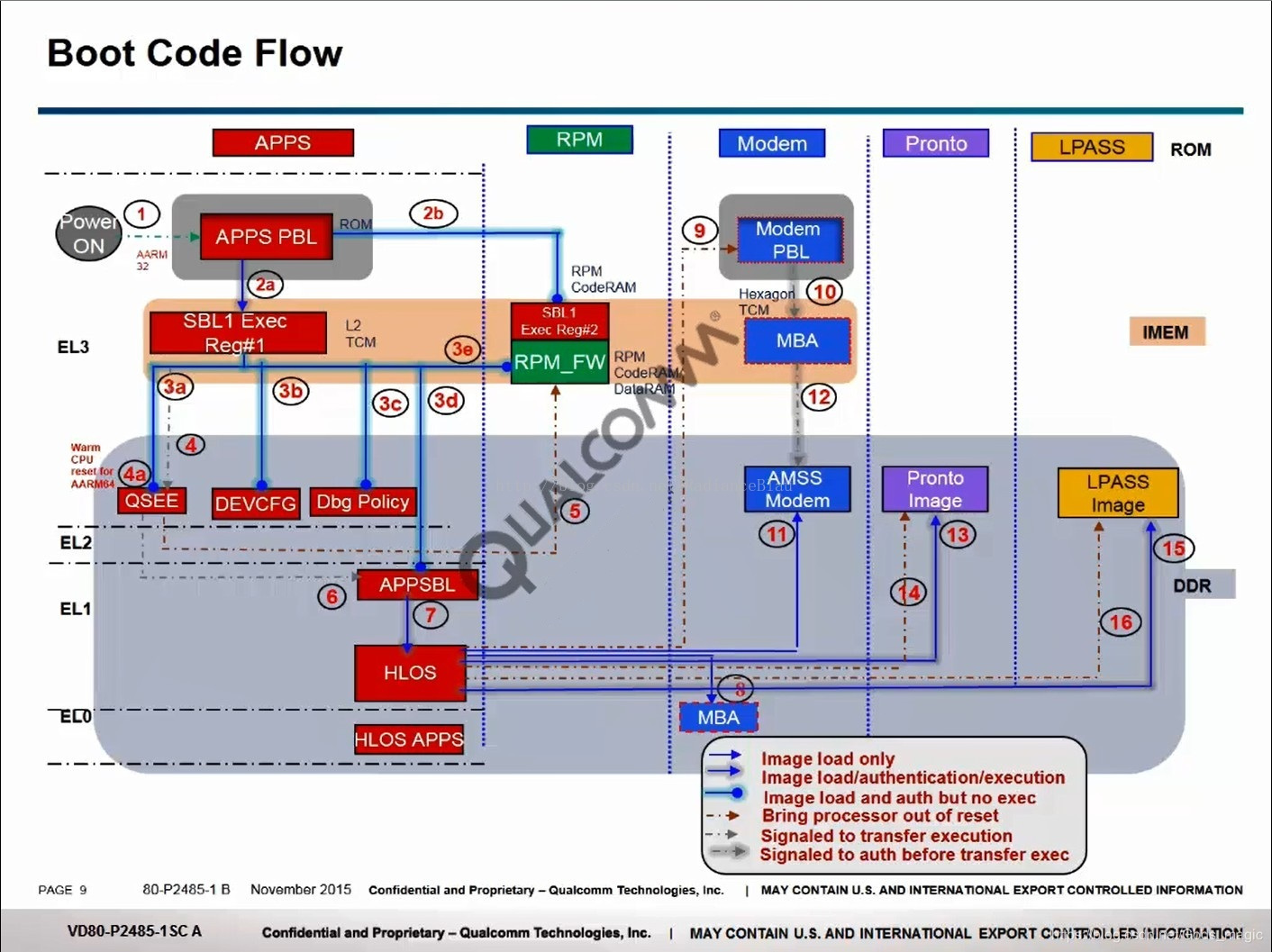
**The BootLoader startup process for the Qualcomm platform.**

**One, what is BootLoader.**

The BootLoader code is a piece of code that is executed before the chip is reset and enters the operating system. Primarily used to complete the transition from hardware boot to operating system boot, thus providing a basic operating environment for the operating system.

BootLoder's main start-up processes can be summarized as: PBL stage, SBL phase, LK stage. The kernel is then loaded and started.

