MediaTek details: Partitions and Preloader

Jul 4, 2015

NOTE: This is a continuation of the previous article in the series. The information was obtained from various sources and through reverse engineering, don't take it as a reference!

UPDATE 03.07.2015: Add Download Agent references.

After the Boot ROM has completed the initialization of the core hardware, it loads the first block from the eMMC flash into the On-chip SRAM and starts execution. Usually this would be the location of the operating system bootloader firmware, but on MediaTek SoCs it usually isn't. There's an intermediate step: the Preloader. It is a piece of software that abstracts a bit between the platform and the actual bootloader, and it offers some additional features like the ability to boot from either MMC or NAND Flash or to read/write various parts of the flash via USB.

Overview

When MediaTek ships a kernel source package to a manufacturer, the package also contains the source code for the Preloader. Depending on the changes the manufacturer makes, the Preloader does slighlty different things on different SoCs and boards, so it's not easy to come up with a generic description that fits all systems.

The following analysis is based on the ThunderKernel source code for the MT6582 SoC. The distribution is made up of the following parts:

- A platform-specific part in `mediatek/platform/${platform}/preloader`, this contains most of the code.
- A custom (manufacturer-specific) part in `mediatek/custom/${platform}/preloader`.
- A device-specific part in `mediatek/custom/${DEVICE}/preloader`.

Now let's analyse the code.

CPU Init

When the preloader starts, the SoC has not yet been fully initialised and many things are in a random state. A piece of assembler code in `mediatek/platform/${platform}/preloader/src/init/init.s` performs the following steps to create a clean state:

1. Clear all registers.
2. Switch to the SVC32 privileged mode.
3. Disable interrupts.
4. Set up the caches and some other, minor details (e.g. the stack).
5. Jump to the `main` method in the C code.

Now execution continues in `mediatek/platform/${platform}/preloader/src/core/main.c`, which calls a lot of other methods, which again call a lot of other methods. It makes no sense to disect every line of code in this article, so I'll just give a high-level view of what's happening and explain some details where necessary.

Platform init

The Preloader relies on some peripherials, so it has to initialize them. This is mostly done in the `platform_pre_init` and `platform_init` methods. The list of peripherials contains the timer, the PLL clock, the DDR memory controller, the Watchdog, the GPIO pins, the UART, the USB 1.1 port and the power management circuit.

There is something special here: after the flash storage has been initialised, the Preloader offers an early "emergency download" mode. The manufacturer can define a hardware key, which, when pressed during Preloader platform init, immediately reboots back into the Boot ROM and waits for a download.

At this point the Preloader also records the reason why the system was booted:

```
typedef enum {
    BR_POWER_KEY = 0,
    BR_USB,
    BR_RTC,
    BR_WDT,
    BR_WDT_BY_PASS_PWK,
```

```
    BR_TOOL_BY_PASS_PWK,
    BR_2SEC_REBOOT,
    BR_UNKNOWN
} boot_reason_t;
```

Partitions

After bringing up the platform, the preloader has full access to the internal storage. MediaTek decided to partition the storage, but the partition table is hardcoded. It is generated during Preloader generation from an Excel file (yes, really) in `mediatek/build/tools/ptgen/${platform}/partition_table_${platform}.xls` by the command `./makeMtk -t ${device} ptgen` (here is an example for the bq Aquaris E4.5 Ubuntu Edition):

