my-tree.dts***

2) dtb 파일로부터 dts 파일 생성하기

***$ scripts/dtc/dtc -I dtb -O dts -o /path/to/fromdtb.dts /path/to/found_this.dtb***

DTS(Device Tree Source)는 SoC 용과 보드 용으로 크게 구분이 가능한데, SoC 용은 특별히 .dtsi 확장자를 사용하고, 보드용 파일은 .dts 확장자를 사용한다. 이 둘의 관계는 아래 그림 4-21에서 볼 수 있듯이 보드용 dts는 SoC 용 dtsi를 상속받는 구조로 이해하면 될 것이다. 물론 dtsi 역시 다른 dtsi를 상속 받을 수도 있다.

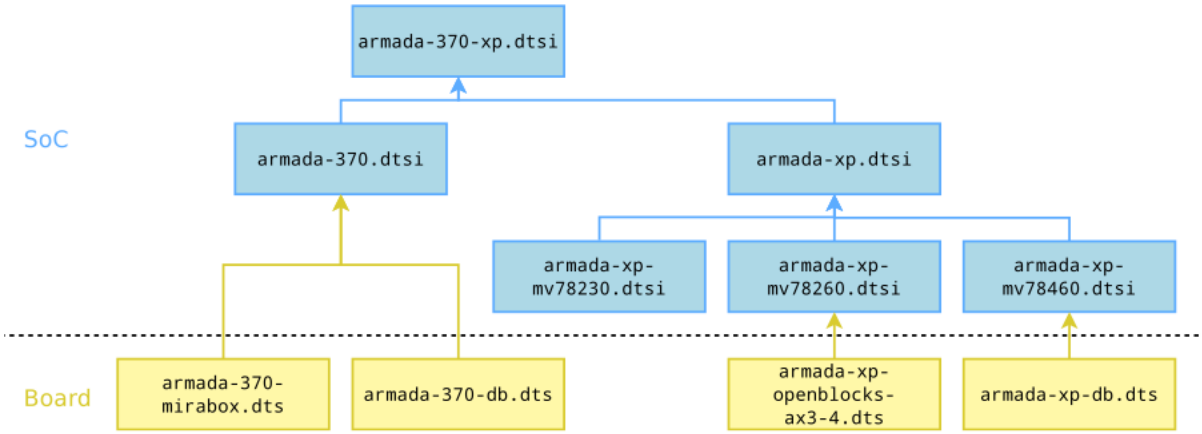


**그림 4-21 디바이스 트리 상속(SoC dtsi -> Board dts) [출처 – 참고 문헌 1-6]**

아래 예제는 위의 디바이스 트리 상속 관계를 기반으로 실제 코딩 시 필요한 4단계의 과정을 정리해 본 것이다.

*1 단계) SoC 용 dtsi 파일 정의*

*2 단계) SoC 용 dtsi 파일을 상속(include 문으로 dtsi 파일을 추가해야 함)한 보드용 dts 파일 정의*

*3 단계) 보드 초기화 파일(board-*.c 파일) 추가 – dts를 고려하여 작성함. 기존의 platform device 기반 코드에 비해 매우 간단해 짐.*

*4 단계) device driver 작성 – dts를 고려하여 작성함(특히 probe() 함수)*

## <tegra20.dtsi device tree 예 – SoC dtsi file>

```
/include/ "skeleton.dtsi"
/ {
```

```
    compatible = "nvidia,tegra20";
    interrupt-parent = <&intc>;
    intc: interrupt-controller {
        compatible = "arm,cortex-a9-gic";
        reg = <0x50041000 0x1000
        0x50040100 0x0100>;
        interrupt-controller;
        #interrupt-cells = <3>;
    };

    serial@70006000 {
    compatible = "nvidia,tegra20-uart";
        reg = <0x70006000 0x40>;
        reg-shift = <2>;
        interrupts = <0 36 0x04>;
        status = "disable";
    };

    serial@70006040 {
        compatible = "nvidia,tegra20-uart";
        reg = <0x70006040 0x40>;
        reg-shift = <2>;
        interrupts = <0 37 0x04>;
        status = "disable";
    };

    i2c@7000c000 {
        compatible = "nvidia,tegra20-i2c";
        reg = <0x7000c000 0x100>;
        interrupts = <0 38 0x04>;
        #address-cells = <1>;
        #size-cells = <0>;
        status = "disable";
    };

    i2c@7000c400 {
        compatible = "nvidia,tegra20-i2c";
        reg = <0x7000c400 0x100>;
        interrupts = <0 84 0x04>;
```

```
        #address-cells = <1>;
        #size-cells = <0>;
        status = "disable";
    };

    usb@c5004000 {
        compatible = "nvidia,tegra20-ehci","usb-ehci";
        reg = <0xc5004000 0x4000>;
        interrupts = <0 21 0x04>;
        phy_type = "ulpi";
        status = "disable";
    };

    [...]
};
```