**Second, noun interpretation.**



**5 processors:**

1. **APPS**：Cortex A53 core(MSM8953)，run android。
2. **RPM**(Resource Power Manager): CortexM3 core, primarily for low-power applications.
3. **Modem**(MSS\_QDSP6): Qualcomm has its own instruction set processor, processing 3G, 4G communication protocols, etc.
4. **Pronto**(WCNSS): Processing wifi-related code.
5. **LPASS:**Audio-related.

**Start-up related:**

1. **PBL**(Primary Boot Loader): Located in the inside ROM of the chip, is the true first line of code executed after the chip is powered up and loads SBL1 during the normal start-up process. If the start exception virtuals out of the 9008 port available for the Emergency download (force\_boot\_from\_usb pins on the short board (MSM8953 for gpio37) to 1.8v can be forced into emergency download mode). (This piece of code is encapsulated inside the chip and is not open source)
2. **SBL1**(Second BootLoader stage 1) ：located in eMMC loaded，Load by PBL, initialize buses, DDR, clocks, etc.
3. **QSEE/TrustZone** Security Related.
4. **Debug Policy** debugging is relevant.
5. **APPSBL** ：It is BootLoader, currently using LK (little kernel).
6. **HLOS**(High LevelOperating System) Namely Linux/Android。

**Third, start the process.**

1. The AP side CPU is powered on.
2. The PBL of the ROM inside the chip runs first, and the PBL is loaded from the boot device (eMMC) and verified in SBL1 to the TCM. The TCM here can be understood as a second-level cache of the CPU. Since PBL is able to load SBL1 from boot device (eMMC), the PBL should have initialized boot device.
3. SBL1 initializes the DDR and loads and verifies the following images from the boot device: QSEE or TZ mirror, QHEE mirror, RPM\_FW, mirror, APPBL, etc.
4. After SBL1 loads and validates the above image, the execution is transferred to QSEE, which sets up and initializes a secure execution environment.
5. QSEE informs RPM to execute the relevant RPM\_FW code.
6. QSEE transfers enforcement power to APPSBL, which is also LK.
7. LK loads HLOS kernel.

**Fourth, the code process brief process.**

**SBL**

The PBL section is not open source, so start with SBL.

**sbl1 entrance: sbl1.s**

This part of the code path is in: boot\_images/core/boot/secboot3/hw/msm89xx/sbl1/sbl1.s, this file boots the processor, mainly to implement the following operations:

Partial source:

* Set up the hardware to continue the boot process.
* Initialize ddr.
* Load Trust\_Zone operating system.
* Load RPM firmware.
* Load the APPSBL and continue the boot process.

1. IMPORT |Image$$SBL1\_SVC\_STACK$$ZI$$Limit|
2. IMPORT |Image$$SBL1\_UND\_STACK$$ZI$$Limit|
3. IMPORT |Image$$SBL1\_ABT\_STACK$$ZI$$Limit|
4. IMPORT boot\_undefined\_instruction\_c\_handler
5. IMPORT boot\_swi\_c\_handler
6. IMPORT boot\_prefetch\_abort\_c\_handler
7. IMPORT boot\_data\_abort\_c\_handler
8. IMPORT boot\_reserved\_c\_handler
9. IMPORT boot\_irq\_c\_handler
10. IMPORT boot\_fiq\_c\_handler
11. IMPORT boot\_nested\_exception\_c\_handler
12. IMPORT sbl1\_main\_ctl #Mainly focus on this function
13. IMPORT boot\_crash\_dump\_regs\_ptr
14. ...

**sbl1\_main\_ctl function.**

Path: ... sbl1 sbl1\_mc.c.

1. /\* Calculate the SBL start time for use during boot logger initialization. \*/
2. sbl\_start\_time = CALCULATE\_TIMESTAMP(HWIO\_IN(TIMETICK\_CLK));
3. boot\_clock\_debug\_init();
4. /\* Enter debug mode if debug cookie is set \*/
5. sbl1\_debug\_mode\_enter();
6. /\* Initialize the stack protection canary \*/
7. boot\_init\_stack\_chk\_canary();
9. /\* Initialize boot shared imem \*/
10. boot\_shared\_imem\_init(&bl\_shared\_data);
11. /\*Initialize RAM \*/
12. boot\_ram\_init(&sbl1\_ram\_init\_data);
13. /\*Initialize the log system, that is, the serial port driver \*/
14. sbl1\_boot\_logger\_init(&boot\_log\_data, pbl\_shared);
15. /\*Retrieve the data passed by PBL \*/
16. sbl1\_retrieve\_shared\_info\_from\_pbl(pbl\_shared);
17. /\* Initialize the QSEE interface \*/
18. sbl1\_init\_sbl\_qsee\_interface(&bl\_shared\_data,&sbl\_verified\_info);
19. /\* Initialize SBL memory map. Initializing early because drivers could be located in RPM Code RAM. \*/
20. sbl1\_populate\_initial\_mem\_map(&bl\_shared\_data);
21. /\*Initialize DAL \*/
22. boot\_DALSYS\_InitMod(NULL);
23. /\*Configure the PMIC chip so that we can pass PS\_HOLD Reset\*/
24. sbl1\_hw\_init();
25. /\*The target dependent process for executing sbl1 \*/
26. boot\_config\_process\_bl(&bl\_shared\_data, SBL1\_IMG, sbl1\_config\_table);

