Android Kernel Hacks

안드로이드를 위한 리눅스 커널 핵스

2013.7.16 ~ 2013.9.27

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

목차

CHAPTER 1

안드로이드 소개

본 장에서는 안드로이드의 개념을 소개한 후, 안드로이드의 부팅 순서를 분석해 봄으로써, 안드로이드의 동작 방식을 이해해 보고자 한다. 또한 안드로이드 빌드 시스템을 파악해 보고, 빌드 결과로 얻어진 boot.img를 역 분석하는 과정을 소개하고자 한다.

- 부트로더부터 안드로이드 애플리케이션까지의 구동 순서 소개
  - Primary 및 Secondary bootloader
  - Application bootloader
  - Kernel과 Root File System
  - Init process와 init.rc
  - 바인더와 서비스 매니저
  - Zygote와 Dalvik VM
  - 시스템 서비스(mediaserver, system_server, phone service)
  - Activity manager와 android applications
- 안드로이드 빌드 시스템 분석
  - 다운로드 & 빌드, 빌드 시스템 분석
  - 빌드 결과 분석(Boot.img 해부)
- 본 서의 대상 및 범위, 참조 코드 소개

1. 안드로이드 개요

Google은 안드로이드(Android)를 설계하면서 기존 제품들과는 한 차원 다른 아키텍처를 선 보였다. 먼저 안드로이드의 인터페이스에 해당하는 응응 프로그래밍 언어로써는 이미 전 세계적으로 가장 넓은 사용자 층을 확보하고 있는 Java를 선택했으며, 내부 핵심 운영체제로는 역시 전세계적으로 가장 많은 인기를 누리고 있는 오픈 소스 운영체제인 리눅스(Linux)를 선택하였다. 이 화려한 두 조합이 정상적으로 결합하기 위해서는 생각보다 많은 노력이 필요했는데, 이는 Dalvik VM(Virtual Machine)의 개발, Java 프레임워크의 구현, Java와 C/C++(Native라고도 함) 간의 인터페이스인 JNI, Native 프레임워크의 구현, C/C++ 기반의 여러 오픈 소스 프로젝트의 최적화 작업 등이 이를 잘 말해주고 있다.

안드로이드의 구조는 그림 1-1에서 볼 수 있듯이 크게 5개의 계층으로 나뉘어져 있는데, 맨 먼저 파란색으로 표시된 최상위 계층은 (1) Java 기반의 응용 프로그램이 차지하고 있으며, 그 아래로는 역시 파란색으로 표시된 (2) Java 기반의 응용 프레임워크(application framework)가 위치하고 있다. 이 응용 프레임워크는 글자 그대로 Java 기반의 응용 프로그램을 구현하기 위해 필요한 내부 루틴으로 이루어져 있는데, 이는 다음 장에서 설명하게 될 시스템 서버(system_server)를 구성하는 내용이기도 하다. 다음으로 노란색으로 표시되어 있는 (3) 안드로이드 런타임은 Dalvik VM과 Core 라이브러리(android.*)로 이루어져 있다. Dalvik VM은 기존의 Java VM과 마찬가지로 Java로 구현된 코드를 번역(컴파일)하는 역할을 담당하게 되며, 안드로이드 Java 라이브러리(jar 파일)로 구성된 Core 라이브러리는 Apache Harmony 프로젝트에 기반하고 있다. 다음으로 녹색으로 표시되어 있는 (4) Native 라이브러리 계층은 C/C++로 구현되어 있으며, Java 응용 프레임워크에 대비되는 C/C++ 응용 프레임워크(surfaceflinger, mediaserver 등이 주축임)는 물론이고, 각종 오픈 소스를 안드로이드에 맞게 최적화 시킨 라이브러리 들로 구성되어 있다. 마지막으로 빨간색으로 표시되어 있는 최 하단은 (5) 리눅스 커널(Linux kernel)이 차지하고 있으며, 디스플레이(Display), 카메라(Camera), 블루투스(Bluetooth), 공유 메모리(ashmem), 바인더(Binder), USB, 키 패드(GPIO key input, touchscreen), 와이파이(Wi-Fi), 오디오 사운드, 파워 관리(Power Management) 드라이버 등으로 구성되어 있다.

<안드로이드 소프트웨어 스택의 구조>

1) Java 기반의 응용 프로그램 계층(파란색)

2) Java 기반의 응용 프레임워크(application framework) 계층(파란색)

3) 안드로이드 런타임(노란색)

- Core Library(파란색) + Dalvik Virtual Machine(노란색)

4) Native 라이브러리 계층(녹색)

- C/C++로 구성된 각종 라이브러리(각종 오픈 소스를 최적화함)

5) 리눅스 커널(빨간색)

- 본 서의 대상이 되는 부분

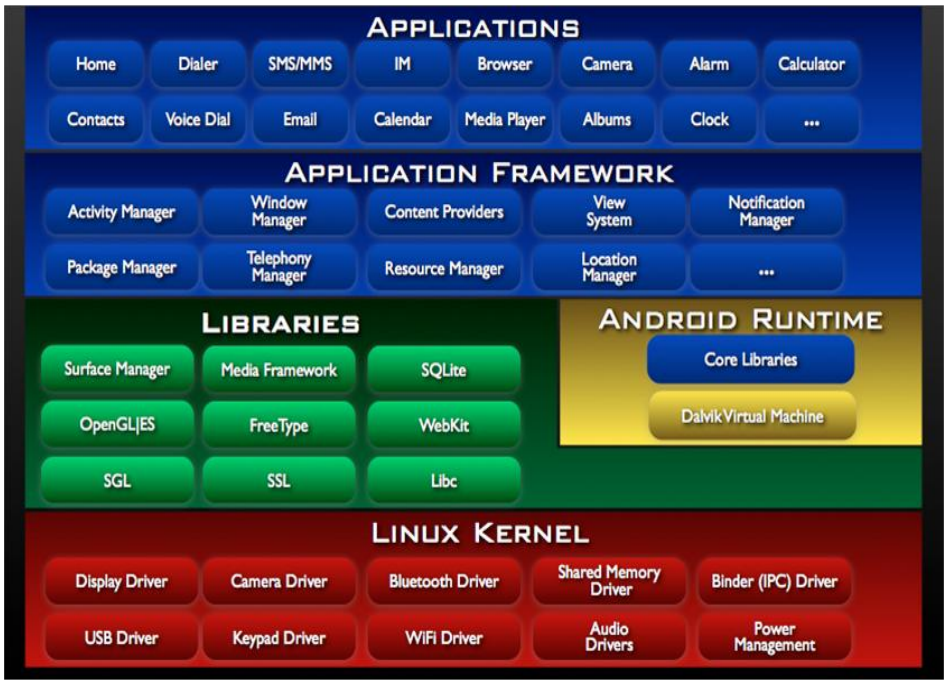

그림 1-1 Google 안드로이드 아키텍처 [출처 – 참고문헌 1]

Google이 안드로이드를 설계하면서 고민한 부분을 좀 더 구체적으로 정리해 보면 아래와 같다.

---

1) Java 기반의 새로운 UI 프레임워크 및 SDK 제공

- Java 개발자를 끌어 들임.
- C++에 비해 보다 쉬운 환경.
- 기존 Oracle(구 Sun) Java의 full 기능과는 차이가 있으며, 새로운 컴포넌트(component) 도입으로 새로 배워야 하는 문제 있음.

2) 새로운 Java VM(Dalvik VM) 개발

- Oracle((구 Sun)과의 Java 라이센스 문제를 타파하기 위해 개발함.
- 기존 방식에 비해 속도 개선함.

3) JNI(Java Native Interface) : Java <-> C/C++ interface

- 기존 C++ 기반의 UI 대비 프로그래밍의 복잡도 증가함.
- 단순히 JNI의 문제가 아니라, Java 프레임워크와 C/C++ 프레임워크와의 관점임.

4) 새로운 IPC 도입(binder)

- 매우 훌륭한 기법이나, 구현의 복잡도 증가(+ JNI 시 더욱더 복잡해짐).
- 또한 기존 C/C++ applications/libraries가 보유한 IPC(예: socket)와 연계하면서 IPC 기법이 매우 복잡해짐.
- 어쩔 수 없는 측면도 있음 – 이보다 더 잘 만들 수 있을까 ?

5) GNU libc -> Bionic libc

- 라이센스 문제를 타파하기 위해 작은 libc를 만듦
- BSD license, high-performance, compact size 제공.
- 문제는 C/C++ 기반의 응용 프로그램 및 라이브러리 포팅이 기존 Embedded Linux 대비 상대적으로 어려워짐.

6) C/C++ 기반의 오픈 소스 최적화

- 스마트 폰 개발에 필요한 대부분의 C/C++ 응용 프로그램 및 라이브러리가 이미 포팅되어 있음.
- 4)에서도 언급했듯이 기존 응용 프로그램 및 라이브러리가 이미 보유하고 있는 IPC를 최대한 유지하면서 최적화 작업 진행함(예: d-bus를 사용하는 Bluetooth).

7) init process의 개선

- init script -> init.rc/init.{hw}.rc로 개선.
- 매우 체계적으로 변모했으나, 새로운 문법이 적용되어 있어 역시 새로 배워야 하는 부담이 있음.

8) Linux kernel의 개선

- Linux kernel 의 메인 트리(main tree)와는 다른 별도의 커널 트리(kernel tree)를 가져감으로써 복잡도 증가
- 그러나 최근에 다시 메인 트리에 통합되었음(kernel 3.x 부터)
- 바인더(binder), 애쉬멤(ashmem), 웨이크락(wakelock), 로우메모리킬러(low memory killer), 램콘솔(ram console), 알람(alarm), USB 가젯(usb gadget), ION 메모리 등이 신규로 추가됨.

9) Bootloader 수정

- 새로운 Android boot 파티션 정립.
- Recovery 개념 도입.
- Fastboot 기능 추가함.

10) 새로운 빌드(build) 시스템 도입

- Makefile -> Android.mk 방식으로 바뀜.
- 복잡하지는 않으나, 그래도 새로 배워야 하는 부분이 존재함

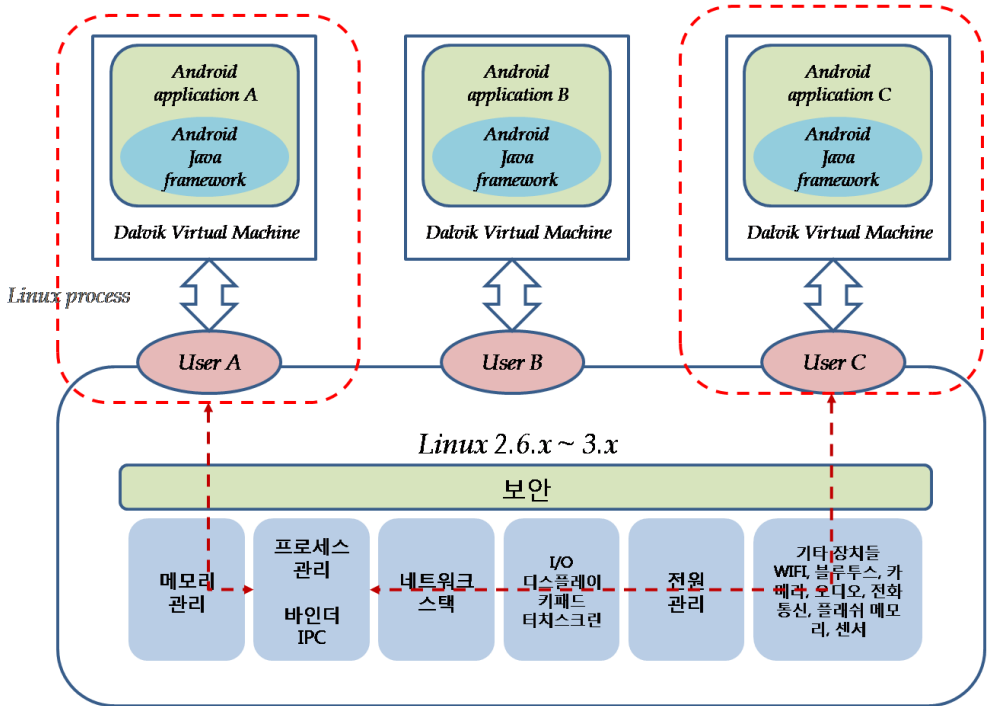

그림 1-2 단순화한 안드로이드 아키텍처

그림 1-2는 앞서 설명한 복잡한 안드로이드 아키텍처를 사용자 프로세스와 커널의 관점에서 간략화시켜 재구성해 본 것이다. 그림에서 볼 수 있듯이 응용 프로그램 개발자가 Java로 프로그램을 작성하면, C/C++로 구현된 Dalvik VM(라이브러리 형태로 존재함)이 이를 컴파일하게 되며, 여기에 C/C++로 구현된 메인 루틴을 추가하게 되면 기존의 C/C++로 작성된 프로그램과 동일한 형태의 프로세스가 만들어지게 된다. 따라서, Dalvik VM 등 복잡한 부분을 잊어 버린다면, 단순히 Linux Kernel 위에 C/C++로 작성된 여러 가지 응용 프로그램이 동작하고 있는 기존의 구조와 크게 다를 바가 없다고 볼 수 있다.

앞서 설명한 내용을 토대로, Java로 만들어진 안드로이드 응용 프로그램의 구조를 그림으로 그려보면 다음과 같다.

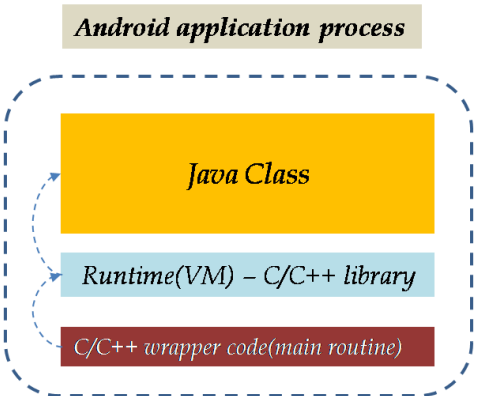

그림 1-3 Java로 만들어진 안드로이드 응용 프로그램의 구조

안드로이드 응용 프로그램 구조 관련하여 좀 더 살펴 보면, Java와 C/C++ 간의 상호 함수 혹은

메서드 호출을 원할하게 해주는 Java Native Interface(줄여서 JNI라고 부름)를 빼 놓고서는 설명이 안될 것이다. 응용 프로그램의 구조가 운이 좋아 그림 1-3과 같이 단순한 형태로 되어 있다면 별 문제가 없겠지만, 대개의 경우는 그 보다는 훨씬 복잡하여 Java 코드에서 C/C++ 라이브러리를 호출한다거나, 반대로 C/C++ 코드 내에서 Java 메서드를 호출해야 하는 상황이 발생할 수 있다. JNI는 이러한 상황을 위해 마련된 인터페이스로 Dalvik VM에서 이를 적절히 처리해 주는 것으로 이해하면 된다. JNI 관련 보다 자세한 사항은 본 서의 범위를 넘어서는 바, 추가 설명은 자제하도록 하겠다.

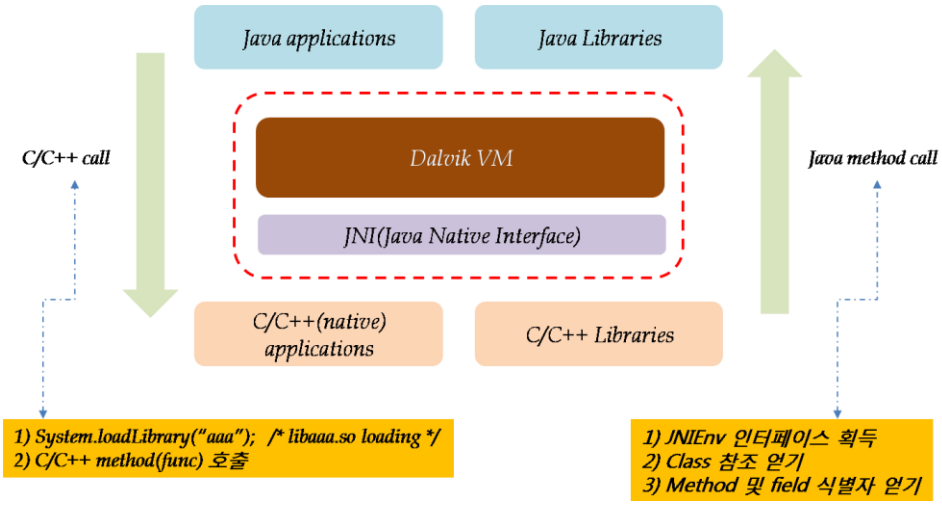

그림 1-4 Java Native Interface

솔직히 이 글을 작성하는 지금은 아주 사소한 내용처럼 느껴지지만, 필자가 처음 안드로이드를 접했을 당시에는 Java로 작성한 응용 프로그램(그림 1-3)이 과연 리눅스 상에서 어떤 식으로 표현(?)될 것인지가 매우 궁금하였던 부분이다. 이 밖에도 안드로이드로 작업을 하다 보면, (다분히 필자의 주관적인 생각이기는 하나) 몇 가지 재미 있는 주제와 만나게 되는데, 이를 정리해 보는 것으로 안드로이드에 대한 간략한 소개를 마칠까 한다.

<왜 개발자들은 안드로이드에 열광하는가 ?>

1) Davik VM과 연동하여 만드는 Java application

2) Zygote의 개념

3) Binder의 동작 원리

4) init.rc와 거대(?)해진 init process

- 죽은 process를 다시 살리는 부분은 대박 !

5) Fastboot

- *원격 부팅 및 flash image fusing*

6) kernel + rootfs로 구성된 boot.img

- 왜 붙여 놓았을까 ? 불편하게 ..
- 부팅 시 boot.img를 통째로 던져 주면, 편할 수 있음. 또한 Security를 염두해 둔 듯 ...

7) recovery kernel(recovery.img)

- 두 개의 kernel(boot.img와 recovery.img 내에 각각 하나씩)

8) Android.mk와 새로운 build system

9) Surfaceflinger & 동영상 출력

- graphic 출력은 당연하지만, 동영상 출력도 surfacflinger를 통해서...

10) 여러 가지를 하나로 담아놓은 system_server와 mediaserver

- 설계의 극치
- 다른 이들이 만들었다면, 아마도 10개 이상되는 프로세스가 만들어 졌을 터 ...

11) adb

12) debuggered와 tombstone

- Good idea

13) toolbox

- Busybox에 비해 좀 불편

14) 그 밖의 커널 추가 사항: wakelock, ashmem, ram console, logger 등 ...

- ram console이 kernel debugging을 위한 것이지만, 좀 더 적극적인 kernel debugging을 위한 노력이 없는 것은 조금 아쉬운 부분

그 동안 임베디드 리눅스를 기반으로 하는 많은 제품이 있었지만, 안드로이드 처럼 모든 것을 재해석하고, 잘 다듬어서 하나로 통합한 것은 없었다. 그것이 사람들이 (아니 필자가) 안드로이드에 열광하는 가장 큰 이유일 것이다.

2. 부팅 순서(Boot Flow) 분석

안드로이드 시스템은 대략적으로 아래의 순서를 따라 부팅을 시작한다. 본 절에서는 이 과정에서 필요한 내용을 간략히 분석해 봄으로써 안드로이드에 대한 이해를 돕고자 한다.

- *부트 로더*
- *리눅스 커널*
- *init process*
- *Native daemons*
- *Zygote(native daemon 중 하나임)*
- *system_server*
- *Java 기반의 응용 프로그램*

2.1 안드로이드 전체 부팅 순서

지금부터는 안드로이드의 전체 부팅 순서를 자세히 확인해 보도록 하자.

<안드로이드 부팅 순서 요약>

1) CPU에 전원이 인가되면, 지정된 번지의 ROM code를 찾아 실행한다. ROM code(Primary Bootloader라고도 함)는 Bootloader를 RAM에 적재시키는 역할을 수행한다.

2) Bootloader는 RAM을 초기화하고, 기본적인 H/W를 초기화하는 작업을 수행한 후, kernel과 RAM disk를 RAM으로 올려준 후, kernel 시작 지점으로 jump하여, kernel을 시작시킨다.

- Bootloader는 단순히 kernel을 구동하는 중간 과정에 머물지 않고, 충전(charging), LCD 출력, emergency 상황 진입, firmware download 등 다양한 역할을 수행하고 있다.
- 시스템에 따라서는 여러 개의 부트로더가 존재하는 경우가 있는데, 여기서 말하는 Bootloader는 Android bootloader(lk or u-boot 등)를 의미한다.

3) Kernel은 C code를 실행하기 위한 환경을 준비한다. 또한 kernel subsystem과 모든 device driver를 초기화하고, root FS를 마운트시킨다. 마지막으로 init process를 구동시킨다.

- 커널은 압축된 형태로 RAM으로 적재되기 때문에, 구동 시점에는 자신의 압축을 직접 풀면서 실행되게 된다.

4) Init process는 먼저 여러 환경 변수를 설정한다. 이어서 마운트 위치를 생성한 후, 여러 파일 시스템을 마운트시키고, 마운트된 파일 시스템의 파일 퍼미션(permission)을 설정한다. 마지막으로 OOM (Out Of Memory) adj 값 등을 지정한 후, 여러 native 데몬을 차례로 실행시킨다.

- Init process는 새로운 문법을 정의하여 사용하고 있는데, 이를 위해 init.rc, init.{hardware}.rc 등의 파일이 사용된다.

5) Init에 의해 여러 native 데몬(daemon)이 실행되는데, 이에는 servicemanager, vold, netd, debuggerd, rild, "app_process –X Zygote", mediaserver, bootanimation, installed, keystore, adbd 등이 있다.

- Init process는 Java application을 직접 실행시켜 주지 않으며, 내부적으로 Zygote를 통하여 java application이 실행되는 형태를 띄고 있다.

6) 이 중, "app_process –X Zygote"는 Java application의 메인 루틴으로써, Dalvik VM을 실행한 후, Zygote의 메인 메서드(함수)를 호출하는 역할을 하게 된다.

- Dalvik VM은 C/C++ code로 구성되어 있으며, 라이브러리 형태로 사용된다.

7) Zygote는 zygote socket을 등록하고, application 구동 속도를 향상 시킬 목적으로 java class와 resource를 미리 로딩해 둔다. 이후 첫 번째 Java application인 System Server process를 먼저 시작시키고, 자신은 socket을 open한 후, 다른 application으로부터 socket connection이 도달하기를 기다린다. 추후에 이 socket으로 연결 요청이 들어오게 되면, System Server를 구동한 방식과 동일하게 새로운 Java application을 구동시켜 주게 된다.

- Zygote가 필요한 이유는 Java application이 Java VM에 의해 컴파일 과정을 거치므로 속도가 느린 단점이 있어 이를 극복하기 위함이다. 이를 위해 java class와 resource를 미리

로딩해 두었다가 새로운 java application에서 바로 사용하도록 되어 있다.