	A	B	C	D	E	F	G	H	I	J	
1	Index	Partition	Type	Size	Main Size(KB)	Size(Byte	Size(Byte)	Down Load?	FB Erase?	FB Download	Re
2	1	PRELOADER	Raw data	256 KB	256	262144	0x40000	1	0	0	BC
3	2	MBR	Raw data	512 KB	512	524288	0x80000	1	0	0	U
4	3	PRO_INFO	Raw data	3 MB	3072	3145728	0x300000	0	0	0	U
5	4	NVRAM	Raw data	5 MB	5120	5242880	0x500000	0	0	0	U
6	5	PROTECT_F	EXT4	10 MB	10240	10485760	0xA00000	0	0	0	U
7	6	PROTECT_S	EXT4	10 MB	10240	10485760	0xA00000	0	0	0	U
8	7	PERSIST	EXT4	48 MB	49152	50331648	0x3000000	0	0	0	U
9	8	SECCFG	Raw data	128 KB	128	131072	0x20000	0	0	0	U
10	9	UBOOT	Raw data	384 KB	384	393216	0x60000	1	0	0	U
11	10	BOOTIMG	Raw data	20 MB	20480	20971520	0x1400000	1	1	1	U
12	11	RECOVERY	Raw data	20 MB	20480	20971520	0x1400000	1	0	0	U
13	12	SEC_RO	EXT4	6 MB	6144	6291456	0x600000	1	0	0	U
14	13	MISC	Raw data	512 KB	512	524288	0x80000	0	0	0	U
15	14	LOGO	Raw data	3 MB	3072	3145728	0x300000	1	0	0	U
16	15	CUSTOM	EXT4	55 MB	56320	57671680	0x3700000	1	0	0	U
17	16	EXPDB	Raw data	10 MB	10240	10485760	0xA00000	0	0	0	U
18	17	TEE1	Raw data	5 MB	5120	5242880	0x500000	1	1	1	U
19	18	TEE2	Raw data	5 MB	5120	5242880	0x500000	1	1	1	U
20	19	ANDROID	EXT4	1024 MB	1048576	1,1E+009	0x40000000	1	1	1	U
21	20	CACHE	EXT4	700 MB	716800	7,3E+008	0x2BC00000	1	1	1	U
22	21	USRDATA	EXT4	924 MB	946176	9,7E+008	0x39C00000	1	1	1	U
23	22	FAT	FAT	0	0	0	0x0	0	0	0	U
24	23	BMTPOOL	Raw data	21 MB	21504	22020096	0x1500000	0	0	0	U
25		END	Flag	0	0	0	0	0	0	0	

The generated partition table structure is stored in `out/target/product/krillin/obj/PRELOADER_OBJ/cust_part.c` and looks like this (example for the bq Aquaris E4.5 Ubuntu Edition):

```
static part_t platform_parts[PART_MAX_COUNT] = {
```

```
    {PART_PRELOADER, 0, PART_SIZE_PRELOADER, 0,PART_FLAG_NONE},
    {PART_MBR, 0, PART_SIZE_MBR, 0,PART_FLAG_NONE},
    {PART_EBR1, 0, PART_SIZE_EBR1, 0,PART_FLAG_NONE},
    {PART_PRO_INFO, 0, PART_SIZE_PRO_INFO, 0,PART_FLAG_NONE},
    {PART_NVRAM, 0, PART_SIZE_NVRAM, 0,PART_FLAG_NONE},
    {PART_PROTECT_F, 0, PART_SIZE_PROTECT_F, 0,PART_FLAG_NONE},
    {PART_PROTECT_S, 0, PART_SIZE_PROTECT_S, 0,PART_FLAG_NONE},
    {PART_SECURE, 0, PART_SIZE_SECCFG, 0,PART_FLAG_NONE},
    {PART_UBOOT, 0, PART_SIZE_UBOOT, 0,PART_FLAG_NONE},
    {PART_BOOTIMG, 0, PART_SIZE_BOOTIMG, 0,PART_FLAG_NONE},
    {PART_RECOVERY, 0, PART_SIZE_RECOVERY, 0,PART_FLAG_NONE},
    {PART_SECSTATIC, 0, PART_SIZE_SEC_RO, 0,PART_FLAG_NONE},
    {PART_MISC, 0, PART_SIZE_MISC, 0,PART_FLAG_NONE},
    {PART_LOGO, 0, PART_SIZE_LOGO, 0,PART_FLAG_NONE},
    {PART_EXPDB, 0, PART_SIZE_EXPDB, 0,PART_FLAG_NONE},
    {PART_ANDSYSIMG, 0, PART_SIZE_ANDROID, 0,PART_FLAG_NONE},
    {PART_CACHE, 0, PART_SIZE_CACHE, 0,PART_FLAG_NONE},
    {PART_USER, 0, PART_SIZE_USRDATA, 0,PART_FLAG_NONE},
    {NULL,0,0,0,PART_FLAG_END},
};
```