**sbl1\_config\_table function.**

路径：boot\_images\core\boot\secboot3\hw\msmxxxx\sbl1\sbl1\_config.c

sbl1\_config\_table is an array of structures that store the configuration parameters and execution functions needed to load Images such as QSEE, RPM, AND APPSBL.

1. boot\_configuration\_table\_entry sbl1\_config\_table[] =
2. {
4. /\* SBL1 -> QSEE \*/
5. {
6. SBL1\_IMG, /\* host\_img\_id \*/
7. CONFIG\_IMG\_QC, /\* host\_img\_type \*/
8. GEN\_IMG, /\* target\_img\_id \*/
9. CONFIG\_IMG\_ELF, /\* target\_img\_type \*/
10. ...
11. load\_qsee\_pre\_procs, /\* pre\_procs \*/
12. load\_qsee\_post\_procs, /\* post\_procs \*/
13. }
14. /\* SBL1 -> QHEE \*/
15. ...
16. /\* SBL1 -> RPM \*/
17. ...
18. /\* SBL1 -> APPSBL （I.e. lk part） \*/
19. ...

**LK**

**LK approximate process:**

**bootloader/lk directory structure:**

* Do a variety of early initialization work, including cpu, emmc, ddr, locks, thread, etc.
* Determines whether the condition entering recovery or fastboot is triggered, or normal boot if not.
* Get **boot.img** from emmc and load it into the specified memory area (scratch region).
* Load Kernel, Ramdisk, DeviceTree until the specified address.
* Guide Kernel.

1. -app #Application related
2. -arch #arm system、CPU Architecture
3. -dev #Device driver
4. -images #image Picture resource
5. -include #head File
6. -kernel #LK System, main file, main.c
7. -lib #Library file
8. -platform #Platform files such as：msm8916
9. -project #mkfile
10. -scripts #Script file
11. -target #Target device file

**Code flow:**

the lk is compiled in conjunction with and language, which favors hardware and the underlying code being written using , while the code that provides functionality on the upper level is written using .arm Compilation carm compilation c

**Entrance:**

The entry point for the lk code is specified in the link script file with the suffix under the directory.arch/arm.ld

The specified entry points are located at **arch/arm/crt0. \_start**function in the S file:

* system-onesegment.ld
* system-twosegment.ld

**\_start**the main role is to set some cpu characteristics, and then initialize the stack environment required for the various c programs to run, and when finished, jump directly to the **kmian**function into the c language environment.

**Kmain() function:**

1. void kmain(void)
2. {
3. thread\_init\_early(); // Initialize the lk thread context
4. arch\_early\_init(); // Architecture initialization，If closed cache，Enable mmu
5. platform\_early\_init(); // Early initialization of platform hardware
6. target\_early\_init(); //Early initialization of the target device
7. bs\_set\_timestamp(BS\_BL\_START);
8. call\_constructors(); //Static constructor initialization
9. heap\_init(); // Heap initialization
10. thread\_init(); // Initialize thread
11. dpc\_init(); //lkSystem controller initialization
12. timer\_init(); //kernelClock initialization
14. #if (!ENABLE\_NANDWRITE)
15. thread\_resume(thread\_create("bootstrap2", &bootstrap2, NULL, DEFAULT\_PRIORITY, DEFAULT\_STACK\_SIZE)); // Create a thread initialization system
16. exit\_critical\_section(); //Enable interrupt
17. thread\_become\_idle(); //This thread is switched to idleThread
18. #else
19. bootstrap\_nandwrite();
20. #endif
21. }

**bootstrap2**

This function is called by a thread created in kmain, with the path: bootable?bootloader?lk?kernel?main.c.