8) Zygote에 의해 실행된 System Server는 여러 시스템 서비스를 초기화하고, 이를 다른 application에서 사용할 수 있도록 서비스 매니저에게 등록해 주는 과정을 거치게 된다. 마지막으로 Activity Manager를 시작시킨다.

9) Activity Manager는 자신을 초기화한 후, intent.CATEGORY_HOME을 전송(startActivity)한다.

- Activity Manager는 이 밖에도 intent.BOOT_COMPLETED를 전송(broadcast)하여 다양한 app들이 실행될 수 있도록 해준다.

10) Launcher app(com.android.launcher)은 Intent.CATEGORY_HOME를 수신한 후, 자신을 초기화(화면 출력) 시킨다. 이 시점부터 사용자는 안드로이드 홈 화면을 눈으로 확인할 수 있는 상태가 된다. Launcher가 수행하는 또 다른 작업은 사용자로부터의 입력을 받아드리기 위하여 onClick( ) 핸들러를 등록하는 부분이다.

11) 마지막으로 사용자로부터의 입력을 받아 새로운 application이 실행된다. 이는 아래의 과정을 통해서 이루어 진다.

- 사용자 action -> Launcher -> Activity Manager -> Zygote -> New application

지금까지 설명한 과정을 그림으로 정리하면 다음과 같다.

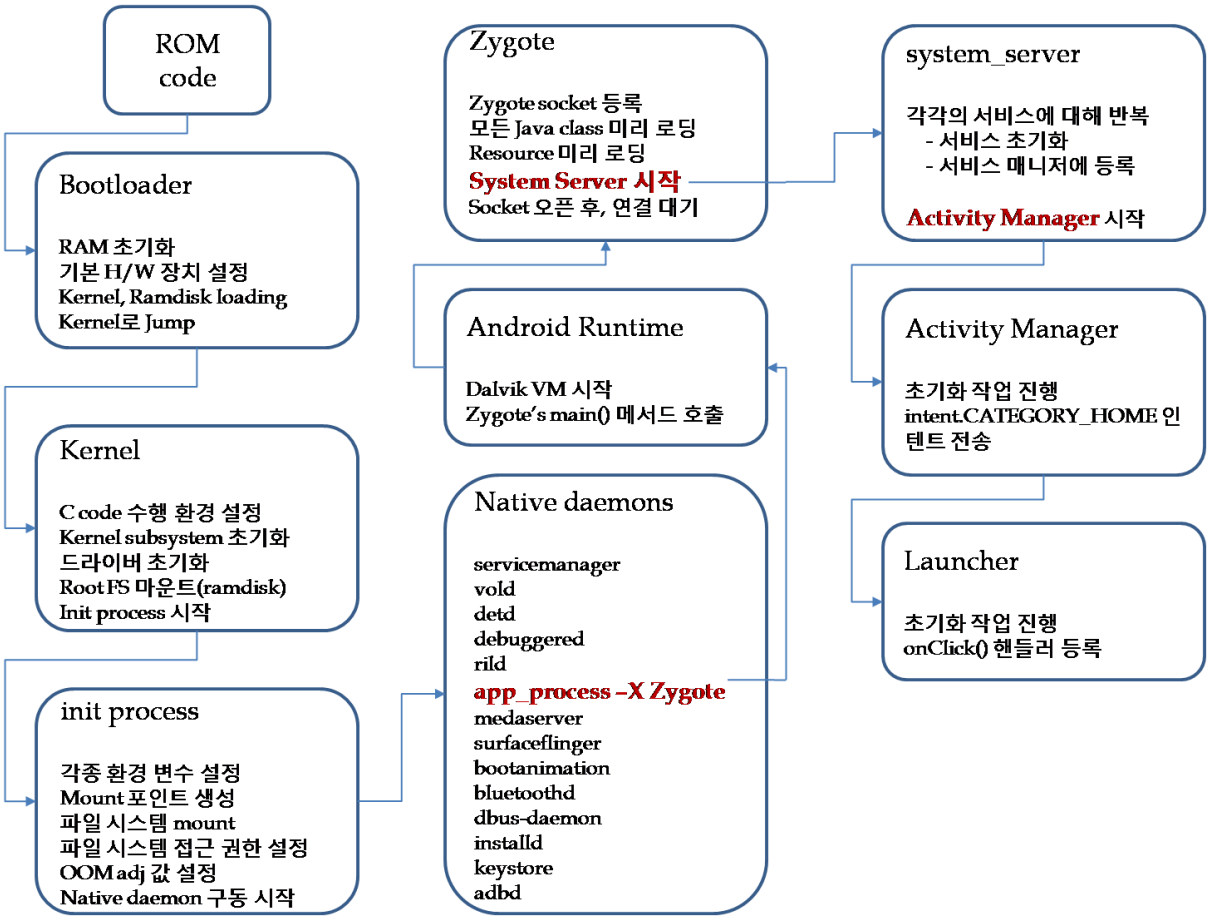

그림 1-5 안드로이드 전체 부팅 순서

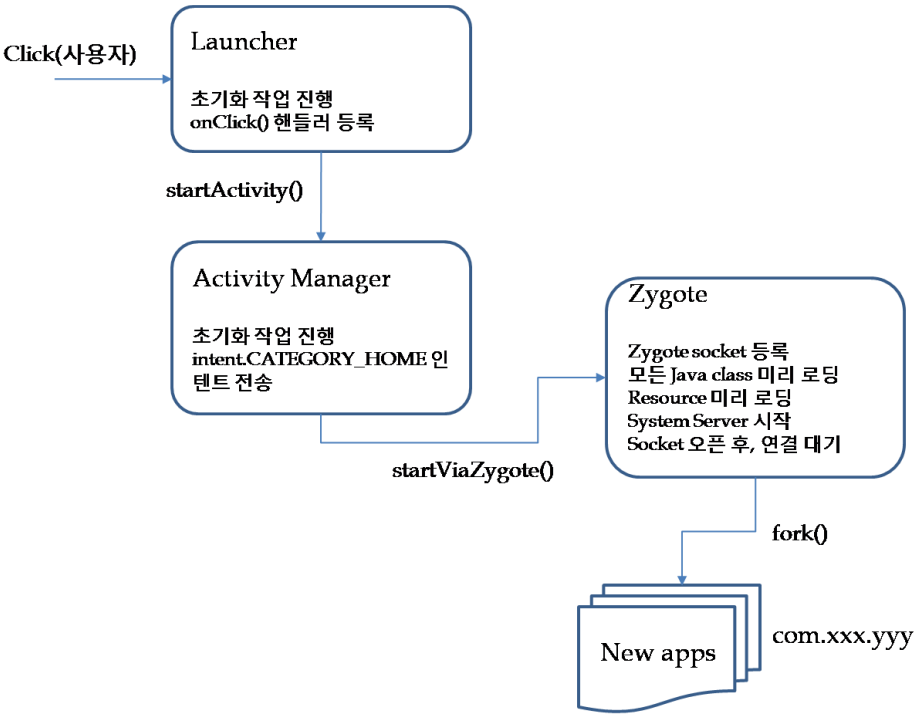

그림 1-6 새로운 안드로이드 응용 프로그램 생성 과정

2.2 부팅의 시작: Primary 및 Secondary 부트로더

부팅 시작을 의미하는 ROM code(Primary Bootloader)와 Secondary Bootloader 등이 실행되는 내용을 먼저 살펴보기로 하자. 이 부분은 vendor 별로 많은 차이를 보이는 부분으로 공통적인 부분만을 추려 정리해 보면 다음과 같다.

<일반적인 Application Processor의 부팅 순서>

1) Primary Boot Loader(줄여서 PBL 이라고도 함)로부터 부팅을 시작한다.

2) Secondary bootloader(줄여서 SBL 이라고도 함)를 eMMC 메모리로부터 Internal SRAM으로 가져와 구동시킨다. 이후 SBL이 다시 Application bootloader(lk 혹은 u-boot) 구동시킨다.

- *Vendor에 따라서는 이 과정이 필요 없이 primary boot loader -> application bootloader로 바로 실행되는 경우도 있다.*
- *반대로, secondary bootloader가 여러 개가 존재하는 경우도 있다.*

3) 마지막으로 Application bootloader가 linux kernel를 DRAM으로 가져와 부팅을 마무리한다.

이상의 과정을 그림으로 정리하면 그림 1-7과 같다.

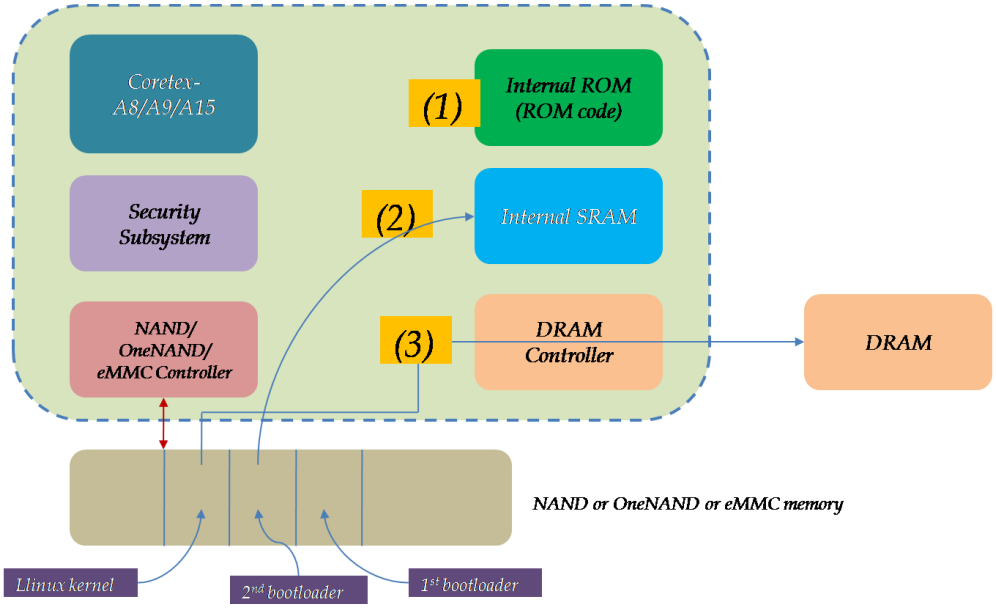

그림 1-7 Primary & Secondary 부트로더

2.3 Application Bootloader

이번 절에서는 application bootloader로 Travis Geiselbrecht, Brian Swetland, Dima Zavin 등이 만든 LK(Little Kernel)라는 부트로더를 소개하고자 한다. 참고로, LK의 경우는 Qualcomm Snapdragon에서 주로 사용되고 있으며, Samsung Exynos나 TI OMAP의 경우는 이 보다는 더 잘 알려진 u-

boot을 사용하고 있다.

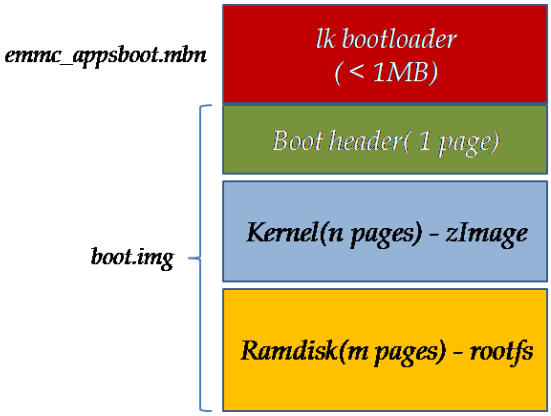

그림 1-8 eMMC 내의 LK bootloader image

<lk bootloader 기능 요약>

1) Linux kernel과 ramdisk를 eMMC 메모리에서 읽어 들어 DRAM으로 올려 놓는다.

2) DRAM 상의 kernel 시작 번지로 jump하여 kernel을 구동시킨다.

3) 사용자의 키 입력에 따라, firmware image download 모드로 진입한 후, USB interface를 통해 firmware image 전체를 download 받는다.

4) 사용자의 키 입력에 따라 fastboot mode로 진입한 후, 파티션 별 firmware 이미지를 download 받는다.

5) 잔류 배터리가 부족하여 충전을 필요로 할 경우, 충전 모드로 진입한다.

6) 정상 부팅 시 화면에 부트 로고를 출력한다.

7) 시스템 디버깅을 목적으로 이전 부팅 시 kernel에 문제가 있었을 경우, 정상 부팅을 멈추고, 디버깅 화면으로 진입한다.

그림 1-9와 1-10은 application bootloader의 가장 중요한 역할인 kernel 구동 과정을 함수 호출 순서를 중심으로 그림으로 표현한 것이다. 단, 아래 내용은 eMMC(혹은 NAND flash)에 이미 커널 및 root file system 이미지가 올라가 있는 경우를 대상으로 한다.

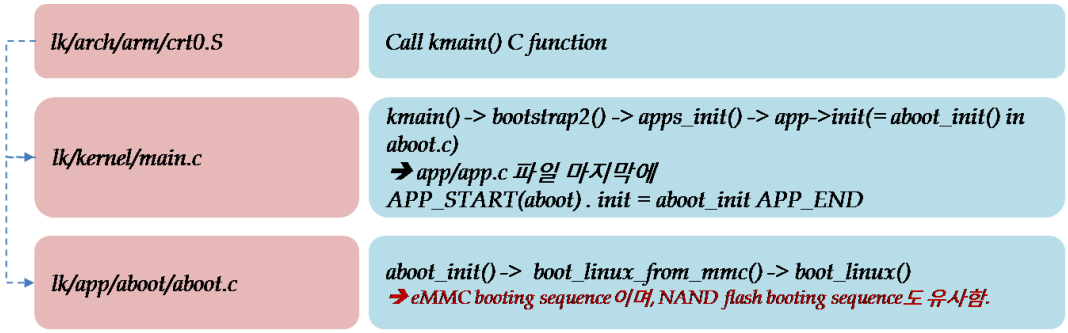

그림 1-9 LK 부트로더 흐름 분석(1) – 전반부

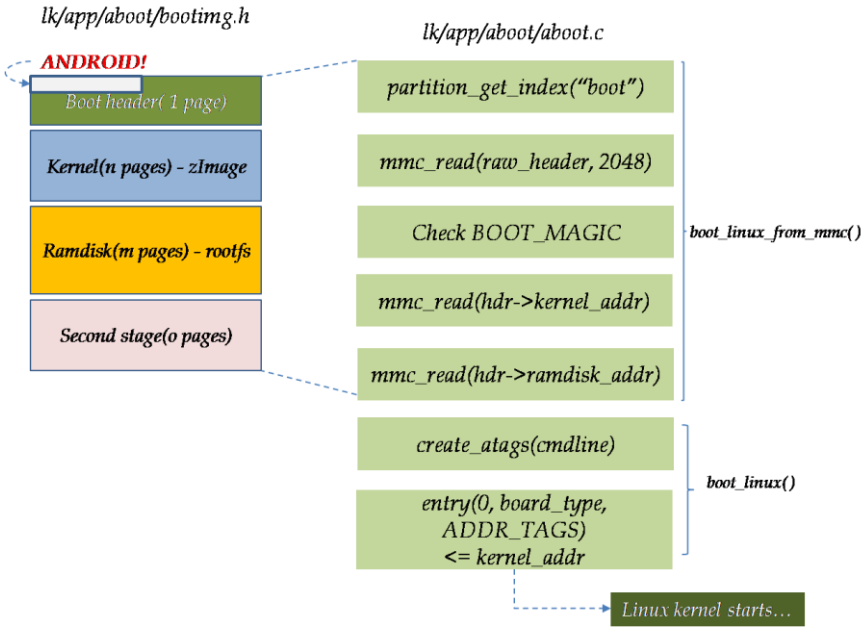

그림 1-10 LK 부트로더 흐름 분석(2) – 후반부

그렇다면, eMMC나 NAND flash에 부팅 이미지가 써져 있지 않은 경우는 어떻게 해야 할까 ? 이와 관련하여 일반적인 경우(안드로이드 이전 방식), 부트로더의 수행을 잠시 멈추고, 네트워크(Ethernet)나 Serial 인터페이스를 통해 커널과 rootfs 이미지를 메모리로 내려 받은 후, 이를 실행하는 형태임을 기억할 것이다.

이번에는 안드로이드의 스마트폰의 경우를 생각해 보자. 일단 안드로이드 스마트폰의 경우는 (일반적으로) ethernet이나 serial port가 없고, USB 인터페이스가 이를 대신할 수 있을 뿐이다. 따라서 기존과는 다른 접근법이 필요할 수 밖에 없게 되었다.

1) 기존 방식(embedded linux)

- 부트로더의 실행을 멈춘다. 대개는 console에서 사용자가 강제로 멈춤.
- 부트로더에서 PC에 있는 내용을 ethernet이나 serial interface를 통해 download 받아 부팅을 진행한다.

2) 안드로이드의 경우

- 부트로더의 실행을 멈춘다. 단말의 특정 키 조합에 의해 강제로 멈춤.
- **PC에서 USB interface를 통해 단말로 이미지를 내리고, 부팅을 진행한다.**
- 물론, 안드로이드를 사용하지만, Ethernet, serial port가 존재하는 경우(예: TV용 보드)는 위의 기존 방식 처럼 운용이 가능할 수도 있다.

Fastboot은 글자 그대로 "빠른 부팅"을 의미한다. 실제로 부트로더를 fastboot 모드로 전환한 상태

에서 PC에서 "$ fastboot boot ~/YOUR_PATH/boot.img" 명령을 실행해 주면, boot.img를 USB 인터페이스를 통해 download 한 후, 바로 부팅을 진행함을 알 수 있다(단, 부트로더에서 해당 기능을 지원해야 함). 이 방법은 커널이나 root file system에 수정 사항이 빈번할 경우, 사용하면 도움이 될 것이다.

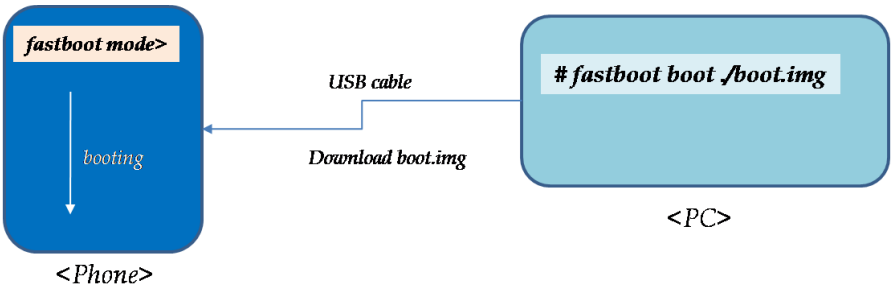

그림 1-11 Fastboot – fast booting

이 밖에도 fastboot를 실제로 자주 사용하는 경우는 e-MMC나 NAND flash 파티션 별 이미지를 퓨징(writing)하는 경우일 것이다.

<fastboot로 image fusing 예>

사용법: fastboot flash <partition name> <image name>

*fastboot flash **aboot** emmc_appsboot.mbn*
*fastboot flash **boot** boot.img*
*fastboot flash **system** system.img*
*fastboot flash **userdata** userdata.img*
*fastboot flash **cache** cache.img*
*fastboot flash **persist** persist.img*
...

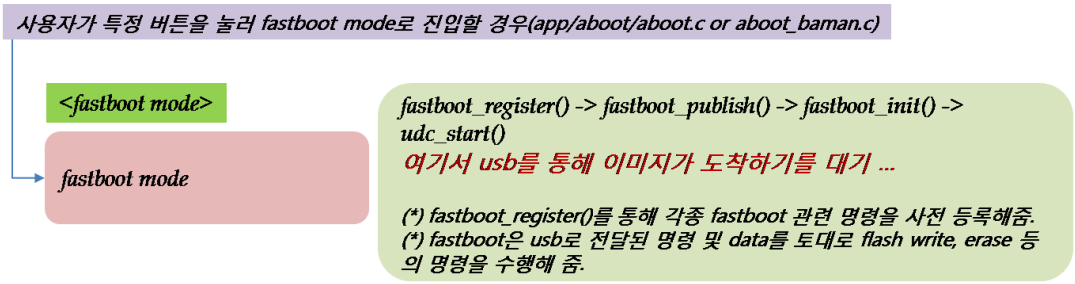

그림 1-12 Fastboot Mode 진입 과정

Fastboot 사용법을 정리해 보면 다음과 같다. 참고로, 아래 fastboot의 모든 기능이 현재 사용중인 시스템에서 정상 동작하는 것을 기대하지는 말기 바란다. 아래 기능이 모두 제대로 동작하려면 bootloader에서 관련 기능을 처리할 수 있는 준비가 이미 되어 있어야 한다.

<fastboot 사용법>

```
fastboot
usage: fastboot [ <option> ] <command>

commands:
  update <filename>                      reflash device from update.zip
  flashall                               flash boot + recovery + system
  flash <partition> [ <filename> ]       write a file to a flash partition
  erase <partition>                      erase a flash partition
  format <partition>                     format a flash partition
  getvar <variable>                      display a bootloader variable
  boot <kernel> [ <ramdisk> ]            download and boot kernel
  flash:raw boot <kernel> [ <ramdisk> ]  create bootimage and flash it
  devices                                list all connected devices
  continue                               continue with autoboot
  reboot                                 reboot device normally
  reboot-bootloader                      reboot device into bootloader
  help                                   show this help message

options:
  -w                                     erase userdata and cache (and format
                                         if supported by partition type)
  -u                                     do not first erase partition before
                                         formatting
  -s <specific device>                   specify device serial number
                                         or path to device port
  -l                                     with "devices", lists device paths
  -p <product>                           specify product name
  -c <cmdline>                           override kernel commandline
  -i <vendor id>                         specify a custom USB vendor id
  -b <base_addr>                         specify a custom kernel base address
  -n <page size>                         specify the nand page size. default: 2048
  -S <size>[K|M|G]                       automatically sparse files greater than size.
                                         0 to disable
```

마지막으로 LK 소스 코드의 전체 구성을 개략적으로 정리해 보았으니, 참고하기 바란다.

<bootable/bootloader/lk 디렉토리 내용 개략 정리>

*1) bootable/bootloader/lk/project/YOUR_BOARD.mk**

- project makefile 위치

*2) bootable/bootloader/lk/app/aboot**

- aboot.c : flash/mmc에서 kernel image 읽어 kernel loading 및 start하는 코드
- fastboot.c: usb cable이용한 fastboot 처리 코드
- recovery.c: recovery 관련 코드

*3) bootable/bootloader/lk/arch**

- arm/ 디렉토리 아래에 실제 arm 환경에서의 boot 관련 코드 위치함(assembly code 다수 존재함).
- 아래 (5)의 main(kmain) code를 호출하는 부분 존재함.

4) bootable/bootloader/lk/dev

- device driver(fbcon, keys, net, pmic, ssbi, usb)

*5) bootloader/lk/kernel**

- boot main(kmain) 이 위치함.
- kernel의 특성에 해당하는 코드(mutex, thread, timer, event ..)