First Secure Boot stage

After loading all the partitions (and if the feature is enabled), the Preloader initialises the SecLib subsystem. The device vendor supplies an RSA key of up to 2048 bits in length (although the keys I've seen are only 1024 bits).

What SecLib does exactly is unknown. It takes configuration data from the SECURE partition (if it exists) and the RSA key, then calls into the binary blob `mediatek/platform/${platform}/preloader/src/SecLib.a`.

Boot mode selection

After (optionally) confirming secure boot, the Preloader decides which boot mode to use.

`NORMAL_BOOT` will be used if secure boot is disabled or the secure boot module doesn't say otherwise. If Download Mode is enabled, this mode will immediately try to enter it.

There is a long list of other possible boot modes, and not all of them are self-explanatory:

```
typedef enum
{
   NORMAL_BOOT = 0,
   META_BOOT = 1,
   RECOVERY_BOOT = 2,
   SW_REBOOT = 3,
   FACTORY_BOOT = 4,
   ADVMETA_BOOT = 5,
   ATE_FACTORY_BOOT = 6,
   ALARM_BOOT = 7,
#if defined (MTK_KERNEL_POWER_OFF_CHARGING)
   KERNEL_POWER_OFF_CHARGING_BOOT = 8,
   LOW_POWER_OFF_CHARGING_BOOT = 9,
#endif
   FASTBOOT = 99,
   DOWNLOAD_BOOT = 100,
   UNKNOWN_BOOT
} BOOTMODE;
```

Download mode

Before Download mode can be entered, the Preloader has to find out if a host is connected via USB or UART and running the MTK SP Flash Tool. It does this by configuring a virtual CDC ACM discipline on USB, so both lines are in fact serial ports and behave similarly.

The USB port will assume that the tool is connected if it receives a “set line coding” (configures baudrate etc.) CDC message. It then sends the string `READY` to the tool and waits for the reception of a token of eight bytes.

After successful detection, the tool can send the special `Start` command sequence (`0xa0 0x0a 0x50 0x05`) to enter a special mode that is only available via USB. It interprets the following commands (I left the ones marked with “legacy” out):

Command	Command	Function

	byte	
CMD_GET_BL_VER	0xfe	Get Preloader version (seems to be always “1”)
CMD_GET_HW_SW_VER	0xfc	Return hardware subcode, hardware version and software version
CMD_GET_HW_CODE	0xfd	Return hardware code and status
CMD_SEND_DA	0xd7	Send a special “Download Agent” binary to the SoC, signed with a key.
CMD_JUMP_DA	0xd5	Set boot mode to DOWNLOAD_BOOT and start execution of the Download Agent sent in the previous step.
CMD_GET_TARGET_CONFIG	0xd8	Get supported Preloader configuration flags
CMD_READ16	0xa2	Read data from the SoC memory (16 bit length parameter)
CMD_WRITE16	0xd2	Write data into SoC memory (16 bit length parameter)
CMD_READ32	0xd1	Read data from the SoC memory (32 bit length parameter)
CMD_WRITE32	0xd4	Write data into SoC memory (32 bit length parameter)
CMD_PWR_INIT	0xc4	Initialise the power management controller (effectively a null op because it is already on)
CMD_PWR_DEINIT	0xc5	Shut down the power management controller (effectively a null o)
CMD_PWR_READ16	0xc6	Read 16 bits of data from the power management controller interface memory
CMD_PWR_WRITE16	0xc7	Write 16 bits of data to the power management controller interface memory

The Download Agent step is necessary because this way the Flash Tool can always send a current version for the exact hardware version that’s being used.