1. static int bootstrap2(void \*arg)
2. {
3. arch\_init(); //Architecture initialization
4. bio\_init();
5. fs\_init();
6. platform\_init(); //Platform initialization, mainly initialize system clock, overclocking, etc.
7. target\_init(); //Initialize the target device, mainly initialize Flash, integrate partition table, etc.
8. apps\_init(); // Application function initialization, call aboot\_init, load kernel, etc.
9. }

**apps\_init () function:**

app\_init The function is located in the file.app/app.c

1. /\* app entry point \*/
2. struct app\_descriptor;
3. typedef void (\*app\_init)(const struct app\_descriptor \*);
4. typedef void (\*app\_entry)(const struct app\_descriptor \*, void \*args);
6. /\* app startup flags \*/
7. #define APP\_FLAG\_DONT\_START\_ON\_BOOT 0x1
9. /\* each app needs to define one of these to define its startup conditions \*/
10. struct app\_descriptor {
11. const char \*name;
12. app\_init init;
13. app\_entry entry;
14. unsigned int flags;
15. };
17. void apps\_init(void)
18. {
19. const struct app\_descriptor \*app;
21. /\* call all the init routines \*/
22. for (app = &\_\_apps\_start; app != &\_\_apps\_end; app++) {
23. if (app->init)
24. app->init(app);
25. }
27. /\* start any that want to start on boot \*/
28. for (app = &\_\_apps\_start; app != &\_\_apps\_end; app++) {
29. if (app->entry && (app->flags & APP\_FLAG\_DONT\_START\_ON\_BOOT) == 0) {
30. start\_app(app);
31. }
32. }
33. }
35. static void start\_app(const struct app\_descriptor \*app)
36. {
37. thread\_t \*thr;
38. printf("starting app %s\n", app->name);
40. thr = thread\_create(app->name, &app\_thread\_entry, (void \*)app, DEFAULT\_PRIORITY, DEFAULT\_STACK\_SIZE);
41. if(!thr)
42. {
43. return;
44. }
45. thread\_resume(thr);
46. }

The entire process of traversing the app and starting is not complicated, and interestingly, **\_\_apps\_start** and **\_\_apps\_end** definitions, these two variable symbols do not exist in all source files, but in **the arch/arm/.ld** link script,**\_\_apps\_start** and **\_\_apps\_end** are custom two symbols that represent the start and end of the custom segment **.apps.** This means that all apps implement a plug-in system by registering in a special segment .apps, which is a very sophisticated design. Any subsequent new apps only need to use the following two macro declarations to register to the .apps segment:

1. #define \_\_SECTION(x) \_\_attribute((section(x)))
2. #define APP\_START(appname) struct app\_descriptor \_app\_##appname \_\_SECTION(".apps") = { .name = #appname,
3. #define APP\_END };

In **bootloader/lk/app/aboot/aboot.c**end：

1. APP\_START(aboot)
2. .init = aboot\_init,
3. APP\_END

Description **apps\_init** function called **the aboot\_init** **function.**

**aboot\_init function:**

It is the **aboot**entry function registered by **APP\_START,**from which all the functions of **aboot**begin. Located in **the bootloader/lk/app/aboot/aboot.c** file.

Simple process:

* Set the size of the NAND/eMMC read information page.
* Print log information and initialization of some device modules.
* Check the keys to determine if you are in fastboot or recovery mode.
* Get device info (whether the device is lock, whether it is root, whether img has been authorized, etc.)
* Load the kernel from eMMC or NAND flash.

Main function:**boot\_linux\_from\_mmc** .

boot.img Parsing：<https://blog.csdn.net/ty3219/article/details/78879398>

* The boot image is loaded first.
* Get the image header size, assign the header image buffer, and load the header image from the specified partition.
* If **Secure Boot** is configured.
* ImagesizeActure sssignature size.
* imagesize = imagesizeActure + image header pagesize
* Assign image buffer and load the image from the partition.
* **Check the signature.**
* The validation failure notifies THE TZ, the startup failed.
* If the validation succeeds, the image is loaded.
* Unzip boot.img, load kernel, ramdisk, device tree, and wait until the specified address.
* Notify the end of the TZ load. Start kernel.
* BootLoader ends.

**boot.image plus sign:**

1. The original boot.img is calculated by sha256 algorithm to obtain a hash value.
2. Use private key to add the hash value.
3. Add the final hash value and signature to the end of the original boot.img.

**Signature check:**

1. Take boot.img for SHA256 hash calculation to get file digest1.
2. Decrypt the signature through the public key and get the file digest2.
3. Compare the two files of digest1 and digest2.
4. The same check succeeds, and the check fails in the other.

A rough understanding of the Boot process.