6) bootloader/lk/lib

- lk에서 사용하는 library codes

7) bootable/bootloader/lk/make

*8) bootable/bootloader/lk/platform/YOUR_BOARD**

- platform specific codes(acpuclock, gpio, hdmi, lcd panel, pmic, pmic battery alarm ...)

9) bootable/bootloader/lk/scripts

- 몇 가지 script 위치함

*10) bootable/bootloader/lk/target**

- YOUR_BOARD/ 디렉토리 아래에 target board와 관련한 몇 가지 초기화 관련 코드 위치함.
- target_init( ) 함수는 위의 (5)의 kmain( ) 함수에서 호출함.

2.4 Linux Kernel과 Ramdisk

LK bootloader는 리눅스 커널과 램 디스크 이미지(ramdisk image)를 eMMC로부터 읽어 들여 DRAM에 적재한다. 이어서 커널의 시작 지점으로 jump하여 커널을 시작한다. 커널은 압축된 형태로 RAM으로 적재되기 때문에, 구동 시점에는 자신의 압축을 직접 풀면서 실행되게 된다. 커널 부팅 관련 보다 자세한 사항은 2장을 참고하기 바라며, 여기서는 중요한 부분만 간략히 요약 정리해 보는 것으로 하겠다.

<kernel 부팅 요약>

1) Kernel과 rootfs(Ramdisk) 이미지를 RAM으로 loading한다.

2) LK bootloader가 kernel의 시작번지로 jump하면, kernel은 자신의 압축을 풀기 시작한다.
3) Kernel은 C code를 실행하기 위한 환경을 준비한다.
4) kernel subsystem을 초기화한다.
5) 각종 device driver를 초기화한다.
6) Ramdisk root FS를 마운트시킨다.
7) init process를 구동시킨다.
8) Idle 상태에서 대기한다.

이 상의 과정을 그림으로 정리해 보면 그림 1-13 및 1-14와 같다.

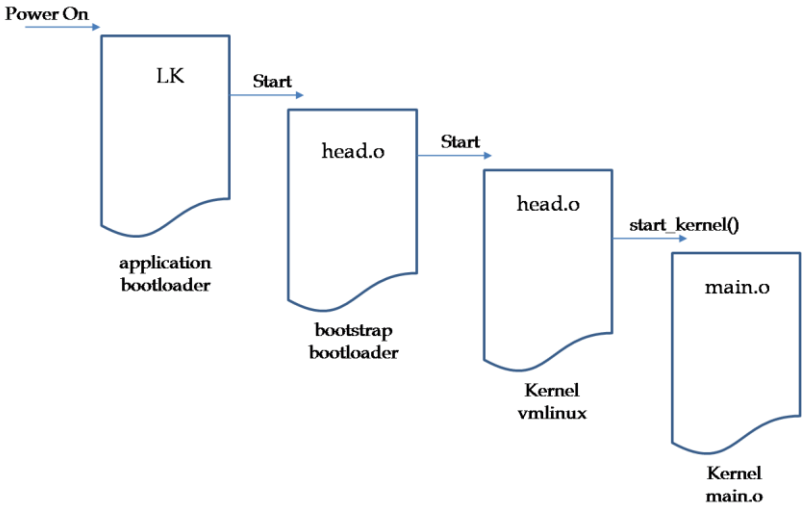

그림 1-13 부트로더, 부트스트랩 로더, 그리고 kernel

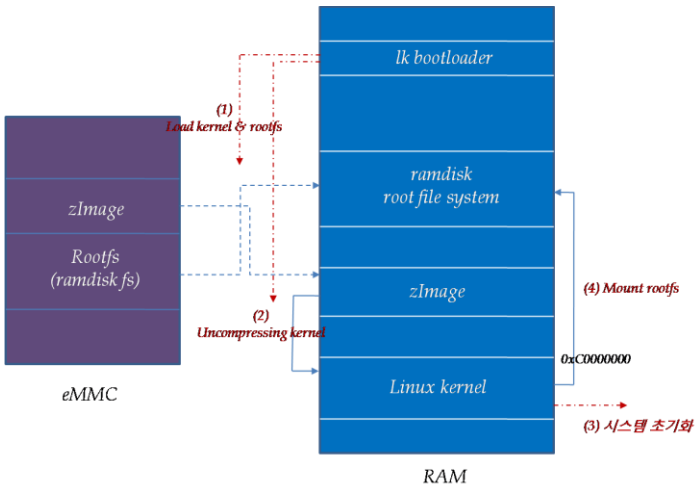

그림 1-14 kernel과 rootfs

위의 커널 부팅 과정을 통해 여러 파일 시스템이 구동 및 마운트되어 사용되게 되는데, 이를 그림 1-15에 정리해 보았다. 실제로 파일 시스템을 마운트하는 과정은 앞으로 설명하게 될 init

process에서 하게 되지만, 편의상 이곳에서 미리 정리해 보았다.

그림의 내용을 잠깐 설명해 보면, 우선 좌측이 RAMDISK 방식으로 구성된 rootfs를 나타내며, 우측의 /system, /data, /cache는 eMMC 영역을, /mnt/sdcard는 외장형 sdcard 영역을 의미한다.

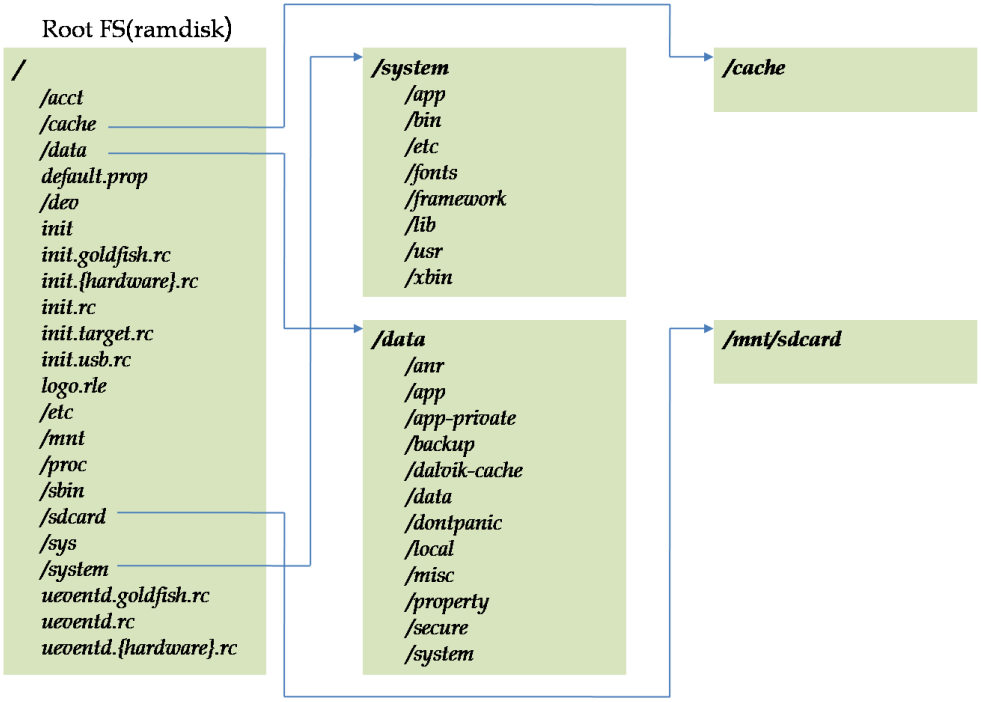

그림 1-15 안드로이드 Root 파일시스템

참고로 /system과 /data 파티션을 수동(shell 진입 후 사용자가 수행) 및 자동(init process가 실행) 마운트해 주는 명령을 정리해 보면 다음과 같다.

<busybox 명령을 사용한 /system, /data 파티션 수동 마운트>

 # mount –o remount,rw –t ext4 /dev/block/mtdblockX /system

 # mount –o remount,rw –t ext4 /dev/block/mtdblockY /data

- 이 방법은 사전에 busybox가 설치되어 있어야 함.

<init.rc에서의 표현 방식 - /system, /data 파티션 마운트>

mount ext4 /dev/block/mtdblockX /system nosuid nodev barrier=1

mount ext4 /dev/block/mtdblockY /data nosuid nodev barrier=1

2.5 init process와 init.rc

init은 kernel이 부팅의 거의 마지막 단계에서 실행시켜 주는 최초의 application process로써, 이후의 모든 user space의 application이 init을 통해 실행되고 관리된다. 앞서도 이미 언급했다시피, 안

드로이드에서는 init process를 매우 중요하게 보고, 그 기능을 대폭 확대하였다. Init process의 기능을 간략히 요약하면 다음과 같다.

<init 기능 process 요약>

1) init.rc, init.{hardware}.rc 처리

- Native daemon 구동
- 즉, servicemanager, vold, netd, debuggerd, rild, "app_process –X Zygote", mediaserver, bootanimation, installed, keystore, adbd 등 실행

2) SIGCHLD signal 처리

- Child process가 죽으면 다시 살림.

3) System property 설정

…

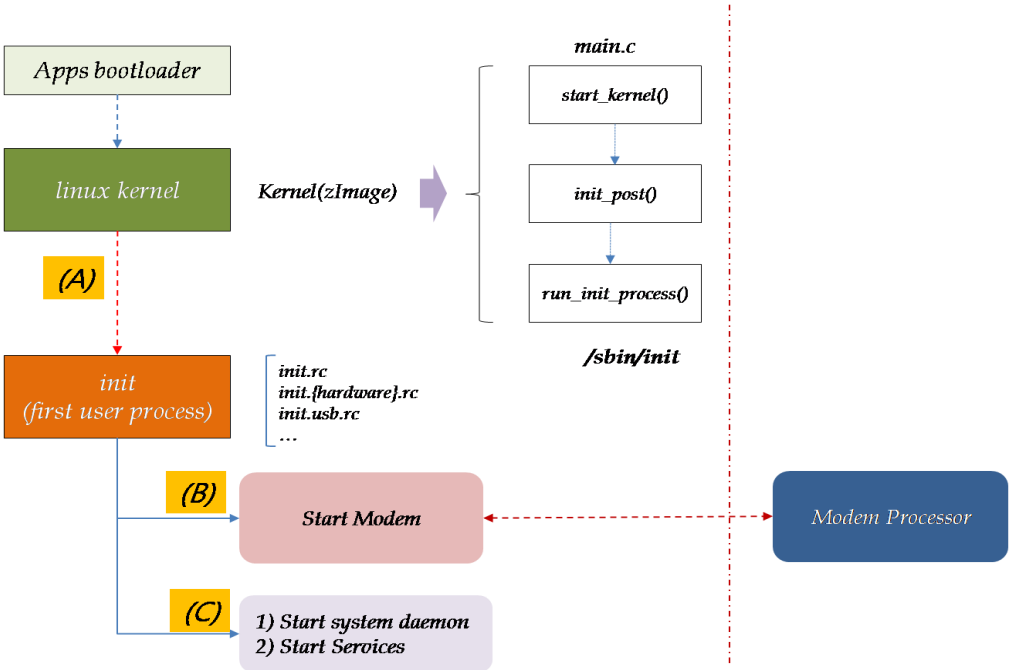

그림 1-16 init process의 시작

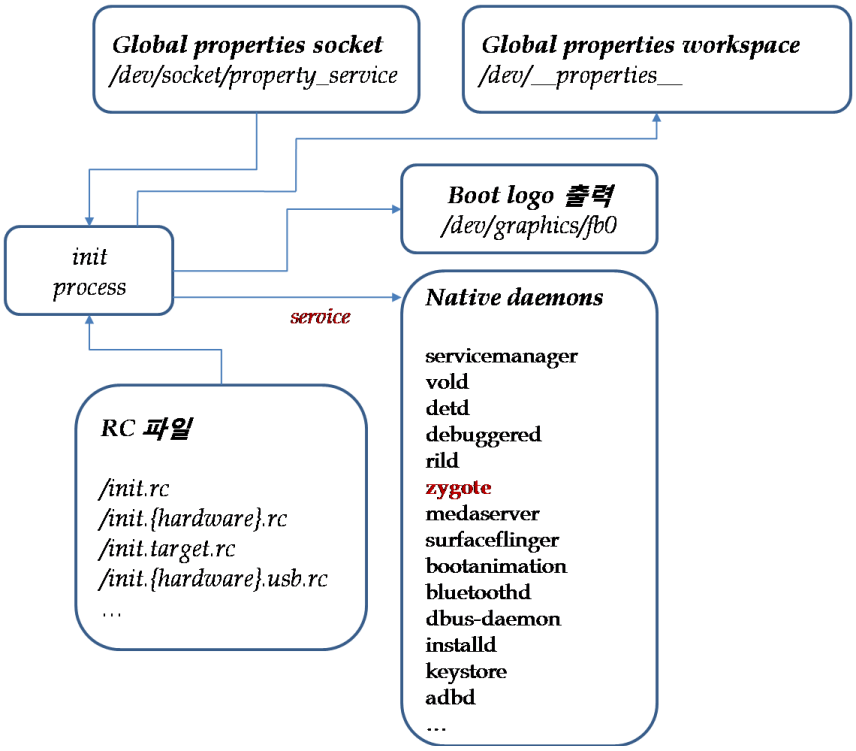

그림 1-17 init process의 역할 [다시 그려야 함]

Init rc 파일은 복수개의 action과 service로 구성되어 있으며, 각각의 형식(문법)은 다음과 같다.

#action

on <trigger>

<command>

<command>

<command>

...

#service

*service <name> <pathname> [<argument>]**

<option>

<option>

<option>

...

이러한 action과 서비스의 묶음으로 구성된 init.rc/init.{hardware}.rc 등의 파일을 처리하는 순서를 살펴 보면 init process의 역할을 대략적으로 짐작할 수 있을 것이다.

<init process 코드 흐름 분석>

1) early-init 명령을 실행한다.

2) coldboot: ueventd가 device를 성공적으로 초기화 했는지 검사한다.

3) property service의 내부 데이터 구조를 초기화한다.
4) keychord를 위한 핸들러를 셋업한다.
5) 콘솔을 초기화하고, 로그 화면 혹은 문자를 출력한다.
6) ro.serialno, ro.baseband, ro.carrier 같은 초기 속성을 설정한다.
7) init 명령을 실행한다.
8) early-fs 명령을 실행한다.
9) fs 명령을 실행한다.
10) post-fs 명령을 실행한다.
11) service 명령을 실행한다.
12) SIGCHLD 시그널을 받을 준비를 한다.
13) service socket과 SIGCHLD 핸들러가 준비되었는지를 확인한다.
14) early-boot 명령을 실행한다.
15) boot 명령을 실행한다.
16) 모든 triggered 명령을 처리한다.

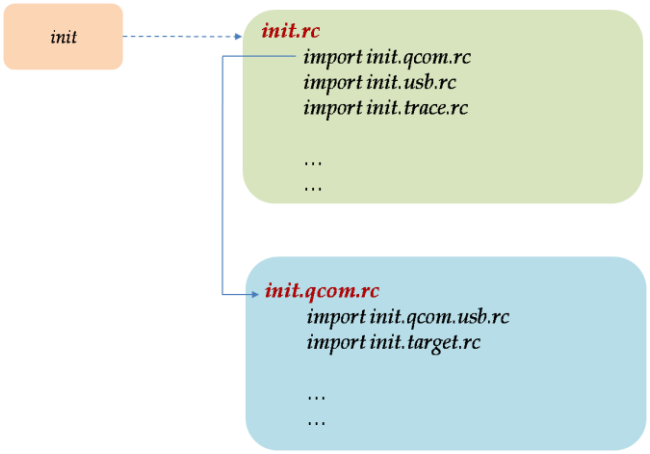

그림 1-18 init – init.rc 구동(1)

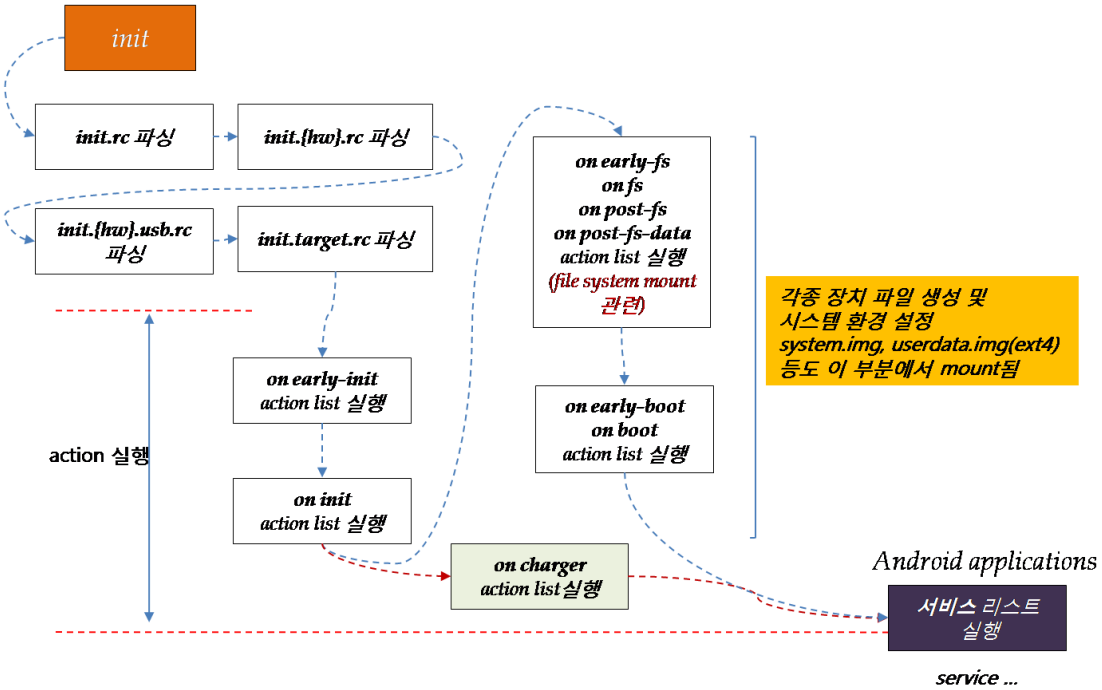

그림 1-19 init – init.rc 구동(2)

<init.rc 내용 발췌>

```
import /init.usb.rc
import /init.${ro.hardware}.rc
import /init.trace.rc

on early-init
    # Set init and its forked children's oom_adj.
    start ueventd
    # create mountpoints
    mkdir /mnt 0775 root system
    ...

on init
    # setup the global environment
    export PATH /sbin:/vendor/bin:/system/sbin:/system/bin:/system/xbin
    export LD_LIBRARY_PATH /vendor/lib:/system/lib
    ...

on fs
    # Mount /system rw first to give the filesystem a chance to save a checkpoint
    mount yaffs2 mtd@system /system     //예전 방식임. 현재는 ext4 사용함.
    ...
```

```
on post-fs
    mount rootfs rootfs / ro remount
    ...

on post-fs-data
   ...

on boot
    # basic network init
    ifup lo
    hostname localhost
    domainname localdomain

    # set RLIMIT_NICE to allow priorities from 19 to -20
    setrlimit 13 40 40

    # Memory management.  Basic kernel parameters, and allow the high
    # level system server to be able to adjust the kernel OOM driver
    # parameters to match how it is managing things.
    write /proc/sys/vm/overcommit_memory 1
    ...

on nonencrypted
    ...

on charger
   ...

on property: vold.decrypt=trigger_reset_main   //지정된 속성이 설정될 경우 trigger됨
    ...

service ueventd /sbin/ueventd
    ...
service adbd /sbin/adbd
    ...
service servicemanager /system/bin/servicemanager
    class core
    user system
    group system
    critical
    onrestart restart zygote
    onrestart restart media
    onrestart restart surfaceflinger
```

```
    onrestart restart drm

service vold /system/bin/vold
    ...

service netd /system/bin/netd
    ...

service debuggerd /system/bin/debuggerd
    ...

service ril-daemon /system/bin/rild
    ...

service surfaceflinger /system/bin/surfaceflinger
    class main
    user system
    group graphics drmrpc
    onrestart restart zygote

service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
    class main
    socket zygote stream 660 root system
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd

service drm /system/bin/drmserver
    ...

service media /system/bin/mediaserver
    class main
    user media
    group audio camera inet net_bt net_bt_admin net_bw_acct drmrpc qcom_diag
    ioprio rt 4

service bootanim /system/bin/bootanimation
    ...

service installd /system/bin/installd
    ...
```

```
service keystore /system/bin/keystore /data/misc/keystore
    ...
```

<init.{hardware}.rc 내용 발췌>

```
import init.qcom.usb.rc
import init.target.rc

on init
    # Set permissions for persist partition
    ...

on early-boot
    # set RLIMIT_MEMLOCK to 64MB
    ...

on boot
    chown bluetooth bluetooth /sys/module/bluetooth_power/parameters/power
    chown bluetooth bluetooth /sys/class/rfkill/rfkill0/type
    ...
    chmod 0660 /sys/module/bluetooth_power/parameters/power
    chmod 0660 /sys/module/hci_smd/parameters/hcismd_set
    ...
    write /proc/sys/net/ipv6/conf/rmnet0/accept_ra 2
    write /proc/sys/net/ipv6/conf/rmnet1/accept_ra 2
    write /proc/sys/net/ipv6/conf/rmnet2/accept_ra 2
    ...

on post-fs
    ...

on post-fs-data
    # msm specific files that need to be created on /data
    ...

service usbhub_init /system/bin/usbhub_init
    ...

on property:init.svc.surfaceflinger=stopped
    ...

service qcom-c_core-sh /system/bin/sh /init.qcom.class_core.sh
    ...
```

```
service rmt_storage /system/bin/rmt_storage
    ...

on property:ro.boot.emmc=true

service config_bluetooth /system/bin/sh /system/etc/init.qcom.bt.sh "onboot"
    ...
```
이하 on property 문과 service 문 반복

ueventd는 init process가 실행시켜 주는 최초의 native daemon으로, ueventd.rc와 ueventd.{hardware}.rc 파일을 토대로 각종 장치 노드(/dev)를 정적으로 생성시켜 줌은 물론이고, 커널에서 동적으로 올라오는 uevent(kobject_uevent() 함수 호출)를 토대로 각종 장치 노드(/dev)를 동적으로 생성 및 초기화 시켜 주는 역할을 담당한다.