The UART has no possibility to detect if the physical line is powered, so it just sends the string READY and hopes that it gets an eight byte token back. If it does, it assumes that the tool is present.

Note that the special commands from the table above are *not* available when communicating over the UART, probably because the Boot ROM already offers most of these commands via UART.

If the special Start command is not issued by the host via USB, the Preloader enters a common mode in which it accepts the following commands over both USB and UART:

Command	Command string	Function
SWITCH_MD_REQ	SWITCHMD	Is probably supposed to switch the modem into firmware download mode,

Command	Command string	Function
		but doesn't seem to do anything on the MT6582?
`ATCMD_NBOOT_REQ`	`AT+NBOOT`	Switch to `NORMAL_BOOT` mode
`META_STR_REQ`	`METAMETA`	Switch to `META_BOOT` mode
`FACTORY_STR_REQ`	`FACTFACT`	Switch to `FACTORY_BOOT` mode
`META_ADV_REQ`	`ADVEMETA`	Switch to `ADVMETA_BOOT` mode
`ATE_STR_REQ`	`FACTORYM`	Switch to `ATE_FACTORY_BOOT` mode
`FB_STR_ACK`	`FASTBOOT`	Switch to `FASTBOOT` mode

Second Secure Boot stage

Again it is unknown that SecLib does at this stage, it calls into the binary blob most of the time.

The following (questionable) information was obtained by looking at the C wrapper and dumping the library symbols and strings:

- The security data comes from the `SECSTATIC` partition
- Validation of cryptographic image signatures using RSA/SHA1
- The `UBOOT`, `LOGO`, `BOOTIMG`, `RECOVERY` and `ANDROID` partitions seem to be checked at some point
- The "customer name" seems to be checked somehow, but why?

The necessary signed images are most likely generated by the `SignTool` binaries in `mediatek/build/tools/SignTool`.

The device manufacturer can add additional security measures.

Load core boot images

Now that the Preloader knows that the system is safe and secure, it can go and load the firmware images from the internal flash.

This is a highly specialised process, because every image has to be processed differently. For example the

firmware for the HSPA modem in the MT6582 has to be fed into the modem using special registers and commands, while the u-Boot boot loader can just be copied to the right memory address. In this step the Preloader will also decide which is the next component that gets executed after itself ends. By default this will be the image stored in the `UBOOT` partition.

Note that in this step only the most basic firmware is loaded, this is usually just the modem and the bootloader.

Platform post init

In this step the platform is put into a defined state for the next boot process component (bootloader, Little Kernel). The most important step is to pass on the boot arguments that were set during Preloader execution. This will hopefully make more sense once we look at what happens after the Preloader, the whole MediaTek design is a bit complicated.

The boot argument structure on the MT6852 looks like this:

```
typedef struct {
    u32 magic;              // 0x504c504c
    boot_mode_t mode;       // Boot mode
    u32 e_flag;
    u32 log_port;
    u32 log_baudrate;
    u8  log_enable;
    u8  part_num;           // Number of partitions
    u8  reserved[2];
    u32 dram_rank_num;
    u32 dram_rank_size[4];
    u32 boot_reason;        // Boot reason
    u32 meta_com_type;
    u32 meta_com_id;
    u32 boot_time;
    da_info_t da_info;
    SEC_LIMIT sec_limit;
    part_hdr_t *part_info;  // Partition info
    u8  md_type[4];
    u32 ddr_reserve_enable;
```

```
    u32 ddr_reserve_success;
    u32 chip_ver;
} boot_arg_t;
```

It is put at a defined memory location, where it “survives” until the next component grabs it.

Boot the next component

The last Preloader step is to jump to the location of the next component, usually the “Little Kernel” loaded from the `UBOOT` partition.

If you know better and/or something has changed, please find me on Launchpad.net or the Freenode IRC and do get in contact!

Resources

- Thunder-Kernel
- MTK6577 Android Start —- pre-loader (in chinese)
- Introduction of MTK Tools