**<tegra-harmony.dts device tree 예 – board dts file>**

```
/dts-v1/;
/include/ "tegra20.dtsi"
/ {
        model = "NVIDIA Tegra2 Harmony evaluation board";
        compatible = "nvidia,harmony", "nvidia,tegra20";
        memory {
                reg = <0x00000000 0x40000000>;
        };

        serial@70006300 {
                status = "okay";
                clock-frequency = <216000000>;
        };

        i2c@7000c400 {
                status = "okay";
                clock-frequency = <400000>;
        };

        i2c@7000c000 {
                status = "okay";
```

```
            clock-frequency = <400000>;
            wm8903: wm8903@1a {
                compatible = "wlf,wm8903";
                reg = <0x1a>;
                interrupt-parent = <&gpio>;
                interrupts = <187 0x04>;
                gpio-controller;
                #gpio-cells = <2>;
                [...]
            };
        };

        usb@c5004000 {
            status = "okay";
            nvidia,phy-reset-gpio = <&gpio 169 0>;
        };

        [...]
};
```

**<arch/arm/mach-tegra/board-dt-tegra20.c 보드 파일 예>**

```
static void __init tegra_dt_init(void)
{
        [...]
        of_platform_populate(NULL, tegra_dt_match_table,
        tegra20_auxdata_lookup, NULL);
}

static const char *tegra20_dt_board_compat[] = {
        "nvidia,tegra20",
        NULL
};

DT_MACHINE_START(TEGRA_DT, "nVidia Tegra20 (Flattened Device Tree)")
        .init_machine = tegra_dt_init,
        .dt_compat = tegra20_dt_board_compat,
        [...]
MACHINE_END
```

**<i2c-tegra.c 드라이버 코드 예>**

```
static const struct of_device_id tegra_i2c_of_match[] __devinitconst = {
	{ .compatible = "nvidia,tegra20-i2c", },
	{ .compatible = "nvidia,tegra20-i2c-dvc", },
	{},
};

MODULE_DEVICE_TABLE(of, tegra_i2c_of_match);

static struct platform_driver tegra_i2c_driver = {
	.probe = tegra_i2c_probe,
	.remove = __devexit_p(tegra_i2c_remove),
	.driver = {
		.name = "tegra-i2c",
		.owner = THIS_MODULE,
		.of_match_table = tegra_i2c_of_match,
	},
};

static int __init tegra_i2c_init_driver(void)
{
	return platform_driver_register(&tegra_i2c_driver);
}
static void __exit tegra_i2c_exit_driver(void)
{
	platform_driver_unregister(&tegra_i2c_driver);
}

subsys_initcall(tegra_i2c_init_driver);
module_exit(tegra_i2c_exit_driver);
```

# 7. ARM Porting 단계 요약

새로운 ARM 플랫폼을 포팅하는 과정을 크게 3단계로 나누어 정리해 보았다.

**<1단계 – 기본 부팅 가능 상태 만들기>**

1) Device Tree를 이용하여 SoC와 보드를 정의(기술)한다.
2) arch/arm/mach-<foo>/ 아래에 보드 파일을 생성한다.
3) drivers/clocksource 아래의 timer driver를 device tree 기반으로 포팅한다.
4) drivers/irqchip 아래의 IRQ controller driver를 device tree 기반으로 포팅한다.
5) debug message 출력이 가능하도록 arch/arm/include/debug 및 시리얼 드라이버 (drivers/tty/serial)를 포팅한다.