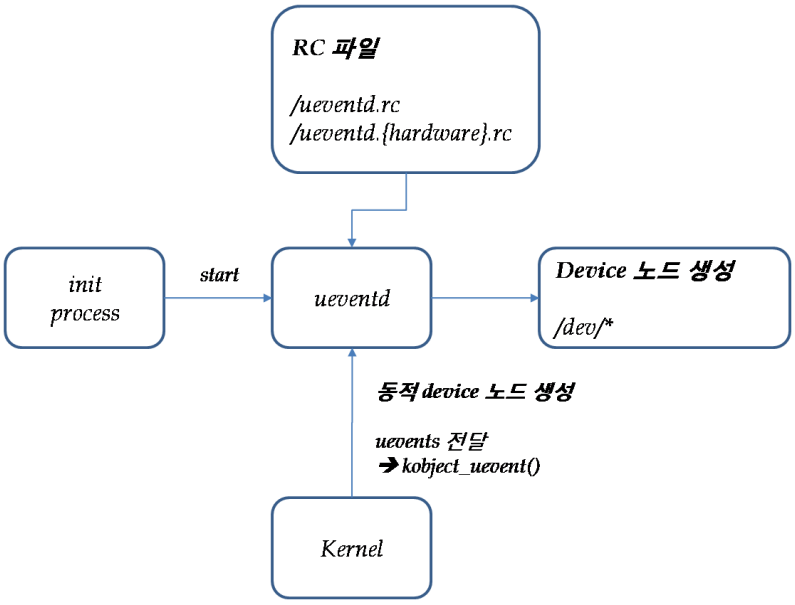

그림 1-20 ueventd 구동

지금까지 init process의 전체적인 역할 및 여러 native daemons를 실행하는 부분까지 살펴 보았다. 이상의 내용을 토대로, init process가 실행하는 여러 native daemons(service manager, mediaserver, app_process -Xzygote 등 포함)와 zygote에 의해 실행되는 system_server 및 com.android.phone 까지의 구동 관계를 하나의 그림으로 정리해 보면 다음과 같다.

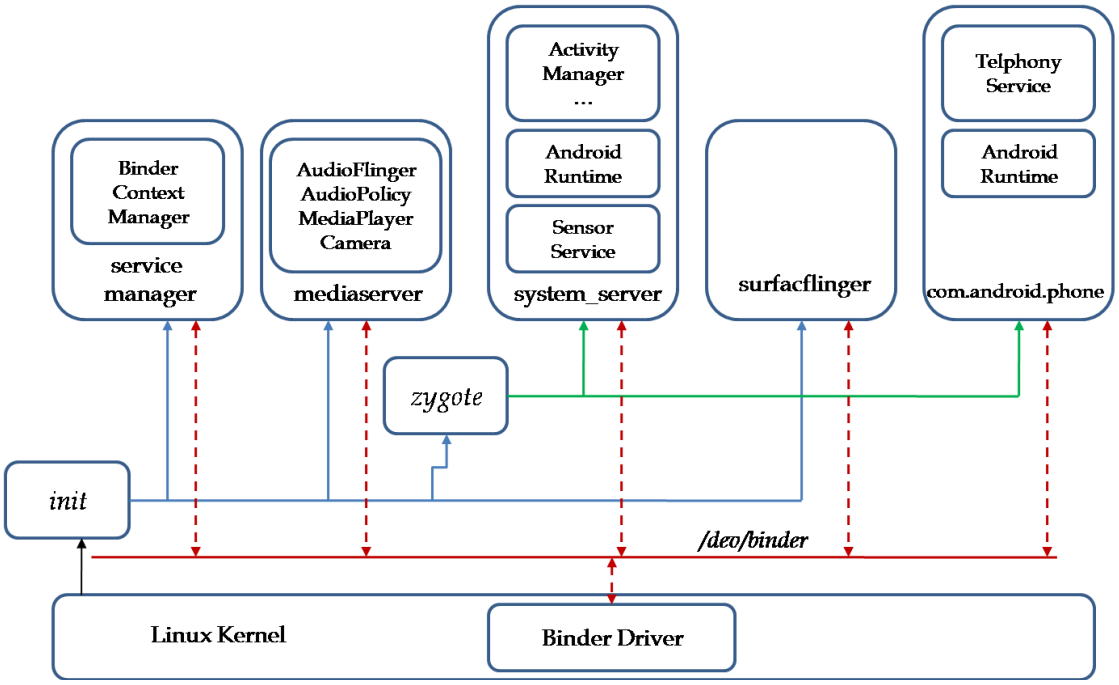

그림 1-21 init 이후의 부팅 과정

2.6 service manager와 binder(바인더)

다음으로 확인해 볼 사항은 안드로이드에서 새롭게 제안한 IPC 기법인 binder와 service manager에 관한 것이다. 사실 안드로이드 처럼 거대한 시스템을 구축하다 보면, 반드시 고려해야만 하는 부분이 프로세스 간의 적절한 통신 방식일 것이다. 실제로 안드로이드 이전 시절에도 리눅스에는 다양한 프로세스간 통신 방법이 존재해 왔다.

<리눅스의 전통적인 프로세스간 통신 방식>

1) file을 이용한 원시적인 방법

2) parent와 child process간에 사용하는 pipe

3) domain socket – 보편적으로 많이 사용하는 방식. 사용이 쉬움. 예: dbus(domain socket 활용)

4) message queue – kernel의 message queue를 활용하는 단 방향 통신 방식

5) Shared memory – 통신 방식이라기 보다는 프로세스 간의 메모리 공유 방식으로, 프로세스간의 통신으로 확장 가능

- 안드로이드에서는 ashmem으로 확장 시켜 사용하고 있음.

6) Sun RPC – remote procedure call

7) ...

안드로이드에서는 아래와 같은 요구 사항을 만족하는 프로세스간의 통신 기법을 설계하고자 하였다.

1) 하나의 통일된 프로세스간 통신 방법의 필요성을 느낌.

2) Remote procedure call 기능이 필요함. 즉, 원격 프로세스의 함수 호출이 가능해야 할 필요성을 느낌.

3) 프로세스간의 대량의 데이터 교환을 위한 방식이 필요하게 됨.

- 오디오 데이터, 비디오 데이터, 그래픽 화면 등 전달 용
- ashmem으로 구현함(3장 3.4절 참조)

안드로이드에서는 커널을 적극 활용하여 binder라는 새로운 개념(정확히는 binder driver, servicemanager daemon)을 만들고, 이를 전체 프레임워크의 대표 통신 방법으로 사용하게 되었다.

<binder와 service manager의 단순 동작 원리>

1) 서비스 서버는 자신이 구현한 서비스를 service manager에게 등록한다.

- 이 과정에서 binder 드라이버를 경유하게 됨.

2) 서비스 클라이언트는 사용할 서비스를 service manager를 통해 검색한다.

- 이 과정에서 binder 드라이버를 경유하게 됨.

3) 마침내, 서비스 클라이언트는 서비스를 사용(Remote Procedure Call)하게 된다.

- 이 과정에서 binder 드라이버를 경유하게 됨.

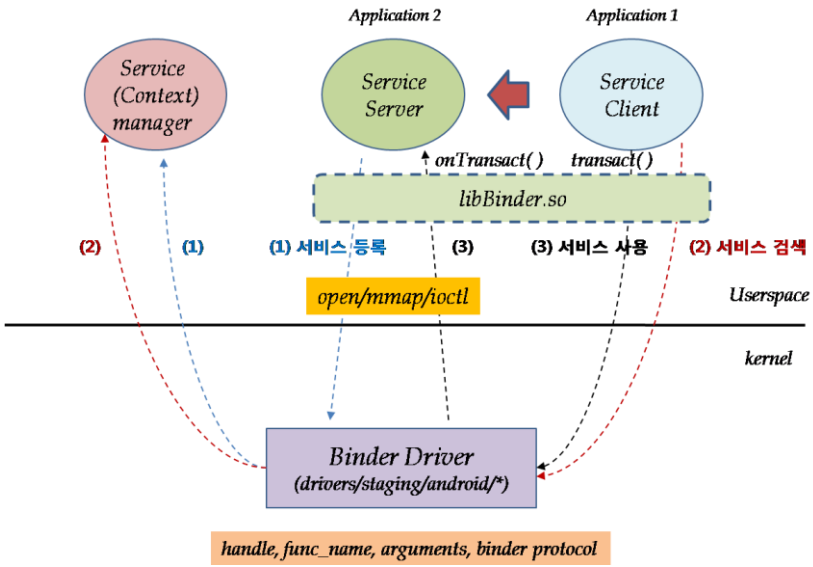

그림 1-22 서비스 매니저와 바인더(1)

Binder 드라이버의 동작 원리를 간략히 기술해 보면 다음과 같다.

A. Process A -> B로 데이터 전송

1) Process A는 ioctl() 함수를 사용하여 데이터 전송(write)을 시작한다. 한편 Process B는 데이터가 전달되기를 기다린다(sleep).

2) binder_thread_write 커널 코드

- copy_from_user() 함수를 사용하여 사용자 영역의 버퍼 내용을 커널 버퍼로 복사해 온다.
- wake_up_interruptible() 함수를 호출하여 대기 중인 Process B를 깨운다.

3) binder_thread_read 커널 코드

- wait_event_interruptible() 함수를 호출하여 데이터가 준비되기를 기다린다.
- wake_up_interruptible() 함수가 호출되면, copy_to_user() 혹은 put_user() 함수를 사용하여 수신 데이터를 사용자 영역(process B)으로 전달한다.

4. 대기 중이던, Process B는 깨어나, ioctl() 함수를 사용하여 데이터를 읽어 들인다.

B. Process A <- B로 데이터 전송

- 이의 내용과 동일한 원리로 진행함.

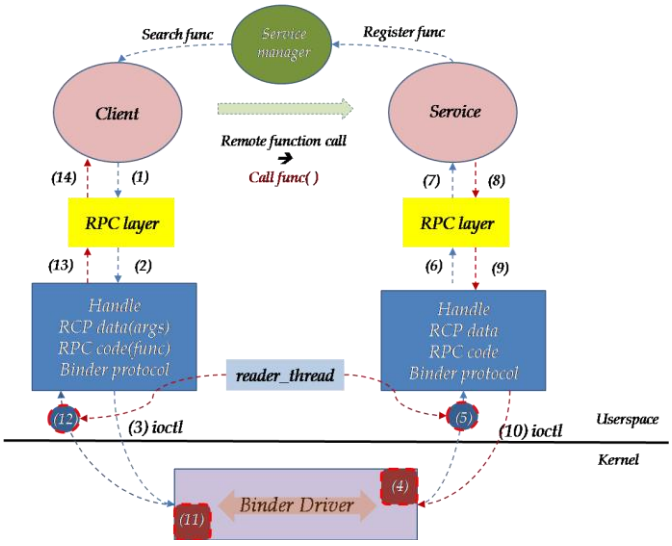

그림 1-23 Binder의 동작 원리(1)

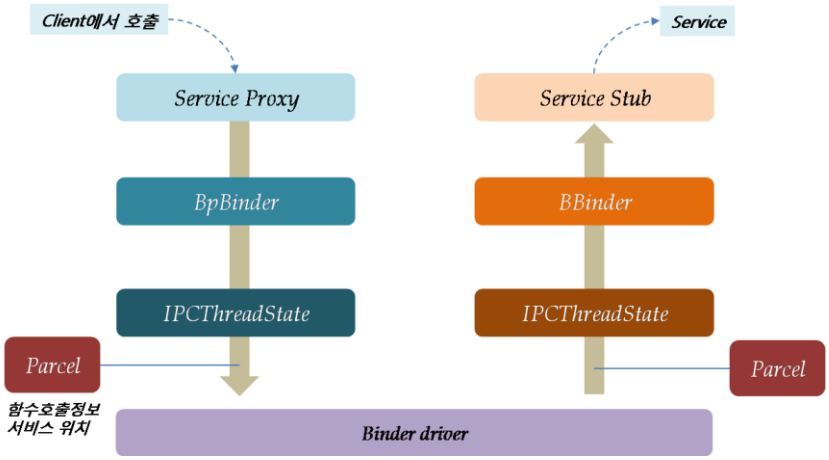

그림 1-24 바인더의 동작 원리(2)

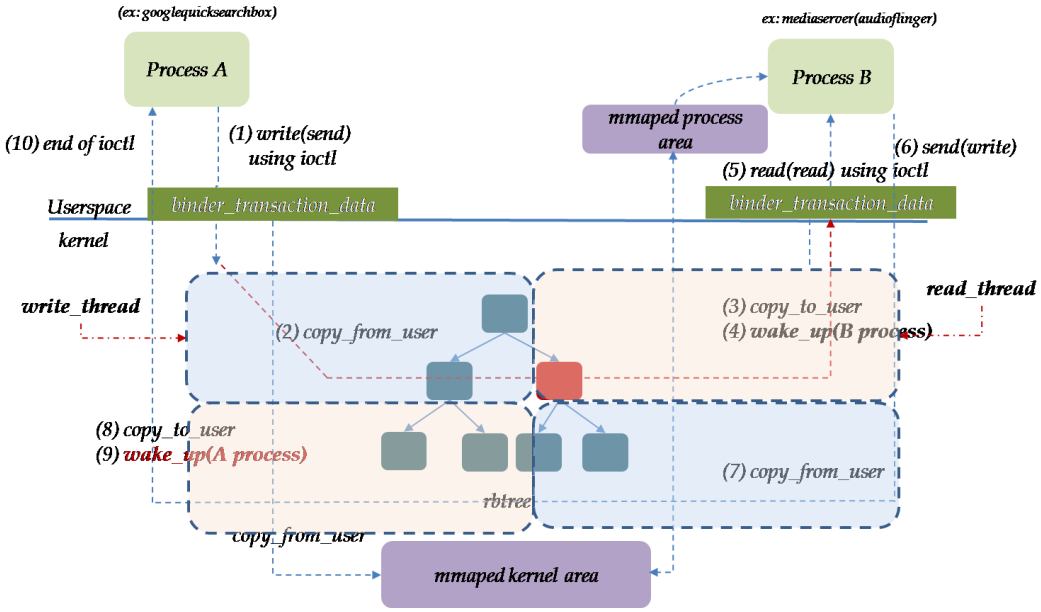

그림 1-25 바인더의 동작 원리(3)

실제로 바인더 관련 코드는 상당히 복잡한 편이다. 바인더 관련하여 보다 자세한 사항을 위해서는 참고 문서 [4-6]을 참조하기 바란다.

2.7 Zygote와 안드로이드 런타임(Runtime)

이번 절에서 살펴볼 내용은 Java 기반의 응용 프로그램의 parent process 격인 Zygote와 android Runtime에 관한 것이다.

먼저 Zygote라는 개념이 왜 등장하게 되었는지를 살펴 보자.

<Zygote process의 필요성>

1) 안드로이드 응용 프로그램은 Java를 기반으로 하며, 따라서 VM(Dalvik)이 반드시 필요하다. 이러한 구조는 Native 방식(C/C++ only)의 프로그램에 비해 속도가 느린 단점이 있다.
2) 따라서, Java 기반의 응용 프로그램의 구동 속도를 개선할 목적으로 Zygote 개념이 도입되었다.
3) Zygote process는 먼저 구동되어 있으면서, 속도에 영향을 주는 Java Class, Resource 등을 미리 메모리에 로딩해 두었다가, 응용 프로그램으로 부터의 요청(일종의 fork)이 있을 시, 미리 로딩해 준 자원을 공유해 줌으로써 응용 프로그램의 구동 속도를 향상시킬 수 있게 된다.

다음 내용은 Android Runtime의 정의해 해당하는 내용이다.

<Android Runtime의 정의>

1) Dalvik VM과 Core 라이브러리를 합쳐서 Android Runtime이라고 부르며, Sun의 그것과는 달리 독자적으로 구현되었다.

- Dalvik VM은 HeapWorker, GC(Garbage Collector), Signal Catcher, Compiler, JDWP 및 Binder Thread 등으로 구성되어 있다.

2) Core 라이브러리에 해당하는 내용으로는 다음과 같은 것들이 있다.

- core.jar, core-junit.jar, bouncycastle.jar, ext.jar, framework.jar, telephony-common.jar
- mms-common.jar, androidpolicy.jar, services.jar, apache-xml.jar, telephony-msim.jar
- qcmediaplayer.jar, WfdCommon.jar 등

3) Android Runtime은 Zygote 초기화 과정에서 사용(호출)된다.

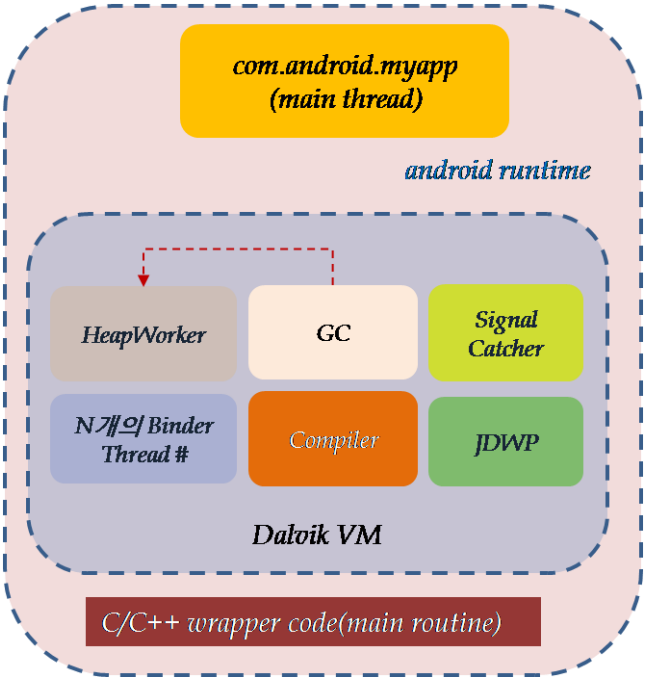

그림 1-26 Dalivk VM과 안드로이드 Runtime

이제 부터는 Zygote가 동작하는 방식을 살펴 보기로 하자.

<Zygote의 동작 원리>

1) init process가 아래 명령을 실행한다.

- /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server

2) frameworks/base/cmds/app_process/app_main.cpp 코드가 수행되어, 아래 메서드를 호출한다.

- runtime.start("com.android.internal.os.ZygoteInit", startSystemServer ?

"start-system-server" : "");

3) JNI_CreateJavaVM( )를 호출하여 가상머신을 생성하고 초기화 시킨다.
4) ZygoteInit::main( )을 시작한다.
5) Domain socket binding 한다.
6) VM, Class, resource preloading 한다.
7) SystemServer를 시작한다.
8) (Activity Manager) Application으로 부터의 요청이 오기를 기다린다.

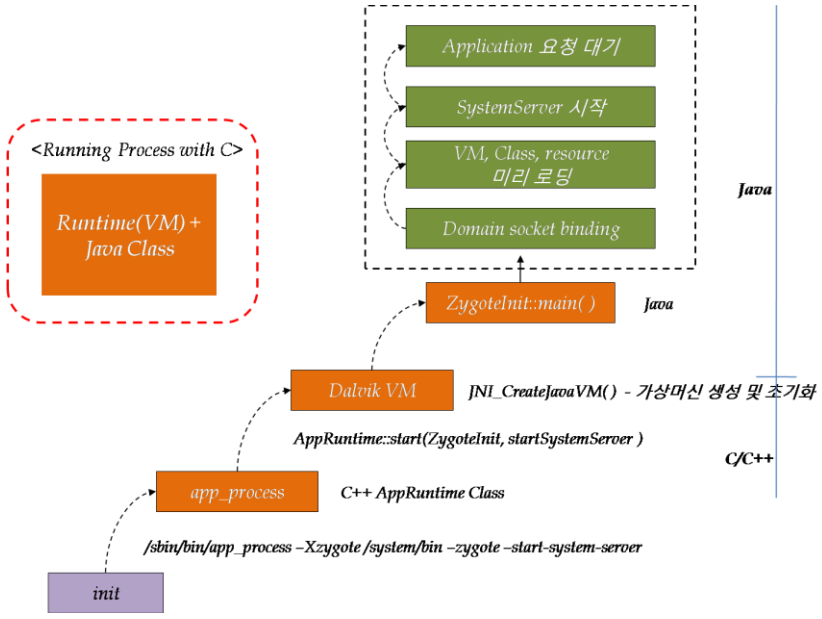

그림 1-27 Zygote와 Dalvik VM(1)

다음 그림은 zygote process(pid: 97)와 zygote에 의해 파생된 process(예: system_server)의 ppid(parent pid)가 동일함을 보여주기 위해 마련하였다. 이는 앞서 설명한 바와 같이 system_server를 비롯한 다수의 Java 기반의 응용 프로그램(com.XXX.YYY 형태)이 zygote로부터 파생되었음을 보여준다.

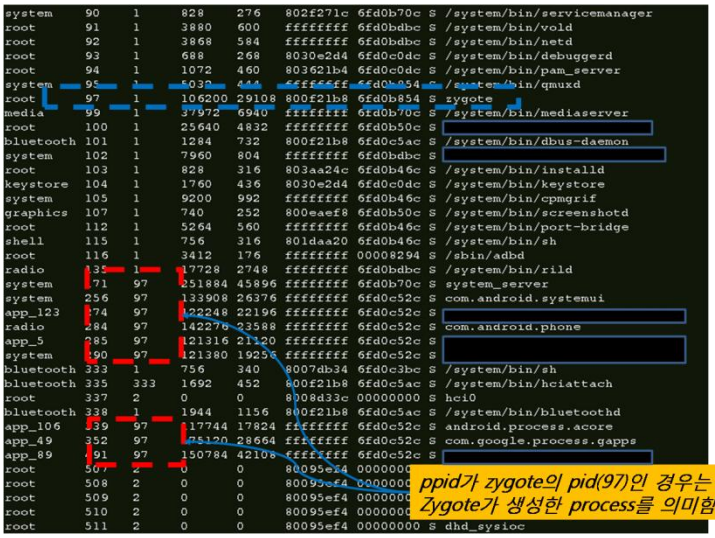

그림 1-28 Zygote와 Dalvik VM(3)

2.8 시스템 서비스

이번 절에서 살펴 볼 내용은 앞서 2.6절에서 설명한 binder & service manager에 기반하는 다양한 서비스(server 기능)들로써, system_server(Java, C/C++ 기반), surfaceflinger(C/C++ 기반), mediaserver(C/C++ 기반), Phone service(Java 기반) 등이 그 대상이라고 볼 수 있다.

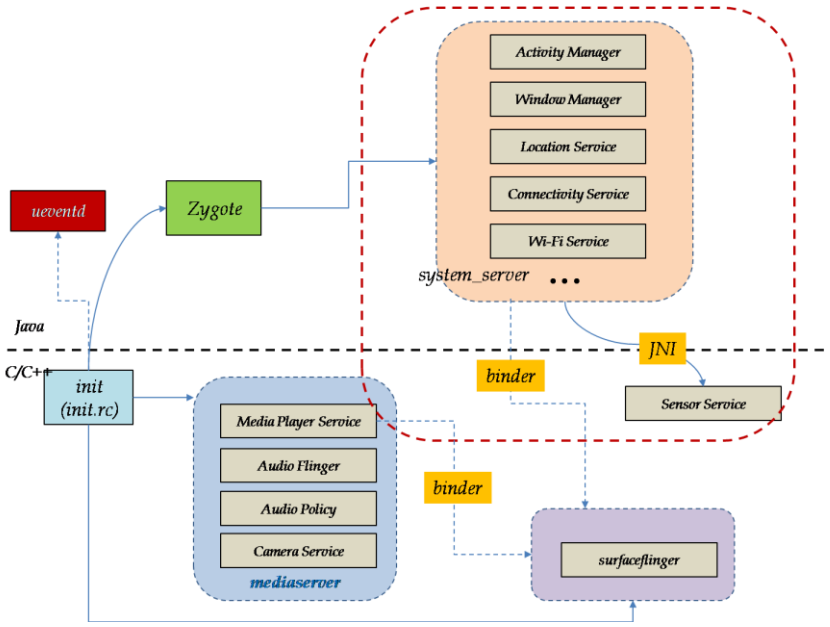

그림 1-29 System Service 구동

아래 그림은 안드로이드에서 제공하는 모든 시스템 서비스를 총 망라해서 표현한 것으로, Java로 작성한 서비스와 C/C++로 작성한 서비스가 조화를 이루고 있다. 특히 이중, system_server는 그야말로 여러 가지 기능을 하나의 프로세스로 통합하고 있는 super service daemon이라고 할 만하다.

<system_server의 주요 구성 요소>

1) Activity Manager

2) Window Manager

3) Package Manager

4) Power Manager

5) ...

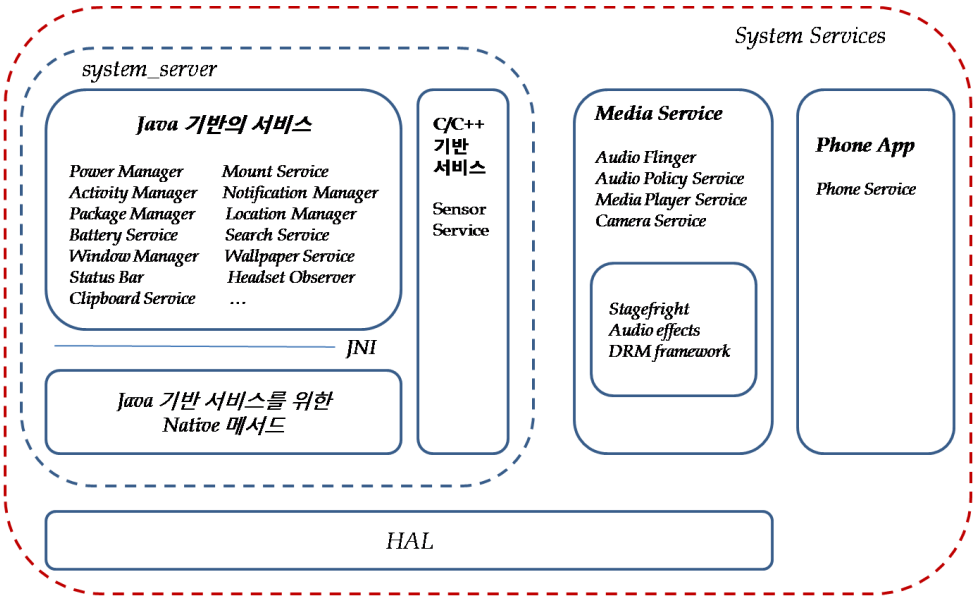

그림 1-30 System Service 정리

다음 두 그림은 동영상 재생과 연관이 있는 mediaserver(MediaPlayer -> stagefright, audioflinger)와 화면을 출력을 담당하는 surfaceflinger(서피스 플링어)를 함께 표현하고 있다.

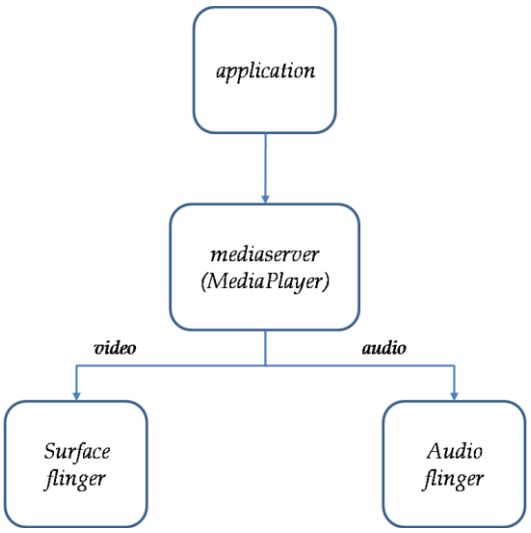

그림 1-31 mediaserver와 surfaceflinger(1)

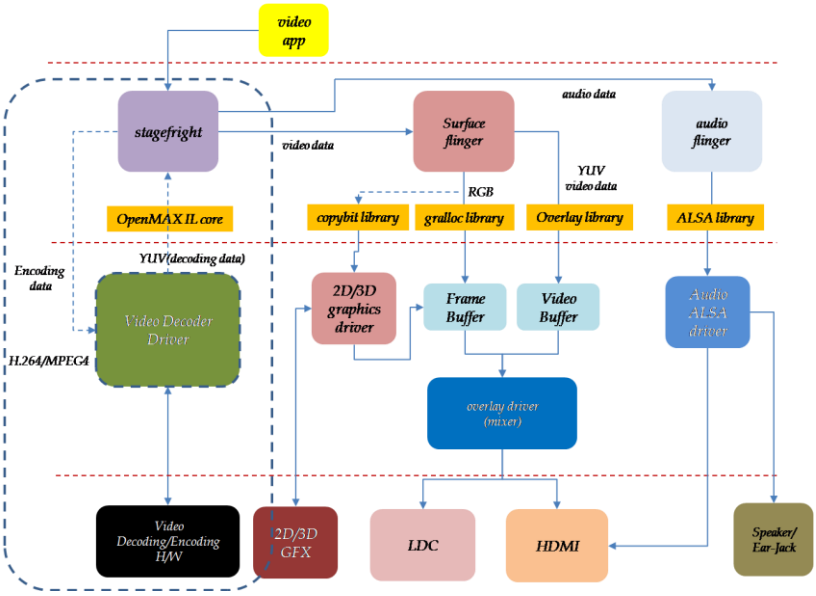

그림 1-32 mediaserver와 surfaceflinger(2)

2.9 Activity Manager와 Android Applications

안드로이드 부팅의 마지막 단계로, system_server 내의 Activity Manager로부터 다양한 application 이 어떻게 구동되는지를 살펴보기로 하자.

<Phone 서비스, Launcher 및 각종 응용 프로그램 구동 절차>

1) system_server의 InputMethod Manager가 <android.view.InputMethod> intent를 전송한다.

- com.android.inputmthod app이 구동된다.

2) 안드로이드 app 중, 아래 조건을 만족하는 경우는, Activity Manager에 의해 자동으로 구동된다.

- manifest 파일의 <application> element에 android:persistent="true" 속성을 가지고 있는 경우
- com.android.system.ui, com.android.phone이 이 경우에 해당하여, 자동으로 실행됨.

3) Activity Manager가 Intent.CATEGORY.HOME intent를 전송한다.

- com.android.launcher가 구동되어, 사용자로 부터의 명령을 기다린다.

4) 마지막으로 Activity Manager는 부팅의 끝을 알리는 Intent.BOOT_COMPLETED intent를 broadcast한다.

- Media Provider, Calendar Provider, Mms app, Email app 등이 구동됨.
- App Widget Service의 경우도 이 인텐트를 수신한 후, 다시 Intent.APPWIDGET_UPDATE intent를 broadcast하여 모든 widget app 들을 구동시킨다.

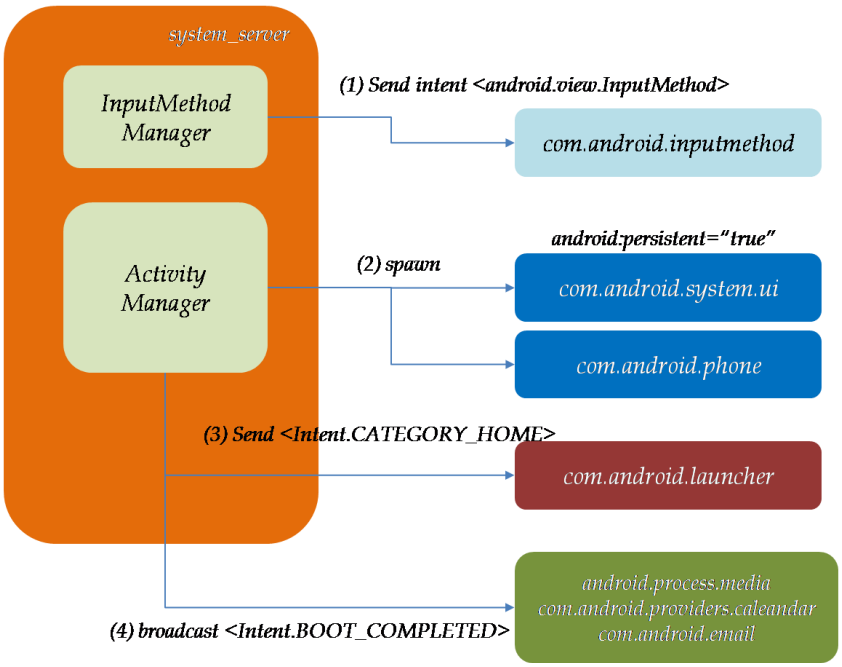

그림 1-33 Persistent App과 Launcher Application의 구동

지금까지 안드로이드 부팅 순서를 살펴 봄으로써, 간략하게 나마 안드로이드의 내부를 들여다 보았다. 아쉽게도 본 서는, 안드로이드 커널에 초점을 맞추고 있는 탓에, 더 이상 안드로이드 프레임워크의 깊이 있는 내용을 소개하지 못하는 것이 유감스럽다. 혹 독자 중에 관련하여 보다 추가 지식을 원하는 분들은 참고문헌 [3-7]을 적극 추천 드린다.

3. 안드로이드 빌드 시스템 이해

이 장에서는 안드로이드 소스를 내려 받아 컴파일을 하는 과정과, 안드로이드 고유의 빌드 시스템을 이해하기 위한 내용을 담아 보았다. 또한 커널 개발자가 알아야 할 만한 몇 가지 사항을 소개하였다.

3.1 안드로이드 소스 다운로드

지금부터는 Qualcomm Snapdragon용 안드로이드 소스코드를 www.codeaurora.org로부터 다운로드 한 후, 이를 빌드하는 과정을 소개하고자 한다. 본 서에서는 편의상 Code Aurora Forum의 오픈 소스를 기반으로 설명을 진행하지만, AOSP(Android Open Source Project), Samsung Exynos 소스코드, TI OMAP 소스코드 등을 다운 받아 사용하는 것도 전혀 문제가 될 것은 없다. 필자의 개인적인 생각으로는 안드로이드 오리지널 소스 코드(AOSP)를 받는 것도 나름 의미가 있겠지만, 실제로 필드에서 많이 사용되고 있는 소스 코드를 다운로드 받아 분석해 보는 것이 더 좋지 않나 생각해 본다.

Code Aurora Forum 소스 코드 다운로드 절차

<다운로드 명령 요약>

$ repo init -u git://codeaurora.org/platform/manifest.git -b [branch] -m [manifest] --repo-url=git://codeaurora.org/tools/repo.git

$ repo sync

안드로이드 소스 코드를 다운로드하기 위해서는 아래의 branch명과 manifest xml 파일 중 원하는 값을 선택/지정해야 한다.

표 1-1 Qualcomm Release 테이블

Releases	
Branch	**Targets**
release	msm7627a, msm7630, msm8660, msm8660_csfb, msm8960, apq8064, mpq8064, msm8930, msm8625, msm8974
jb	msm7627a, msm7630, msm8660, msm8660_csfb, msm8960, apq8064, mpq8064, msm8930, msm8625
jb_rel	apq8064
ics	msm7627a, msm7630, msm8660, msm8660_csfb, msm8960, apq8064, mpq8064, msm8930
ics_rb	msm8960
ics_rb1	apq8064
ics_rb2	msm8625
ics_rb3	msm8930
ics_rb4	msm8625
ics_rb6	msm8960
ics_rb7	mpq8064

표 1-2 Qualcomm Build ID 테이블

Date	Tag / Build ID	Chipset	Manifest	Android Version
July 19, 2013	A8064AAAAANLYA16103504	apq8064	A8064AAAAANLYA16103504.xml	04.02.02
July 18, 2013	A8064AAAAANLYA25040424	apq8064	A8064AAAAANLYA25040424.xml	04.02.02
July 17, 2013	M8974AAAAANLYA31050129	msm8974	M8974AAAAANLYA31050129.xml	04.02.02
July 17, 2013	A8064AAAAANLYA25136001	apq8064	A8064AAAAANLYA25136001.xml	04.02.02
July 16, 2013	M8974AAAAANLYA31050126	msm8974	M8974AAAAANLYA31050126.xml	04.02.02
July 16, 2013	M8974AAAAANLYA31050128	msm8974	M8974AAAAANLYA31050128.xml	04.02.02
July 16, 2013	M8960AAAAAYA20041014	msm8960	M8960AAAAAYA20041014.xml	04.01.01

July 16, 2013	M8930AAAAANLYA25138001	msm8930	M8930AAAAANLYA25138001.xml	04.02.02
July 16, 2013	M8930AAAAANLYA25040414	msm8930	M8930AAAAANLYA25040414.xml	04.02.02
July 16, 2013	M8626AAAAANLYA321120006	msm8226	M8626AAAAANLYA321120006.xml	04.02.02
July 12, 2013	M8974AAAAANLYA32123006	msm8974	M8974AAAAANLYA32123006.xml	04.02.02
July 12, 2013	M8974AAAAANLYA31050125	msm8974	M8974AAAAANLYA31050125.xml	04.02.02
July 12, 2013	M8930AAAAANLYA255092066	msm8930	M8930AAAAANLYA255092066.xml	04.02.02
July 09, 2013	M8974AAAAANLYA31050124	msm8974	M8974AAAAANLYA31050124.xml	04.02.02
July 09, 2013	M8974AAAAANLYA311098024	msm8974	M8974AAAAANLYA311098024.xml	04.02.02
July 05, 2013	M8626AAAAANLYA321101018	msm8226	M8626AAAAANLYA321101018.xml	04.02.02
July 05, 2013	M8610AAAAANLYA3211010302	msm8610	M8610AAAAANLYA3211010302.xml	04.02.02
July 04, 2013	M8930AAAAANLYA25040401	msm8930	M8930AAAAANLYA25040401.xml	04.02.02
July 04, 2013	A8064AAAAANLYA16103503	apq8064	A8064AAAAANLYA16103503.xml	04.02.02
July 04, 2013	M8930AAAAANLYA25040379	msm8930	M8930AAAAANLYA25040379.xml	04.02.02

Branch 명과 manifest 파일 정보 관련하여 보다 자세한 사항은 아래 site를 참조하기 바란다.

https://www.codeaurora.org/xwiki/bin/QAEP/release

아래 예는 branch를 "release"로 하고, manifest 값으로 "M8960AAAAAYA20041014.xml"를 선택한 상태에서 downloading하는 과정을 보여준다.

<안드로이드 소스 다운로드 하기>

$ mkdir –p ~/CAF/bin

- repo binary file을 복사해 둘 디렉토리 생성
- repo는 Google이 git을 기반하여 만든 소스 관리 도구임. [참고] repo이외에도 Google은 Gerrit이라는 웹 기반의 코드 리뷰 시스템을 도입해서 사용하고 있음.

$ curl https://dl-ssl.google.com/dl/googlesource/git-repo/repo > ~/CAF/bin/repo

- Curl program을 사용하여 repo binary download 시작

$ export PATH= ~/CAF/bin:$PATH

- repo의 실행 패스 설정

$ repo init -u git://codeaurora.org/platform/manifest.git -b ***release***
-m ***M8960AAAAAYA20041014.xml*** *--repo-url=git://codeaurora.org/tools/repo.git*

- 실제 소스 다운로드를 받기 위한 준비 작업

$ repo sync -j4

- 실제 소스 코드 다운로드 시작함.
- 모든 소스 코드를 내려 받는데 상당한 시간이 걸리므로, 인내심을 가질 필요가 있겠다.
- -j 다음의 숫자는 make –J4의 경우와 마찬가지로 CPU의 성능에 따라 적당한 값을 선택해야 함.
- [참고] 만일 소스코드를 받다가 중간에 연결이 끊긴다면, 다시 repo sync –j4를 해주면 됨.

다 내려 받은 안드로이드 소스 코드(Jelly Bean 4.2) 목록을 정리해 보면 다음과 같다.

```
$ ls -l
-r--r--r--   1 chyi chyi    87  7월 17 12:21 Makefile   - 대표 Makefile(build/core/main.mk 호출)
drwxrwxr-x   3 chyi chyi 4096  7월 17 12:21 abi   - C++ runtime type 정보
drwxrwxr-x  10 chyi chyi 4096  7월 17 12:21 bionic   - C library
drwxrwxr-x   5 chyi chyi 4096  7월 17 12:21 bootable   - LK bootloader, OTA, recovery 등
drwxrwxr-x   7 chyi chyi 4096  7월 20 21:15 build   - build system
-r-xr-xr-x   1 chyi chyi 5775  7월 17 12:22 build.sh   - build script
drwxrwxr-x  11 chyi chyi 4096  7월 17 12:21 cts   - CTS(Compatibility Test Suite) 관련
drwxrwxr-x  18 chyi chyi 4096  7월 17 12:21 dalvik   - Dalvik VM code
drwxrwxr-x  18 chyi chyi 4096  7월 17 12:22 development   - Development 도구
drwxrwxr-x  10 chyi chyi 4096  7월 17 12:22 device   - 보드(디바이스) 별 파일(porting 시작점)
drwxrwxr-x   3 chyi chyi 4096  7월 17 12:22 docs   - http://source.android.com 내용
drwxrwxr-x 171 chyi chyi 4096  7월 17 12:25 external   - external open source projects 모음
drwxrwxr-x  14 chyi chyi 4096  7월 17 12:27 frameworks   - 안드로이드 프레임워크
drwxrwxr-x  10 chyi chyi 4096  7월 17 12:27 gdk   - GDK(Graphics Development Kit) 관련
drwxrwxr-x  10 chyi chyi 4096  7월 17 12:27 hardware   - HAL 및 하드웨어 관련 라이브러리
drwxrwxr-x  24 chyi chyi 4096  7월 26 13:46 kernel   - 리눅스 커널
drwxrwxr-x  11 chyi chyi 4096  7월 17 12:28 libcore   - Apache Harmony(core library)
drwxrwxr-x   4 chyi chyi 4096  7월 17 12:28 libnativehelper   - JNI 사용을 위한 helper function
drwxrwxr-x   8 chyi chyi 4096  7월 17 12:28 ndk   - NDK(Native Development Kit) 관련
drwxrwxr-x   8 chyi chyi 4096  7월 17 12:29 packages   - 안드로이드 기본 제공 apps, providers, IME 등
drwxrwxr-x   5 chyi chyi 4096  7월 17 12:29 pdk   - PDK(Platform Development Kit) 관련
drwxrwxr-x  12 chyi chyi 4096  7월 17 12:32 prebuilts   - 미리 빌드해둔 바이너리(toolchain도 포함)
drwxrwxr-x  51 chyi chyi 4096  7월 17 12:32 sdk   - SDK(Software Development Kit) 관련
drwxrwxr-x  10 chyi chyi 4096  7월 17 12:32 system   - Embedded android platform 관련
```

drwxrwxr-x	*4*	*chyi*	*chyi*	*4096*	*7월 17 12:32* ***tools*** *- 여러 IDE 도구*
drwxrwxr-x	*3*	*chyi*	*chyi*	*4096*	*7월 17 12:32* ***vendor*** *- 여기서는 qualcomm 전용 코드 위치함*

위의 안드로이드 소스 트리 중, 중요하다가 여겨지는 부분을 위주로 정리해 보면 다음과 같다.

<주요 디렉토리 정리>

1) device/, system/ 디렉토리

2) bootable/

3) kernel/

4) hardware/, vendor/

5) frameworks/, external/

6) packages/, 사용자 UI

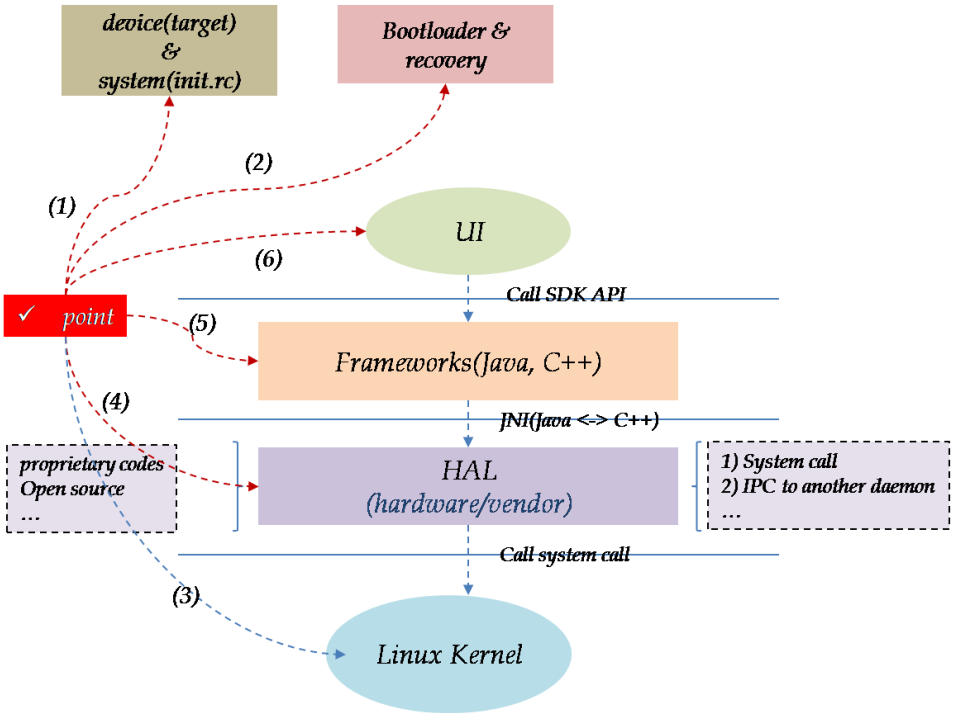

그림 1-34 안드로이드 소스 트리 개요

3.2 안드로이드 빌드(build) 하기

이 절에서는 3.1 절에서 다운로드한 안드로이드 소스코드를 빌드하는 방법과 한번 빌드한 내용을 clean하는 방법을 소개한다. 또한, 본서의 독자들이 자주 작업하게 될 bootloader와 kernel 단독 빌드 방법을 설명하고자 한다.

전체 빌드 하기

먼저 안드로이드 전체 빌드를 시도해 보자.