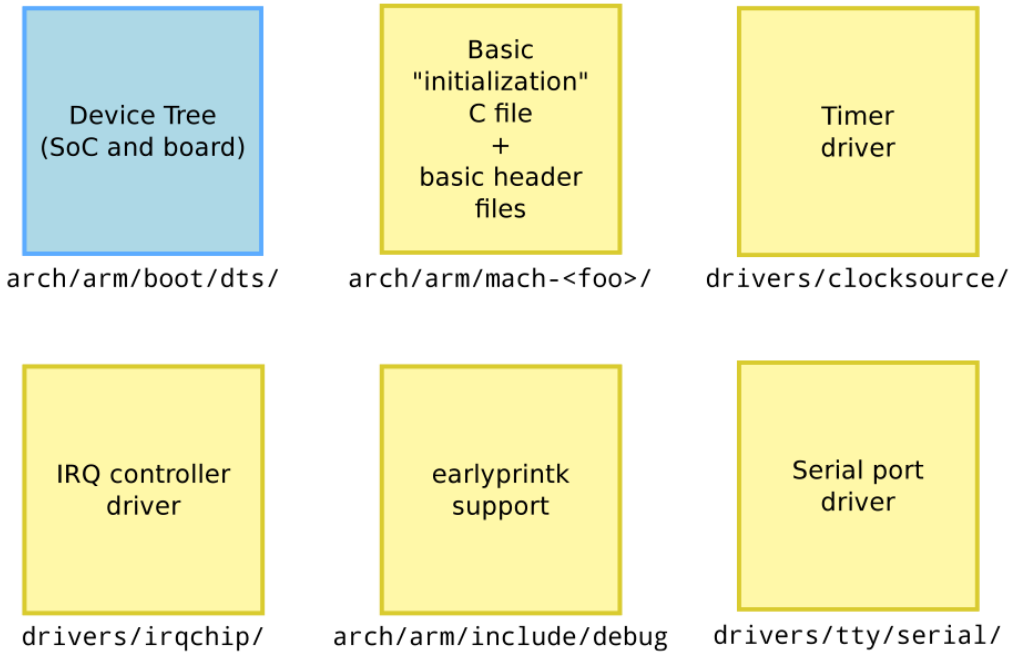


**그림 4-22 ARM Porting 1단계 – 디바이스 트리 정의, 보드 초기화, 기본 드라이버 포팅 [출처 – 참고 문헌 1-6]**

**<2단계 – Clock, Pin control 부분 적용 단계>**

6) Clock code를 CCF에 맞게 수정한다.
7) Pin muxing code를 pinctrl subsystem에 맞게 수정한다.
8) GPIO를 새로운 code에 맞게 수정한다.

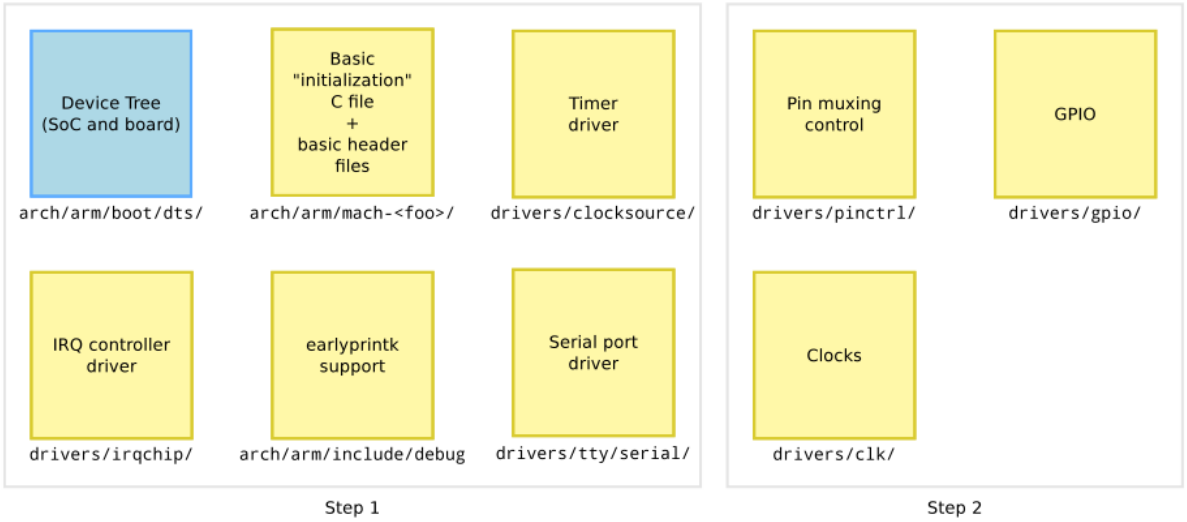


**그림 4-23 ARM Porting 2단계 – CCF, Pin muxing, GPIO [출처 – 참고 문헌 1-6]**

**<3단계 – 나머지 드라이버 수정하기>**

9) 마지막으로 각종 디바이스 드라이버를 device tree 기반으로 전환한다.


**그림 4-24 ARM Porting 3단계 – 나머지 디바이스 드라이버 포팅 [출처 – 참고 문헌 1-6]**

본 서의 내용 중 일부는 Free Electrons의 문서[참고문헌 1-6]를 참고하였음을 밝힌다. 특히 그림 중 일부는 그대로 복사하여 사용하였다.

---



---

# References


*[1] Linux kernel: consolidation in the ARM architecture support, Thomas Petazzoni, Free Electrons, thomas.petazzoni@free-electrons.com*

*[2] ARM support in the Linux kernel, Thomas Petazzoni, Free Electrons, thomas.petazzoni@free-electrons.com*

*[3] Your new ARM SoC Linux support check-list! Thomas Petazzoni, Free Electrons, thomas.petazzoni@free-electrons.com*

*[4] Common clock framework: how to use it, Gregory CLEMENT, Free Electrons, gregory.clement@free-electrons.com*

*[5] Driver development Kernel architecture for device drivers, Free Electrons*

*[6] Linux Kernel and Driver Development Training, Gregory Clement, Michael Opdenacker, Maxime Ripard, S_ebastien Jan, Thomas Petazzoni, Free Electrons*

*[7] Standard for Embedded Power Architecture™ Platform Requirements (ePAPR) Version 1.1 – 08 April 2011, Power.org*

*[8] http://www.devicetree.org/Device_Tree_Usage*

*[9] Embedded Android, Karim Yaghmour, O'reilly, 2013*

*[10] Jelly Bean BSP Porting Guide, Chunghan Yi, www.kandroid.org*