<안드로이드 소스 full-build 절차>

 # cd <android-source-tree>

 # source build/envsetup.sh

- 안드로이드 빌드를 위한 환경을 설정해 준다.
- . build/envsetup.sh 명령도 동일한 작업 수행함.

 # choosecombo 1 msm8960 eng

- Target 장치, build mode 등을 선택한다.
- Build 모드로는 eng, debug, userdebug 모드 등이 있다(자세한 내용은 3.3절 참조).
- choosecombo 명령 대신, lunch 명령을 사용해도 동일한 결과를 얻을 수 있다.

 # make –j4

- 실제 소스 코드를 빌드하기 시작한다.
- -j 다음의 숫자는 동시 compile을 수행하는 CPU의 성능에 따라 적당한 값을 사용해야 한다.

<build output>

- out/target/product/msm8960/emmc_appsboot.mbn - application bootloader image
- out/target/product/msm8960/boot.img – kernel/rootfs 통합 image
- out/target/product/msm8960/cache.img – cache image(/cache 디렉토리 구성)
- out/target/product/msm8960/persist.img – persist image
- out/target/product/msm8960/ramdisk.img – ramdisk image(rootfs 만으로 구성)
- out/target/product/msm8960/recovery.img – recovery kernel/rootfs 통합 image
- out/target/product/msm8960/system.img - /system 디렉토리 구성 image
  - system.img.ext4는 ext4 file system 형식의 이미지를 뜻함.
- out/target/product/msm8960/userdata.img - /data 디렉토리 구성 image
- …

빌드 내용 전체 삭제하기

앞서 전체 빌드한 내용을 삭제해 보도록 하자.

 # make clean

- 지금까지 빌드한 전체 내용을 삭제하는 명령으로, out/ 디렉토리 전체를 지우는 효과가 있다.
- main.mk에 정의되어 있다.

 # make installclean

- TARGET_PRODUCT, TARGET_BUILD_VARIANT, PRODUCT_LOCALES 등의 환경 값이 바뀔 경우(choosecombo, lunch) 자동으로 호출되며, out/ 디렉토리 전체를 지우는 것이 아니라 환경 값이 바뀜에 따라 재 빌드를 요하는 부분만을 삭제한다.
- cleanbuild.mk에 정의되어 있다.

지금까지 안드로이드를 전체 빌드하고, 삭제하는 방법에 관하여 알아 보았다. 전체 빌드는 빌드 서버의 성능에 따라서는 1시간 이상이 소요되기도 하는 매우 지루한 작업이다. 아래 내용은 특별히 커널 개발자들을 위하여 부트로더와 커널을 단독으로 빌드하는 방법을 소개해 보고자 한다.

부트로더 단독 빌드하기

LK 부트로더를 효과적으로 빌드하기 위해, 아래와 같은 전용 빌드 스크립트를 작성해 보았다. 다소 복잡한 듯 보이지만, 스크립트 전반부의 내용은 모두 환경 변수를 지정하는 것에 불구하며, 결국은 맨 마지막의 make 명령을 한번 실행해 주는 것이 전부임을 알 수 있다.

<build 하기>

 # source ./build_lk_bootloader.sh

<build_lk_bootloader.sh>

```
#!/bin/bash
. build/envsetup.sh
choosecombo 1 msm8960 eng

TARGET_PRODUCT=msm8960
TARGET_BUILD_VARIANT="TARGET_BUILD_VARIANT=eng"
SIGNED_KERNEL="SIGNED_KERNEL=0"
SOURCE_HOME_DIR=~/QCOM
ANDROID_HOME_DIR=$SOURCE_HOME_DIR/android
ANDROID_TARGET_OUT_DIR=$ANDROID_HOME_DIR/out/target/product/$TARGET_PRODUCT
ANDROID_HOST_OUT_DIR=$ANDROID_HOME_DIR/out/host/linux-x86/bin
CROSSTOOL_HOME_DIR=$ANDROID_HOME_DIR/prebuilt/linux-x86/toolchain/arm-eabi-4.4.3/bin
CROSS_TOOL=$CROSSTOOL_HOME_DIR/arm-eabi-

LK_HOME_DIR=$ANDROID_HOME_DIR/bootable/bootloader/lk
LK_OUT_DIR=$ANDROID_TARGET_OUT_DIR/obj/EMMC_BOOTLOADER_OBJ
```

mkdir -p ₩
$ANDROID_HOME_DIR/out/target/product/$TARGET_PRODUCT/obj/EMMC_BOOTLOADER_OBJ

make -C $LK_HOME_DIR TOOLCHAIN_PREFIX=$CROSS_TOOL BOOTLOADER_OUT=$LK_OUT_DIR $TARGET_PRODUCT EMMC_BOOT=1 $TARGET_BUILD_VARIANT $SIGNED_KERNEL

<결과 파일>
*out/target/product/msm8960/obj/EMMC_BOOTLOADER_OBJ/build-msm8960/**
*out/target/product/msm8960/**emmc_appsboot.mbn***

커널 단독 빌드하기

커널의 경우도 단독으로 빌드하는 절차를 아래 정리해 보았다. 역시 초반에 toolchain 경로를 자동으로 맞춰주기 위해 안드로이드 환경을 잡아주는 부분을 제외하면, 기존에 알고 있던 방법과 커다란 차이가 없음을 알 수 있다.

<build 하기>
 # . build/envsetup.sh
 # choosecombo 1 msm8960 eng

- 동일한 콘솔 창이라면, 위의 내용은 한차례만 수행해 주면 됨.

 # mkdir –p out/target/product/msm8960/obj/KERNEL_OBJ

- 커널 빌드 후, object 및 결과 파일이 위치하는 디렉토리를 생성해 주어야 함(디렉토리가 없는 경우).

 # make -C kernel O=../out/target/product/msm8960/obj/KERNEL_OBJ ₩
ARCH=arm CROSS_COMPILE=arm-eabi- msm8960_defconfig

- Config 파일을 지정한다.

 # make -C kernel O=../out/target/product/msm8960/obj/KERNEL_OBJ ₩
ARCH=arm CROSS_COMPILE=arm-eabi- menuconfig

- menuconfig로 config 내용을 target board에 맞게 조정한다.

 # make -C kernel O=../out/target/product/msm8960/obj/KERNEL_OBJ ₩
ARCH=arm CROSS_COMPILE=arm-eabi- zImage

- 커널 이미지를 생성하기 위해 빌드한다.

 # make -C kernel O=../out/target/product/msm8960/obj/KERNEL_OBJ ₩
ARCH=arm CROSS_COMPILE=arm-eabi- modules

- 커널 모듈을 생성하기 위해 빌드한다.

<결과 파일>

out/target/product/msm8960/obj/KERNEL_OBJ/.config

- 현재 커널에서 사용중인 config 내용

out/target/product/msm8960/obj/KERNEL_OBJ/arch/arm/boot/zImage

- zImage 파일

3.3 안드로이드 빌드 시스템 분석

이제부터는 뭔가 좀 다른 듯 보이는 안드로이드 빌드 시스템에 대해서 구체적으로 분석해 보기로 하겠다.

안드로이드 빌드 시스템은 각각의 하위 디렉토리에 Android.mk라는 새로운 형식의 파일을 만들어 두고, 빌드하기 직전에 이를 하나로 모아 거대한 Makefile을 만든 후 빌드하는 것이 특징이라 할 수 있다. Android.mk는 새로운 형식의 문법을 제공하고 있으나, 내부를 따라가 보면 궁극적으로는 GNU make의 문법과 동일한데, 이를 위해서 build/core 디렉토리 아래에 다양한 *.mk 파일을 배치하고 있으며, 이 안에서 새로운 문법을 정의하게 된다.

그림 1-35는 안드로이드 빌드 시스템을 하나로 통합하여 표현한 것으로, build/envsetup.sh 혹은 lunch 명령 등이 빌드의 시작점에 해당하며, 내부적으로는 다양한 build/core/*.mk 파일과 보드의 특성을 기술하는 AndroidProducts.mk, BoardConfig.mk 등이 핵심 내용을 이루고 있다. 빌드가 완료된 후에 생성되는 결과 파일은 out/ 디렉토리 아래에 생성된다.

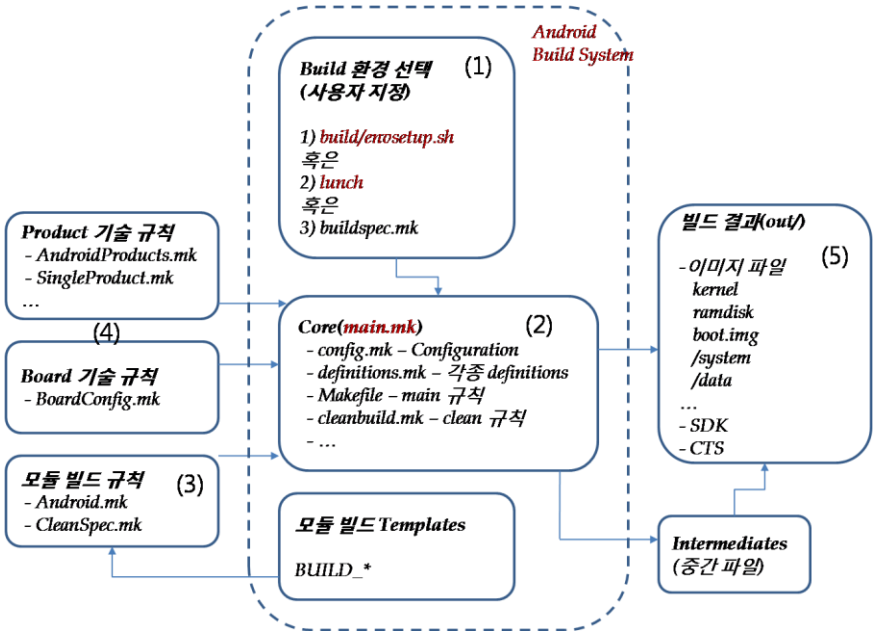

그림 1-35 Android 빌드 시스템

빌드 시작점 – envsetup.sh

앞서 이미, envsetup.sh를 사용하여 빌드하는 방법을 소개했었다. 그렇다면, 여기서는 build/envsetup.sh 명령어를 실행 한 후, 사용 가능한 명령어에 관하여 자세히 알아보기로 하자.

<단계 1 – envsetup.sh 실행하기>

 # . build/envsetup.sh

- toolchain path 등 개발 환경을 설정한다.
- 기타 아래의 몇 가지 유용한 명령어를 제공한다.

 # . build/envsetup.sh

 # hmm

lunch

- lunch <product name>-<build_variant>

tapas

- tapas [<App1><App2> ...] [arm|x86|mips] [eng|userdebug|user]

croot

- 안드로이드 빌드 트리의 최 상위 디렉토리로 이동해주는 명령

m

- 안드로이디 빌드 트리의 top에서 부터 make(build) 시작
- 모든 Android.mk 파일을 찾아서 하나로 만든 후, make를 돌림.

mm

- 현재 directory 아래의 모든 module을 build

mmm

- 파라미터로 지정한 directory 아래의 모든 module을 build
- 예) mmm test

cgrep

- 모든 C/C++ file에 대해 특정 패턴을 grep하고자 할 때 사용함.

jgrep

- 모든 java file에 대해 특정 패턴을 grep하고자 할 때 사용함.

resgrep

- 모든 res/*.xml 파일에 대해 특정 패턴을 grep하고자 할 때 사용함.

godir

- 특정 파일을 포함하는 디렉토리로 이동하는 명령

<단계 2 – choosecombo or lunch 메뉴 선택>

choosecombo

혹은

lunch

- emulator 혹은 여러 device 중, 바이너리를 upload할 대상을 선택한다.
- Build mode(release, debug)를 선택한다.
- Product model을 선택한다.
- 구동 mode(user, userdebug, engineer)를 선택한다.

위의 메뉴로 빌드 조건을 선택하게 되면, 아래 환경 변수 값이 내부적으로 설정되게 된다.

TARGET_PRODUCT

- device/ 디렉토리에 존재하는 각 vendor별 디바이스 목록 중, 원하는 device를 선택하면 이 변수로 값이 지정된다.

TARGET_BUILD_VARIANT

- eng | userdebug | user

TARGET_BUILD_TYPE

- release | debug

TARGET_TOOLS_PREFIX

- Default toolchain(prebuilts 디렉토리)을 사용하는 대신, 사용자 지정 toolchain을 사용하고자 할 경우 지정해 주게 됨(자주 사용하지는 않음).

OUT_DIR

- 빌드 결과물을 위치하는 디렉토리를 사용자가 특별히 지정하고 싶을 경우에 사용함. 이를 지정하지 않을 경우, out/ 디렉토리에 default로 결과물이 생성됨.

<단계 3 – make 명령 실행하기>

make -j4

- android/Makefile이 대상 파일임.
- -j 다음의 숫자 값은 빌드 서버의 CPU의 성능에 맞는 적당한 값을 입력하면 됨.

참고적으로 build/envsetup.sh 파일을 이용하지 않고, build/buildspec.mk.default 파일을 사용자가 원하는 대로 편집하여 사용할 수도 있는데 이를 아래에 정리해 보았다.

 # mv build/buildspec.mk.default build/buildspec.mk
 # vi build/buildspec.mk

- 원하는 형태로 편집

 # . build/buildspec.mk

 # make –j4

Build/core/*.mk 소개

build/core 디렉토리 아래에는 매우 많은 *.mk 파일이 있는데 이 중, main.mk 파일이 전체의 시작 지점이라고 볼 수 있다. 그림 1-36을 보면 알 수 있듯이, main.mk는 config.mk 파일을 읽어 들여 lunch나 choosecombo 등에서 사용하는 환경을 설정하고, definitions.mk 파일을 읽어 들여 Android.mk 문법의 기초를 정의하게 된다.

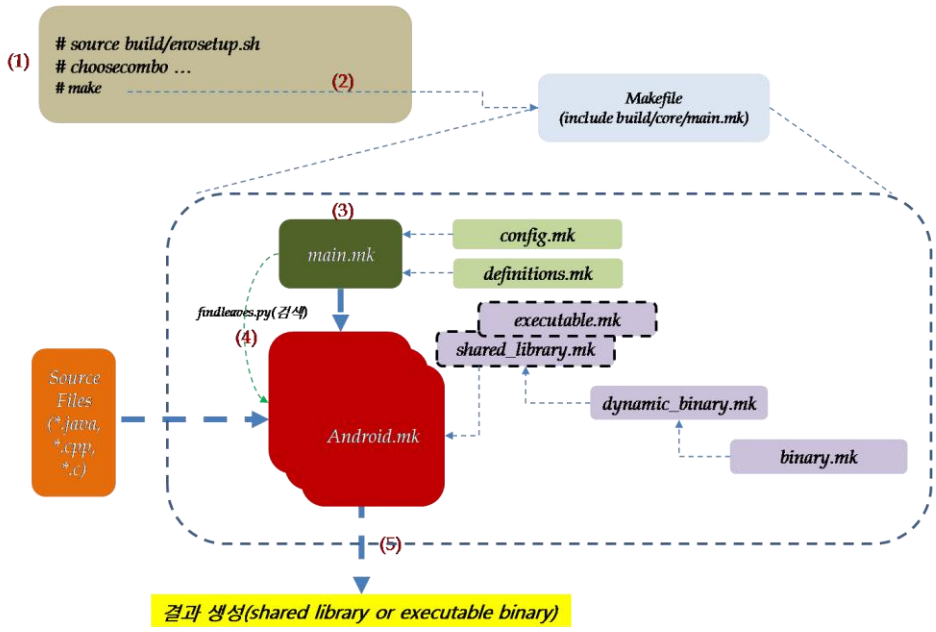

그림 1-36 Core 파일과 Android Makefile 개요

build/core 아래에 무슨 파일이 있는지 목록을 확인해 보기로 하자.

$ ls -l *.mk

-rw-rw-r-- 1 chyi chyi 24948 7월 17 12:21 base_rules.mk
-rw-rw-r-- 1 chyi chyi 28859 7월 17 12:21 binary.mk
-rw-rw-r-- 1 chyi chyi 820 7월 17 12:21 build_id.mk
-rw-rw-r-- 1 chyi chyi 8339 7월 17 12:21 cleanbuild.mk
-rw-rw-r-- 1 chyi chyi 3841 7월 17 12:21 cleanspec.mk
-rw-rw-r-- 1 chyi chyi 3783 7월 17 12:21 clear_vars.mk
-rw-rw-r-- 1 chyi chyi 18225 7월 17 12:21 config.mk
-rw-rw-r-- 1 chyi chyi 731 7월 17 12:21 copy_headers.mk
-rw-rw-r-- 1 chyi chyi 74445 7월 17 12:21 definitions.mk
-rw-rw-r-- 1 chyi chyi 1882 7월 17 12:21 device.mk

-rw-rw-r-- 1 chyi chyi 3397 7월 17 12:21 dex_preopt.mk
-rw-rw-r-- 1 chyi chyi 2261 7월 17 12:21 distdir.mk
-rw-rw-r-- 1 chyi chyi 8643 7월 17 12:21 droiddoc.mk
-rw-rw-r-- 1 chyi chyi 3752 7월 17 12:21 dumpvar.mk
-rw-rw-r-- 1 chyi chyi 5176 7월 17 12:21 dynamic_binary.mk
-rw-rw-r-- 1 chyi chyi 8892 7월 17 12:21 envsetup.mk
-rw-rw-r-- 1 chyi chyi 2886 7월 17 12:21 executable.mk
-rw-rw-r-- 1 chyi chyi 1563 7월 17 12:21 help.mk
-rw-rw-r-- 1 chyi chyi 556 7월 17 12:21 host_executable.mk
-rw-rw-r-- 1 chyi chyi 3435 7월 17 12:21 host_java_library.mk
-rw-rw-r-- 1 chyi chyi 428 7월 17 12:21 host_native_test.mk
-rw-rw-r-- 1 chyi chyi 666 7월 17 12:21 host_prebuilt.mk
-rw-rw-r-- 1 chyi chyi 1059 7월 17 12:21 host_shared_library.mk
-rw-rw-r-- 1 chyi chyi 795 7월 17 12:21 host_static_library.mk
-rw-rw-r-- 1 chyi chyi 17807 7월 17 12:21 java.mk
-rw-rw-r-- 1 chyi chyi 3719 7월 17 12:21 java_library.mk
-rw-rw-r-- 1 chyi chyi 2035 7월 17 12:21 legacy_prebuilts.mk
-rw-rw-r-- 1 chyi chyi 3651 7월 17 12:21 llvm_config.mk
-rw-rw-r-- 1 chyi chyi 31636 7월 17 12:21 ***main.mk***
-rw-rw-r-- 1 chyi chyi 3830 7월 17 12:21 multi_prebuilt.mk
-rw-rw-r-- 1 chyi chyi 621 7월 17 12:21 native_test.mk
-rw-rw-r-- 1 chyi chyi 7548 7월 17 12:21 node_fns.mk
-rw-rw-r-- 1 chyi chyi 2998 7월 17 12:21 notice_files.mk
-rw-rw-r-- 1 chyi chyi 16207 7월 17 12:21 package.mk
-rw-rw-r-- 1 chyi chyi 4077 7월 17 12:21 pathmap.mk
-rw-rw-r-- 1 chyi chyi 6456 7월 17 12:21 pdk_config.mk
-rw-rw-r-- 1 chyi chyi 299 7월 17 12:21 phony_package.mk
-rw-rw-r-- 1 chyi chyi 2472 7월 17 12:21 post_clean.mk
-rw-rw-r-- 1 chyi chyi 5803 7월 17 12:21 prebuilt.mk
-rw-rw-r-- 1 chyi chyi 8226 7월 17 12:21 product.mk
-rw-rw-r-- 1 chyi chyi 13888 7월 17 12:21 product_config.mk
-rw-rw-r-- 1 chyi chyi 939 7월 17 12:21 raw_executable.mk
-rw-rw-r-- 1 chyi chyi 66 7월 17 12:21 raw_static_library.mk
-rw-rw-r-- 1 chyi chyi 87 7월 17 12:21 root.mk
-rw-rw-r-- 1 chyi chyi 2920 7월 17 12:21 shared_library.mk
-rw-rw-r-- 1 chyi chyi 4280 7월 17 12:21 static_java_library.mk
-rw-rw-r-- 1 chyi chyi 1264 7월 17 12:21 static_library.mk
-rw-rw-r-- 1 chyi chyi 3869 7월 17 12:21 version_defaults.mk

---

build/core 디렉토리에 존재하는 주요 mk 파일의 의미를 간략히 정리해 보면 다음과 같다.

---

1) executable.mk

- BUILD_EXECUTABLE
- Target용 실행 파일(binary 파일) 생성

2) host_executable.mk

- BUILD_HOST_EXECUTABLE
- Host용 실행 파일 생성

3) raw_executable.mk

- BUILD_RAW_EXECUTABLE
- Target용 실행 파일 생성(Target binaries that run on bare metal)

4) java_library.mk

- BUILD_JAVA_LIBRARY
- Target용 Java 라이브러리 파일 생성

5) static_java_library.mk

- BUILD_STATIC_JAVA_LIBRARY
- Target용 정적 Java 라이브러리 파일 생성

6) host_java_library.mk

- BUILD_HOST_JAVA_LIBRARY
- Host용 Java 라이브러리 파일 생성

7) shared_library.mk

- BUILD_SHARED_LIBRARY
- Target용 동적 라이브러리 파일 생성

8) static_library.mk

- BUILD_STATIC_LIBRARY
- Target 용 정적 라이브러리 파일 생성

9) host_shared_library.mk

- BUILD_HOST_SHARED_LIBRARY
- Host용 동적 라이브러리 파일 생성

10) host_static_library.mk

- BUILD_HOST_STATIC_LIBRARY

- Host용 정적 라이브러리 파일 생성

11) raw_static_library.mk

- BUILD_RAW_STATIC_LIBRARY
- Target 용 정적 라이브러리 생성(Target static libraries that run on bare metal)

12) prebuilt.mk

- BUILD_PREBUILT
- 미리 빌드해 둔 파일(Target용 파일) 복사

13) host_prebuilt.mk

- BUILD_HOST_PREBUILT
- 미리 빌드해 둔 파일(Host용 파일) 복사

14) multi_prebuilt.mk

- BUILD_MULTI_PREBUILT

15) package.mk

- BUILD_PACKAGE
- *.apk 파일(built-in apps 파일)

16) key_char_map.mk

- BUILD_KEY_CHAR_MAP
- Device character maps

...

Android.mk 파일 분석

앞서 언급한 바와 같이 Android.mk 파일은 새로운 문법을 정의하고 있는데, 이를 구성하는 주요 구문의 의미를 정리해 보기로 하자.

*1) **include $(CLEAR_VARS)***

- build 관련 local 변수(아래 내용들)를 모두 clear해 줌.
- build/core/clear_vars.mk 파일이 include될 것임.
- 이 파일을 보면, Android.mk file에서 사용 가능한 local 변수를 확인할 수 있음.

2) LOCAL 변수

*2-1) **LOCAL_MODULE***

- build하려는 module의 이름(결과 파일명)

2-2) ***LOCAL_SRC_FILES***

- build하려는 source 파일들

2-3) ***LOCAL_STATIC_LIBRARIES***

- 이 module(결과물)에 static하게 link하는 libraries

2-4) ***LOCAL_SHARED_LIBRARIES***

- 이 module에 link되는 shared libraries

2-5) ***LOCAL_C_INCLUDES***

- include file을 위한 path 지정(가령: $KERNEL_HEADERS)

2-6) ***LOCAL_CFLAGS***

- compiler에게 전달하는 추가 CFLAGS 지정

2-7) ***LOCAL_LDFLAGS***

- linker에게 전달하는 추가 LDFLAGS 지정

3) Include BUILD rules

3-1) ***include $(BUILD_EXECUTABLE)***

- 실행파일을 build하는 rule이 추가됨(여기에서 build함 – 기존 Makefile format)
- build/core/executable.mk 파일이 include될 것임.

3-2) ***include $(BUILD_SHARED_LIBRARY)***

- shared library를 build하는 rule이 추가됨.

3-3) ***include $(BUILD_STATIC_LIBRARY)***

- static library를 build하는 rule이 추가됨.

3-4) ***include $(BUILD_PREBUILT)***

- prebuilt file을 복사하는 rule이 추가됨.

<Android.mk 파일 예제>

```
#libmylibrary.so를 만드는 예(shared library)
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
#Name of target to build
LOCAL_MODULE:= libmylibrary
#Source files to compile
LOCAL_SRC_FILES:= mysrcfile.c mysothersrcfile.c
#The shared libraries to link against
LOCAL_SHARED_LIBRARIES := libcutils
#No special headers needed
LOCAL_C_INCLUDES +=
#Prelink this library, also need to add it to the prelink map
LOCAL_PRELINK_MODULE := true
include $(BUILD_SHARED_LIBRARY)
```

```
#mycmd 실행파일을 만드는 예
#Clear variables and build the executable
include $(CLEAR_VARS)
LOCAL_MODULE:= mycmd
LOCAL_SRC_FILES:= mycmdsrc.c
include $(BUILD_EXECUTABLE)
```

보드 설정 파일 분석

다음으로 확인해 볼 부분은 특정 보드(플랫폼) 관련 환경 설정 부분이다. 이는 device/qcom/msm8960 아래에 있는 AndroidProduct.mk, msm8960.mk, BoardConfig.mk 파일 등이 이 것과 상호 연관이 있다. 이 중 AndroidProduct.mk가 include하는 msm8960.mk 파일과 커널 command line 값 지정, 파티션 정보 설정 등 보드 관련 설정을 담당하는 BoardConfig.mk 파일을 살펴 보도록 하자.

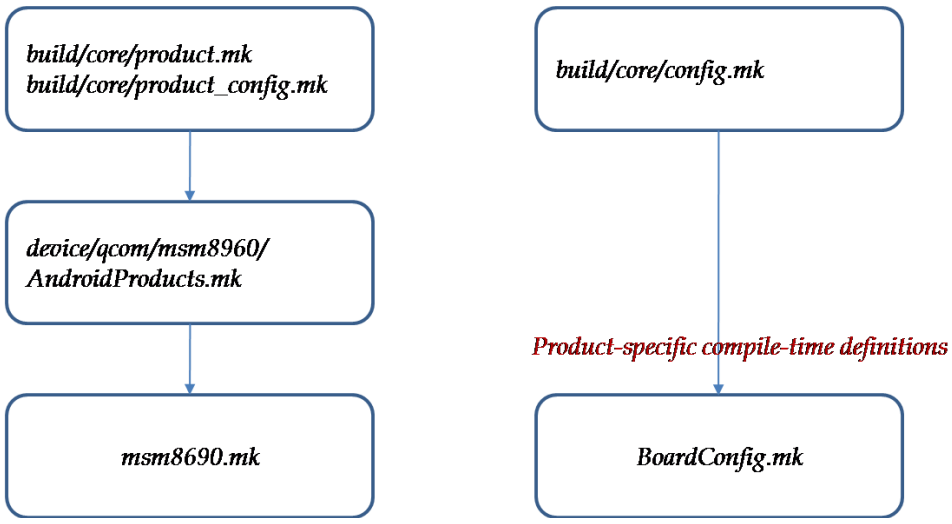

그림 1-37 Target Board 관련 설정 [다시 그려야 함]

<AndroidProducts.mk – msm8960.mk 파일>

```
DEVICE_PACKAGE_OVERLAYS := device/qcom/msm8960/overlay
    # DEVICE_PACKAGE_OVERLAYS – default 패키지를 device 전용 패키지로 대체(overlay)하도록 해줌.

$(call inherit-product, device/qcom/common/common.mk)

PRODUCT_NAME := msm8960
```

```
    # choosecombo 혹은 lunch에서 입력한 내용에 해당함.

PRODUCT_DEVICE := msm8960
    # /device/qcom/msm8960에 해당하는 내용임.

PRODUCT_COPY_FILES += system/bluetooth/data/main.le.conf:system/etc/bluetooth/main.conf
    #target 파일 시스템에 추가하고 싶은 파일을 기술(: 이전의 파일을 : 이후의 파일로 복사)

PRODUCT_PACKAGES += Galaxy4
PRODUCT_PACKAGES += HoloSpiralWallpaper
PRODUCT_PACKAGES += MagicSmokeWallpapers
PRODUCT_PACKAGES += NoiseField
PRODUCT_PACKAGES += PhaseBeam
    #추가로 추가하고 싶은 패키지를 지정해 줌.
...
```

<BoardConfig.mk 파일>

```
# 보드 설정과 관련된 내용으로 수정 시, 다시 전체 빌드를 해 주어야 함.
# 필요시 새로운 항목을 추가할 수도 있겠음.

ifeq ($(TARGET_ARCH),)
TARGET_ARCH := arm
endif

BOARD_USES_GENERIC_AUDIO := true
USE_CAMERA_STUB := true

-include vendor/qcom/proprietary/common/msm8960/BoardConfigVendor.mk

TARGET_USE_HDMI_AS_PRIMARY := false

ifeq ($(TARGET_USE_HDMI_AS_PRIMARY),true)
    TARGET_HAVE_HDMI_OUT := false
else
    TARGET_HAVE_HDMI_OUT := true
endif # TARGET_USE_HDMI_AS_PRIMARY
#TODO: Fix-me: Setting TARGET_HAVE_HDMI_OUT to false
# to get rid of compilation error.
TARGET_HAVE_HDMI_OUT := false
```

```
TARGET_USES_OVERLAY := true
TARGET_NO_BOOTLOADER := false
TARGET_NO_KERNEL := false
TARGET_NO_RADIOIMAGE := true
TARGET_NO_RPC := true

TARGET_GLOBAL_CFLAGS += -mfpu=neon -mfloat-abi=softfp
TARGET_GLOBAL_CPPFLAGS += -mfpu=neon -mfloat-abi=softfp
TARGET_CPU_ABI  := armeabi-v7a
TARGET_CPU_ABI2 := armeabi
TARGET_ARCH_VARIANT := armv7-a-neon
TARGET_CPU_SMP := true
ARCH_ARM_HAVE_TLS_REGISTER := true

TARGET_HARDWARE_3D := false
TARGET_BOARD_PLATFORM := msm8960
TARGET_BOOTLOADER_BOARD_NAME := MSM8960

# eMMC(혹은 NAND flash) 내의 정보(kernel base 주소, page size, ramdisk offset 등) 지정
BOARD_KERNEL_BASE     := 0x80200000
BOARD_KERNEL_PAGESIZE := 2048
BOARD_RAMDISK_OFFSET := 0x02000000
BOARD_MKBOOTIMG_ARGS := --ramdisk_offset 0x02000000

# target file system type(ext4) 지정
TARGET_USERIMAGES_USE_EXT4 := true
BOARD_CACHEIMAGE_FILE_SYSTEM_TYPE := ext4
BOARD_PERSISTIMAGE_FILE_SYSTEM_TYPE := ext4

#kernel command line option 지정
BOARD_KERNEL_CMDLINE     := console=ttyHSL0,115200,n8 androidboot.hardware=qcom
user_debug=31 msm_rtb.filter=0x3F ehci-hcd.park=3 maxcpus=2
BOARD_EGL_CFG := device/qcom/$(TARGET_PRODUCT)/egl.cfg

#eMMC(혹은 NAND flash) 파티션 별 size 지정(실제로 생성되는 이미지 크기를 고려해 조정해야
함)
BOARD_BOOTIMAGE_PARTITION_SIZE := 23068672 # 22M
BOARD_RECOVERYIMAGE_PARTITION_SIZE := 23068672 # 22M
# 실제로 system 파티션의 크기는 아래 내용 보다 훨씬 커지게 됨. 따라서 이에 맞춰 조정해 주
```

어야 함.
BOARD_SYSTEMIMAGE_PARTITION_SIZE := 536870912
data 파티션의 크기가 작을 경우, android app이 구동되는데 문제가 있을 수 있으니, 적절한 크기로 조정해 주어야 함.
BOARD_USERDATAIMAGE_PARTITION_SIZE := 10737418240

BOARD_CACHEIMAGE_PARTITION_SIZE := 33554432
BOARD_PERSISTIMAGE_PARTITION_SIZE := 5242880
BOARD_TOMBSTONESIMAGE_PARTITION_SIZE := 268435456
BOARD_FLASH_BLOCK_SIZE := 131072 # (BOARD_KERNEL_PAGESIZE * 64)

Use signed boot and recovery image
TARGET_BOOTIMG_SIGNED := true

TARGET_USE_KRAIT_BIONIC_OPTIMIZATION := true
TARGET_USE_KRAIT_PLD_SET := true
TARGET_KRAIT_BIONIC_PLDOFFS := 10
TARGET_KRAIT_BIONIC_PLDTHRESH := 10
TARGET_KRAIT_BIONIC_BBTHRESH := 64
TARGET_KRAIT_BIONIC_PLDSIZE := 64

HAVE_CYTTSP_FW_UPGRADE := true
HAVE_MXT_FW_UPGRADE := true
HAVE_MXT_CFG := true
HAVE_SYNAPTICS_I2C_RMI4_FW_UPGARDE := true

Add NON-HLOS files for ota upgrade
ADD_RADIO_FILES ?= false
TARGET_USES_QCOM_BSP := true

Added to indicate that protobuf-c is supported in this build
PROTOBUF_SUPPORTED := true

Add building support AR8151 ALX ethernet driver
BOARD_HAS_ATH_ETH_ALX := true

#TARGET_RECOVERY_PIXEL_FORMAT := "RGBX_8888"
#TARGET_RECOVERY_UI_LIB := librecovery_ui_qcom

```
TARGET_USES_ION := true
TARGET_ADDITIONAL_BOOTCLASSPATH := qcmediaplayer:WfdCommon
TARGET_ENABLE_QC_AV_ENHANCEMENTS := true
```

부트로더/커널 빌드 파일 분석

Qualcomm의 경우는 안드로이드 전체 빌드 과정에 부트로더와 커널 빌드를 기본으로 포함시키고 있다.

1) build/target/board/Android.mk 파일의 아래 include 문장을 통해 아래 파일이 include 되며,

- device/qcom/msm8960/AndroidBoard.mk
- *-include $(TARGET_DEVICE_DIR)/AndroidBoard.mk*

2) 다시 이 파일에서 부트로더를 build 해주는 AndroidBoot.mk와 커널을 build 해주는 AndroidKernel.mk 파일이 include되게 된다.

- AndroidBoot.mk
- AndroidKernel.mk

그림 1-38 부트로더 빌드 - AndroidBoot.mk 파일

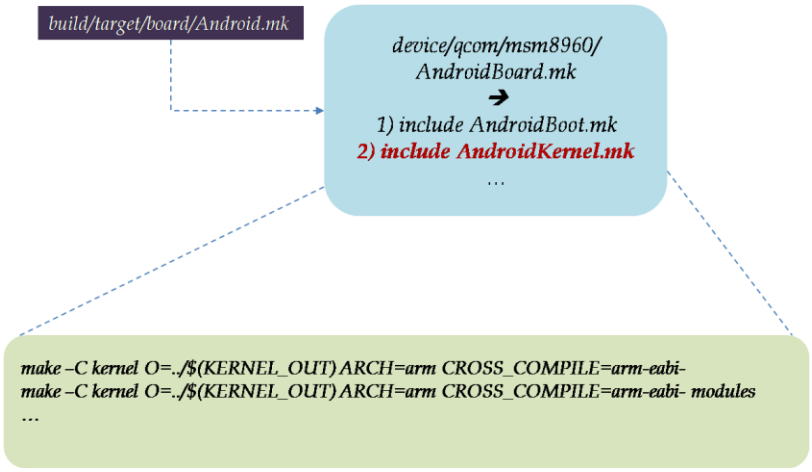

그림 1-39 Kernel 빌드 - AndroidKernel.mk 파일

부트로더와 커널을 단독으로 빌드하는 과정을 이미 소개했으므로, 여기서는 둘의 빌드 과정을 다시 소개하지는 않겠다.

3.4 빌드 결과물 분석

이제부터는 빌드의 마지막이라고 할 수 있는 결과물 분석에 들어가 보자. 빌드 결과로 생성되는 여러 중간 파일은 일단 무시하고, 최종 이미지들이 들어 있는 아래 디렉토리로 이동해 보자.

<빌드 결과물 일부 발췌>

out/target/product/msm8960/

cache.img.ext4	*- cache/ 이미지(ext4 file system 형태)*
boot.img	*- kernel + rootfs 이미지*
data/	*- userdata.img 파일을 만드는 원본 디렉토리*
kernel	*- zImage와 동일함.*
obj/	
persist.img.ext4	*- persistent image(ext4 file system 형태)*
ramdisk.img	*- rootfs로 구성된 ramdisk 이미지*
recovery.img	*- recovery kernel + rootfs(init.rc 내용이 간소화되어 있음) 이미지*
root/	*- rootfs를 구성하는 원본 디렉토리*
symbols/	*- debugging 정보가 포함된 system 디렉토리*
system/	*- system.img 파일을 만드는 원본 디렉토리*
system.img.ext4	*- system 이미지 파일(ext4 file system 형태)*
tombstones.img.ext4	*- tombstone 이미지 파일(ext4 file system 형태)*
userdata.img.ext4	*- userdata 이미지 파일((ext4 file system 형태)*
…	

boot.img 해부

이 절에서는 안드로이드를 전체 빌드하여 얻은 결과 파일 중, kernel과 rootfs로 구성된 boot.img를 먼저 해부해 보도록 하겠다.

<boot.img 파일의 구성>

1) 1 page로 이루어진 boot header

- "ANDROID!" 문자열이 맨 앞 부분에 저장되어 있음.
  - 부트로더에서 부팅 시, 이 문자열이 없으면, 부팅을 중단해 버린다.
- Kernel command line이 저장됨

2) kernel image

- gzip 압축 파일임
- zImage

3) ramdisk rootfs image

- gzip 압축 파일임(예: ramdisk.gz)

4) 기타 추가 사항이 있을 경우에 여기에 위치하게 됨.

- 보통 boot.img를 타인이 변경하지 못하도록 boot.img에 대한 hash 값을 계산하여 이 영역에 추가하게 됨.

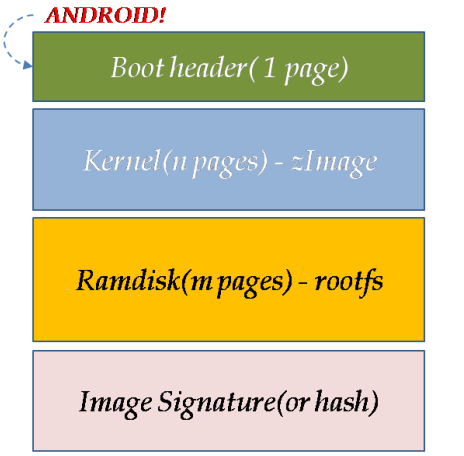

그림 1-40 boot.img의 구성

안드로이드에서 boot.img를 만들기 위해서는 mkbootimg(out/host 이하 디렉토리 참조) 명령이 사용된다.

<boot.img 파일 생성>

 # mkbootimg ₩

--cmdline "console=ttyHSL0,115200,n8 androidboot.hardware=qcom" ₩

*--kernel **zImage** ₩*

*--ramdisk **ramdisk.gz** ₩*

–-base 0x80200000

*-o **boot.img***

[참고 사항]

*1) cmdline은 BoardConfig.mk 파일의 **BOARD_KERNEL_CMDLINE** 내용을 참조하여 작성*

2) kernel과 ramdisk는 gzip 압축 파일이어야 함.

*3) –base 주소(eMMC 혹은 NAND flash의 커널이 위치한 번지)는 BoardConfig.mk 파일의 **BOARD_KERNEL_BASE** 내용을 참조하여 작성*

mkbootimg를 써서 boot.img를 만들었다면, 반대로 boot.img로부터 kernel과 ramdisk.gz를 추출해 낼 수는 없을까 ? 아래에 왜 이러한 짓(?)을 해야 하는지 그 이유를 정리해 보았다.

<boot.img를 분리해야 하는 이유>

1) 분리 가능하다면, 원본 이미지를 내가 만든 kernel이나 ramdisk.gz로 교체하여 새로운 boot.img를 만들어 낼 수 있을 것이다. 즉, 커널만 build 하거나, ramdisk의 내용만을 수정(가령: init.rc 파일 수정)하여 빠른 시간 안에 새로운 boot.img를 만들어 낼 수 있게 된다.

- 안드로이드의 빌드 시간이 매우 오래 걸리는 점을 감안하면 현실적으로 작업에 도움이 될 것임.
- 필자의 경우 프로젝트 수행 중, 이 과정을 매우 빈번히 수행하여 debugging 시간을 매우 단축시킨 경험이 있음.

2) (남이 만든) boot.img가 오동작 할 경우, 그 원인을 파악하는데도 사용할 수 있다.

- 가령, kernel이 문제가 아니고, init.rc 파일에 문제가 있다면, boot.img를 해부해 보는 것이 문제 해결에 도움이 될 것임.

Boot.img를 분리해야만 하는 이유가 타당하다고 느껴진다면, 아래 내용을 살펴 보기 바란다. 아래 그림 1-41은 원본 boot.img를 분리하여, 자신의 kernel과 ramdisk 이미지로 교체한 후, 새로운 boot.img를 만드는 절차를 보여주고 있다.

<새로운 boot.img 생성 절차>

1) boot.img를 분리한다.

- split_bootimg.pl 펄 스크립트를 이용한다.
- 자세한 사항은 아래 내용 (A) 참조

2) 새로 수정한 kernel로 교체한다.

3) 새로 만든 ramdisk.gz 파일로 교체한다.

- 자세한 사항은 아래 내용 (B) 참조

4) mkbootimg로 다시 새로운 boot.img 파일을 생성한다.

5) fastboot 명령으로 boot.img를 eMMC에 write하여, 부팅을 시도해 본다.

- 자세한 사항은 아래 내용 (C) 참조

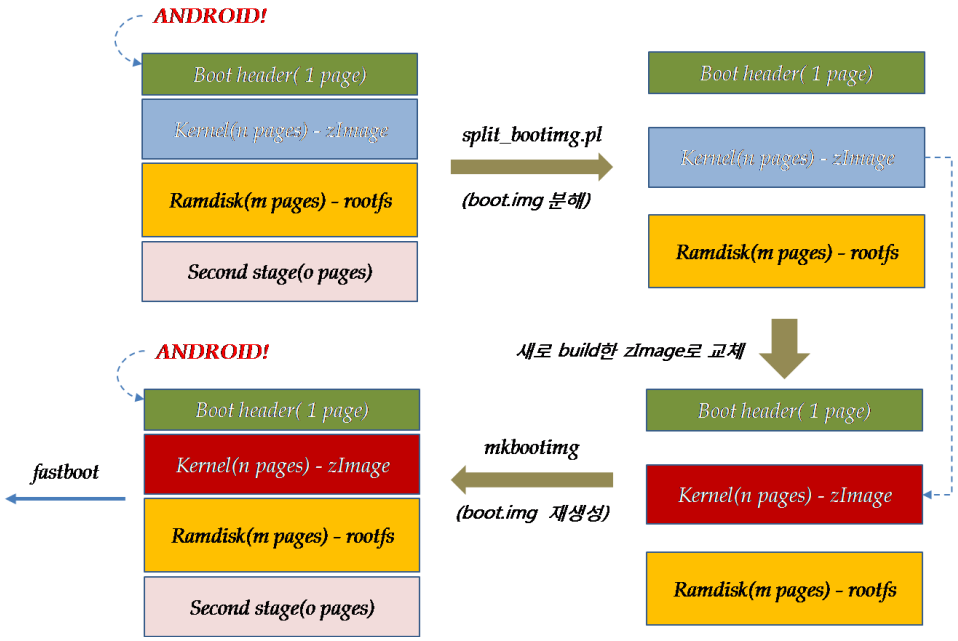

그림 1-41 새로운 boot.img 생성 절차

(A) boot.img 파일 분리하기

 # ./split_bootimg.pl boot.img

-> boot header ← command line 정보 출력됨

-> boot.img-kernel ← kernel

-> boot.img-ramdisk.gz ← ramdisk root file system

- split_bootimg.pl은 William Enck가 작성한 것으로 아래에 그 내용을 실어 보았다. 자세한 사항은 http://www.enck.org/tools.html를 참조하기 바란다.

```
#!/usr/bin/perl
##############################################################################
#
#    File           : split_bootimg.pl
#    Author(s)      : William Enck <enck@cse.psu.edu>
#    Description    : Split appart an Android boot image created
#                     with mkbootimg. The format can be found in
#                     android-src/system/core/mkbootimg/bootimg.h
#
#                     Thanks to alansj on xda-developers.com for
#                     identifying the format in bootimg.h and
#                     describing initial instructions for splitting
#                     the boot.img file.
#
```

```
#    Last Modified : Tue Dec  2 23:36:25 EST 2008
#    By            : William Enck <enck@cse.psu.edu>
#
#    Copyright (c) 2008 The Pennsylvania State University
#    Systems and Internet Infrastructure Security Laboratory
#
#    Licensed under the Apache License, Version 2.0 (the "License");
#    you may not use this file except in compliance with the License.
#    You may obtain a copy of the License at
#
#        http://www.apache.org/licenses/LICENSE-2.0
#
#    Unless required by applicable law or agreed to in writing, software
#    distributed under the License is distributed on an "AS IS" BASIS,
#    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
#    See the License for the specific language governing permissions and
#    limitations under the License.
#
########################################################################

use strict;
use warnings;

# Turn on print flushing
$|++;

########################################################################
## Global Variables and Constants

my $SCRIPT = __FILE__;
my $IMAGE_FN = undef;

# Constants (from bootimg.h)
use constant BOOT_MAGIC => 'ANDROID!';
use constant BOOT_MAGIC_SIZE => 8;
use constant BOOT_NAME_SIZE => 16;
use constant BOOT_ARGS_SIZE => 512;

# Unsigned integers are 4 bytes
use constant UNSIGNED_SIZE => 4;

# Parsed Values
my $PAGE_SIZE = undef;
my $KERNEL_SIZE = undef;
my $RAMDISK_SIZE = undef;
my $SECOND_SIZE = undef;

########################################################################
## Main Code
```

```
&parse_cmdline();
&parse_header($IMAGE_FN);

=format (from bootimg.h)
** +-----------------+
** | boot header       | 1 page
** +-----------------+
** | kernel               | n pages
** +-----------------+
** | ramdisk              | m pages
** +-----------------+
** | second stage      | o pages
** +-----------------+
**
** n = (kernel_size + page_size - 1) / page_size
** m = (ramdisk_size + page_size - 1) / page_size
** o = (second_size + page_size - 1) / page_size
=cut

my $n = int(($KERNEL_SIZE + $PAGE_SIZE - 1) / $PAGE_SIZE);
my $m = int(($RAMDISK_SIZE + $PAGE_SIZE - 1) / $PAGE_SIZE);
my $o = int(($SECOND_SIZE + $PAGE_SIZE - 1) / $PAGE_SIZE);

my $k_offset = $PAGE_SIZE;
my $r_offset = $k_offset + ($n * $PAGE_SIZE);
my $s_offset = $r_offset + ($m * $PAGE_SIZE);

(my $base = $IMAGE_FN) =~ s/.*₩/(.*)$/$1/;
my $k_file = $base . "-kernel";
my $r_file = $base . "-ramdisk.gz";
my $s_file = $base . "-second.gz";

# The kernel is always there
print "Writing $k_file ...";
&dump_file($IMAGE_FN, $k_file, $k_offset, $KERNEL_SIZE);
print " complete.₩n";

# The ramdisk is always there
print "Writing $r_file ...";
&dump_file($IMAGE_FN, $r_file, $r_offset, $RAMDISK_SIZE);
print " complete.₩n";

# The Second stage bootloader is optional
unless ($SECOND_SIZE == 0) {
    print "Writing $s_file ...";
    &dump_file($IMAGE_FN, $s_file, $s_offset, $SECOND_SIZE);
    print " complete.₩n";
}
```

```
################################################################
## Supporting Subroutines

=header_format (from bootimg.h)
struct boot_img_hdr
{
    unsigned char magic[BOOT_MAGIC_SIZE];

    unsigned kernel_size;   /* size in bytes */
    unsigned kernel_addr;   /* physical load addr */

    unsigned ramdisk_size; /* size in bytes */
    unsigned ramdisk_addr; /* physical load addr */

    unsigned second_size;   /* size in bytes */
    unsigned second_addr;   /* physical load addr */

    unsigned tags_addr;     /* physical addr for kernel tags */
    unsigned page_size;     /* flash page size we assume */
    unsigned unused[2];     /* future expansion: should be 0 */

    unsigned char name[BOOT_NAME_SIZE]; /* asciiz product name */

    unsigned char cmdline[BOOT_ARGS_SIZE];

    unsigned id[8]; /* timestamp / checksum / sha1 / etc */
};
=cut
sub parse_header {
    my ($fn) = @_;
    my $buf = undef;

    open INF, $fn or die "Could not open $fn: $!₩n";
    binmode INF;

    # Read the Magic
    read(INF, $buf, BOOT_MAGIC_SIZE);
    unless ($buf eq BOOT_MAGIC) {
        die "Android Magic not found in $fn. Giving up.₩n";
    }

    # Read kernel size and address (assume little-endian)
    read(INF, $buf, UNSIGNED_SIZE * 2);
    my ($k_size, $k_addr) = unpack("VV", $buf);

    # Read ramdisk size and address (assume little-endian)
    read(INF, $buf, UNSIGNED_SIZE * 2);
    my ($r_size, $r_addr) = unpack("VV", $buf);
```

```
    # Read second size and address (assume little-endian)
    read(INF, $buf, UNSIGNED_SIZE * 2);
    my ($s_size, $s_addr) = unpack("VV", $buf);

    # Ignore tags_addr
    read(INF, $buf, UNSIGNED_SIZE);

    # get the page size (assume little-endian)
    read(INF, $buf, UNSIGNED_SIZE);
    my ($p_size) = unpack("V", $buf);

    # Ignore unused
    read(INF, $buf, UNSIGNED_SIZE * 2);

    # Read the name (board name)
    read(INF, $buf, BOOT_NAME_SIZE);
    my $name = $buf;

    # Read the command line
    read(INF, $buf, BOOT_ARGS_SIZE);
    my $cmdline = $buf;

    # Ignore the id
    read(INF, $buf, UNSIGNED_SIZE * 8);

    # Close the file
    close INF;

    # Print important values
    printf "Page size: %d (0x%08x)₩n", $p_size, $p_size;
    printf "Kernel size: %d (0x%08x)₩n", $k_size, $k_size;
    printf "Ramdisk size: %d (0x%08x)₩n", $r_size, $r_size;
    printf "Second size: %d (0x%08x)₩n", $s_size, $s_size;
    printf "Board name: $name₩n";
    printf "Command line: $cmdline₩n";

    # Save the values
    $PAGE_SIZE = $p_size;
    $KERNEL_SIZE = $k_size;
    $RAMDISK_SIZE = $r_size;
    $SECOND_SIZE = $s_size;
}

sub dump_file {
    my ($infn, $outfn, $offset, $size) = @_;
    my $buf = undef;

    open INF, $infn or die "Could not open $infn: $!₩n";
```

```
    open OUTF, ">$outfn" or die "Could not open $outfn: $!₩n";

    binmode INF;
    binmode OUTF;

    seek(INF, $offset, 0) or die "Could not seek in $infn: $!₩n";
    read(INF, $buf, $size) or die "Could not read $infn: $!₩n";
    print OUTF $buf or die "Could not write $outfn: $!₩n";

    close INF;
    close OUTF;
}

###############################################################
## Configuration Subroutines

sub parse_cmdline {
    unless ($#ARGV == 0) {
        die "Usage: $SCRIPT boot.img₩n";
    }
    $IMAGE_FN = $ARGV[0];
}
```

(B) boot.img-ramdisk.gz(ramdisk image) 파일 해부하기

 # gzip -d boot.img-ramdisk.gz

- gzip 압축 해제하기

 # cpio -i < boot.img-ramdisk

- cpio 명령으로 ramdisk 해제하기
- 현재 디렉토리에 ramdisk file system을 구성하는 파일이 풀리게 됨.

... 디렉토리로 이동하여 파일 직접 수정(예: vi init.rc) ...

```
default.prop
/dev
init
init.goldfish.rc
init.qcom.rc
init.rc
init.target.rc
init.usb.rc
logo.rle
/etc
/sbin
/sys
ueventd.goldfish.rc
ueventd.rc
```

ueventd.{hardware}.rc

 # find . | cpio –o –H newc | gzip > ../newramdisk.cpio.gz

- cpio와 gzip 명령으로 새로운 ramdisk.gz 파일(newramdisk.cpio.gz)을 재 생성함.
- 위에서 cpio로 파일을 풀어둔 디렉토리에서 명령을 실행.

(C) fastboot으로 boot.img write하기

 # adb reboot-bootloader

- fastboot mode로 전환
- 이 명령이 동작하지 않을 경우, 시스템에서 정의한 key 버튼 조합으로 fastboot에 진입 가능함.

$ sudo fastboot devices
???????????? fastboot

- 장치 인식(확인)

 # fastboot flash boot ./boot.img

- 새로 만든 boot.img를 eMMC 혹은 NAND flash에 써줌.
- 쓰기 전에 erase를 해야 할 경우도 있음(fastboot erase boot).

$ sudo fastboot reboot
rebooting...
finished. total time: 0.000s

- 정상적으로 써 진 경우, 시스템을 재 부팅하여, 정상 동작하는지 확인함.

recovery.img 해부

지금까지 boot.img를 조작하는 방법을 살펴 보았다. recovery.img의 경우도 boot.img와 동일한 구조로 이루어져 있는데, 차이가 있다면 ramdisk의 내용이 recovery 상황에 맞도록 간소화되어 있다는 점(recovery 전용 화면을 출력하고, 새로운 firmware 이미지로 교체 작업을 수행함)이다. 따라서, boot.img를 위해 수행했던 작업을 그대로 recovery.img에 적용이 가능하다. 아래에 recovery.img의 구성 내용을 정리해 보았다.

<recovery.img의 구성>

1) boot header + kernel(zImage) + ramdisk.gz + image signature(선택 사항임)
2) kernel은 boot.img에서 사용한 동일한 커널 이미지를 그대로 사용함.
3) ramdisk.gz의 내용은 init.rc, init.{hardware}.rc가 수정됨(간소화됨)

- 그래픽 화면을 초기화
- 간단한 설치 UI 화면을 출력하는 application 실행

■ Firmware 업그레이드 진행 프로그램 시작

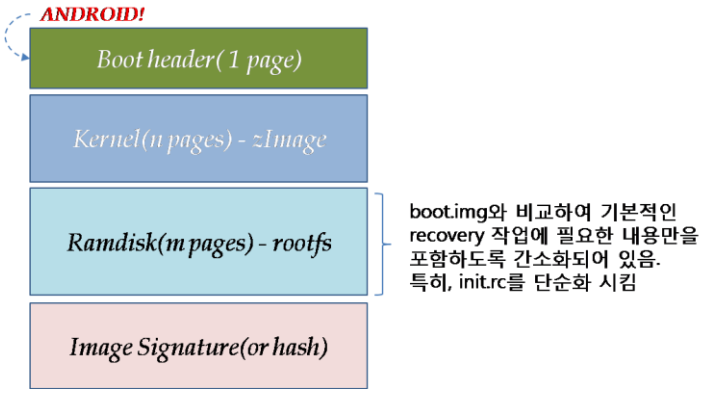

그림 1-42 recovery.img의 구성

system.img.ext4 해부

다음으로 분석해 볼 사항은 system 이미지이다. system.img 내에는 android system과 관련한 거의 모든 내용이 담겨 있다고 볼 수 있다. 따라서, 그 크기도 매우 크다(수백 MB ~ 1GB 이상이 되는 경우도 있음).

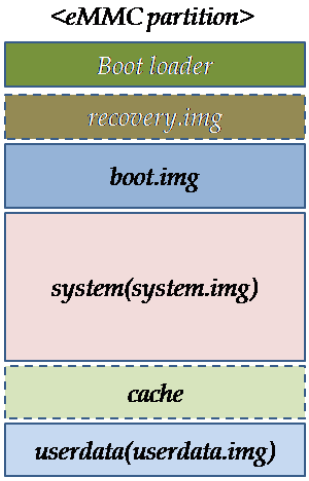

그림 1-43 eMMC 파티션 정보 – system.img

system.img.ext4는 ext4 파일 시스템을 사용하는 system 이미지를 뜻하는데, 앞서 언급한 바와 같이 out/target/product/msm8960/system 디렉토리의 내용과 make_ext4fs(out/host 이하 디렉토리 참조)를 사용하여 만들게 된다. 아래에서 system.img.ext4를 생성하고, 분해하는 절차를 따라가 보자.

<system.img.ext4 생성 절차>

 # make_ext4fs -l 500M -a system system.img.ext4 system/

■ system/ 디렉토리 내용을 가지고, 500MB 크기의 system.img.ext4 파일 생성

<option 의 의미>

-l : 파일 사이즈를 설정(예: 500M)

-a : android mount point(system)

파일명: 생성 되는 파일명

폴더: system 폴더내의 파일들이 ext4 image 로 만들어지게 됨.

그럼, 반대로 system.img.ext4 파일로부터 system/ 디렉토리를 추출해 내기 위해서는 어떻게 해야 할까? 아래에 이 과정을 정리해 보았다.

<system.img.ext4 해부 절차>

*# ./**ext2simg** system.img.ext4 system.spc*

- 결과물이 될 파일, 적당한 이름을 지정해 주면 됨.
- Ext4 -> Sparse 파일로 전환

*# ./**simg2img** system.spc system.img.raw*

- 결과물이 될 파일, 적당한 이름을 지정해 주면 됨.
- Sparse 파일을 Raw 파일로 전환

 # mkdir temp

- system.img.raw 파일을 mount할 임시 디렉토리 생성

 # mount -t ext4 -o loop ./system.img.raw ./temp

- temp 디렉토리에 system.img.raw 파일을 mount해 줌.
- Mount에 성공 시, temp 디렉토리로 이동해 보면, system.img.ext4의 내용이 모두 보이게 될 것임.

 # umount temp

- unmount하고자 할 경우 사용함.

[참고 사항]

1) ext2simg 명령어는 ext 파일 포맷으로 된 파일을 sparse 파일로 바꾸어 줌.

- Sparse 파일은 큰 용량을 디스크에 미리 잡아 두어야 하지만 실제로 그 안에 들어가는 데이터는 훨씬 적은 경우에 생성해 주는 파일을 말함(즉, 커다란 영역 내에 듬성 듬성 실제 내용이 들어가는 방식임).

2) simg2img 명령은 sparse 파일을 실제 raw image 파일로 만들어 줌.

3) 위의 두 명령은 out/host 이하 디렉토리 아래에서 찾을 수 있음.

4. 본 서의 대상 범위, 참조 코드 소개

1. 본서의 대상 범위

본서는 아래 그림에서 빨간색으로 표시된 LINUX KERNEL 만을 그 대상으로 한다.

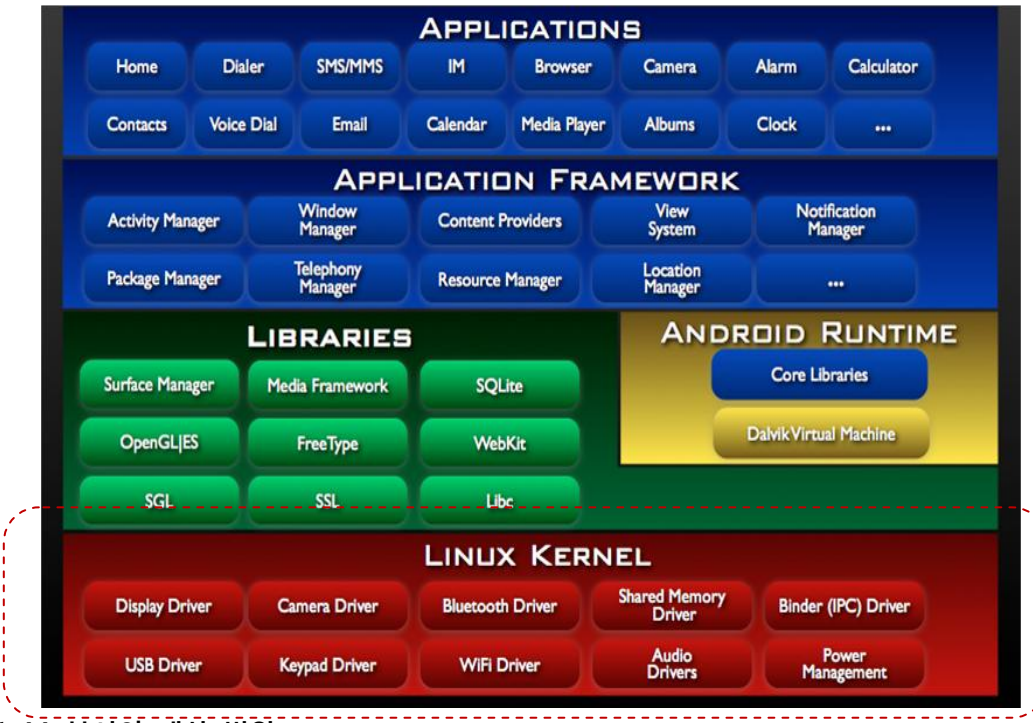

그림 1-44 본서의 대상 범위

<본서의 각 장 내용 요약>

2장 및 3장에서는 커널 프로그램 기법 중, 개발자에게 꼭 필요한 부분만을 엄선하여 정리하였다. 기존의 커널 관련 서적과는 달리 원리 위주의 설명 보다는 실제로 개발자가 디바이스 드라이버를 작성할 경우에 반드시 알아야 할 커널 프로그래밍 기법이 무엇인가에 초점을 맞추어 내용을 전개하였다.

4장에서는 ARM 기반 보드 초기화를 위해 고려해야 할 사항 및 최근에 ARM 개발자 그룹에 의해 새롭게 추가된 몇 가지 커널 프레임워크(clock, pinctrl, device tree 등)를 정리하였다. 특히 최근에 표준으로 자리 잡은 Device Tree의 개념과 개발을 위한 필수 내용을 정리하였으며, 기존 방식인 Platform Device에 대해서도 소개를 빼 놓지 않았다.

5장에서는 ARM 보드 초기화 후, 동작 시켜야하는 여러 장치 중, 가장 먼저 파워 관리(Power Management) 부분을 소개하였다. 이에 포함되는 내용으로는 CPUFreq, CPUIdle, Kernel PM, Runtime PM, Regulator 등이 있다.

6장에서는 파워 관리 부분을 제외한 여러 장치 중, 기본이 되는 연결 장치를 위주로 소개를 진행하였다. 이에 해당되는 내용으로는 SD/MMC, I^2C, SPI, USB, Input 드라이버 등이 있다.

7장에서는 스마트 폰 및 태블릿에 주로 사용되는 장치 중, 핵심 장치 몇 가지를 중심으로 소개하였다. 즉, 디스플레이, 카메라, 오디오, 충전, Wi-Fi 가 이에 해당한다.

마지막으로, **8장에서는** 반드시 필요한 핵심 커널 디버깅 기법을 엄선하여 정리해 보았다.

위의 내용을 그림으로 모아 보면 다음과 같다.

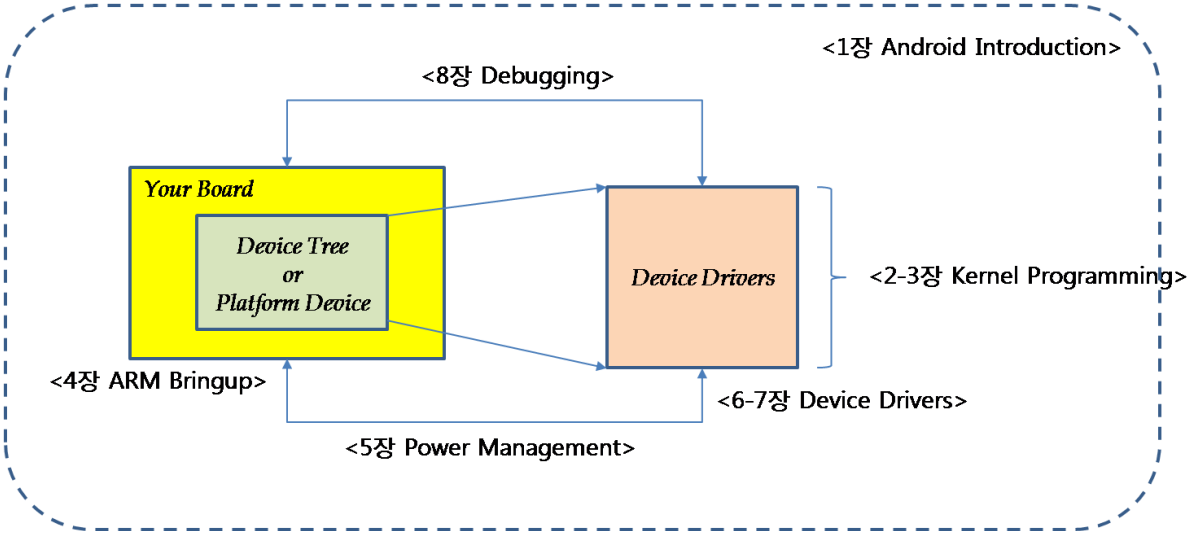

그림 1-45 각 장별 소개

2. 본서에서 참조한 코드 소개

본 서는 안드로이드 커널 개발자를 위해 작성되었다. 따라서 최신 안드로이드 소스 코드(현재 Jelly Bean 4.3) 내의 커널 코드를 소개하는 것이 합당하다 할 것이다. 하지만, 최신 안드로이드 코드 내에 포함되어 있는 리눅스 커널의 버전을 살펴보면, 업체에 따라 3.0.x ~ 3.4로 제각각 차이를 보이고 있는 것이 사실이다. 또한 3.4 버전을 사용하는 경우도 실상을 드려다 보면, 3.0.x의 많은 feature(예: PM/wakelock)를 그대로 사용하고 있어, 3.4 버전이라고 말하기도 좀 애매한 부분이 있다.

따라서, 필자는 고민 끝에 리눅스 커널을 소개함에 있어, 안드로이드 소스 코드 내의 코드를 참조하기 보다는 바닐라(Vanilla) 리눅스 커널을 따라가는 것이 좋겠다는 판단을 내렸다. 이러한 판단을 내리게 된 데에는, 현재 시중에 나와 있는 대부분의 리눅스 서적이 예전 2.6 버전을 위주로 소개하고 있다는 점도 한 목 작용했다.

본서에서 참조한 내용을 정리해 보면 다음과 같다.

참조 코드	관련 정보
Kernel 3.10.3 (vanilla)	http://www.kernel.org
QAF Android JB 4.2	http://www.codeaurora.org
Beagleboard Android JB 4.2	https://code.google.com/p/rowboat/

자, 이제 모든 준비는 끝났으니, 다음 장부터 커널 영역으로 넘어가 보자.

References

[1] Android Anatomy and Physiology, Google IO 2008

[2] Mastering the Android Media Framework, Google IO 2009

[3] Embedded Android, Karim Yaghmour, O'reilly, 2013

[4] 인사이드 안드로이드, 송형주 외 4인, 위키북스

[5] 안드로이드 아나토미, 김태연 외 3인, 개발자가 행복한 세상

[6] 안드로이드 미디어 프레임워크, 김태연 외 4인, 개발자가 행복한 세상

[7] 안드로이드의 모든 것 분석과 포팅, 고현철/유형목, 한빛미디어

[8] 8th.kandroid.application_framework_print.ppt, 양정수(www.kandroid.org)

[9] kandroid_for_jco_20090228_final.pdf, 양정수(www.kandroid.org)

[10] android_app_lecture_note_2nd_of_5days_v2.9.pdf, 양정수(www.kandroid.org)

[11] 01.Android-gingerbread-multimedia-framework-structure.pdf , 고현철, 이솝포럼

[12] AndroidMMF-Details_v04.pdf, 김태용, windriver

[13] Android ICS Porting Guide, Chunghan Yi, www.kandroid.org

[14] Android Build System, Chunghan Yi, www.kandroid.org