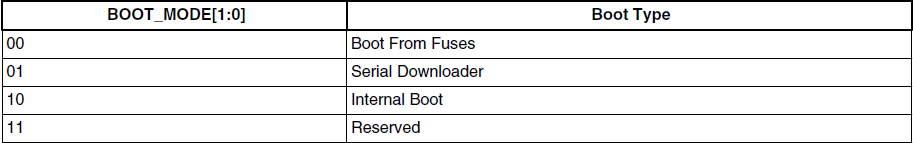
# Boot mode



板子上面有两个用来配置boot mode的GPIO，reset后SBMR2寄存器会加载这个值。ROM根据SBMR2[25：24]来判断是什么mode。

0x400f801c

SBMR1用来加载boot device类型

0x400f8004

## Boot from fuse (0x00)

芯片内部有fuse阵列，可以理解为只可烧写一次的存储器。一旦BT\_FUSE\_SEL 烧写为1，同时配置boot device 相关的fuse。ROM就只能通过fuse的值来获取boot device类型

## Internal boot (0x10)

如果BT\_FUSE\_SEL不烧写为1，ROM会根据板子上面的GPIO拨码来确定boot device的类型

## Serial download (0x01)

进入这个模式后，板子可以使用SDP协议与ROM进行通讯。我们有一个SDP的上位机，用来跟ROM进行通信。

PS：

1. 如果boot 失败，ROM 默认进到serial download 模式

2. mscale的板子如果拨到这个模式，只有manufacturing boot 失败才会进到这个模式。

## Test mode (0x11)

目前没用到

# 几个重要的函数

## pu\_irom\_setup\_boot\_selection()

初始化boot device的属性，设置image 存储地址，image initial address，IVT address，secondary boot的支持，设备的初始化、读等功能。具体请参考[Boot device](#_Boot_device)

## Download image

1. 执行download initial image

2. 如果存在IVT，则根据IVT将boot image拷贝到运行地址。认证pass后返回image entry

3. 如果IVT中开启了plug in， 则进行plug in boot，返回plug in image的entry

## Download initial image

1. 对 boot device进行初始化

2. 如果boot device的 driver\_data->device\_read\_data != NULL， 则

ret\_val = driver\_data->device\_read\_data(driver\_data->initial\_image\_address,

driver\_data->initial\_image\_size,

driver\_data->image\_offset);

简单来说，就是将iamge存储地址处0x1000字节的数据读取到RAM中

3. 如果支持secondary boot，就会从2中读出secondary boot的table，然后根据table中的first sector number重新指定image的存储地址。然后再从新的image存储地址中执行2中的动作将0x1000长度的数据拷贝到RAM中

rom\_memcpy((UINT32\*)&secondary\_image\_table,

(UINT32\*)((UINT32)driver\_data->initial\_image\_address + SECONDARY\_IMG\_TBL\_OFFSET),

sizeof(SECONDARY\_IMG\_TBL\_T));

/\* Update image offset to value from secondary image table \*/

driver\_data->image\_offset = secondary\_image\_table.firstSectorNumber;

/\* Add device read data log entry \*/

pu\_irom\_log\_and\_data\_add(ROM\_LOG\_DEVICE\_READ\_DATA\_CALL, driver\_data->image\_offset);

ret\_val = driver\_data->device\_read\_data(driver\_data->initial\_image\_address,

driver\_data->initial\_image\_size,

driver\_data->image\_offset);

## pu\_irom\_hwcnfg\_setup

运行地址为IVT 中boot data中的start address

Image 长度为boot data中的image length

此函数的目的是为了将image 拷贝到运行地址。

对nand 、 sd 、 emmc而言，在image size > driver\_data->initial\_image\_size时会分两次copy。 第一次是copy initial\_image\_size长度的数据到运行地址，第二次将image size - initial\_image\_size长度的数据copy到运行地址

对nor上面的code，如果运行地址不在nor上面，则一次性将整个image copy到运行地址

len\_to\_copy = driver\_data->device\_read\_data == NULL ? length : // Copy them all

length > driver\_data->initial\_image\_size ? driver\_data->initial\_image\_size : length;

TRACE("BootROM: to copy 0x%x bytes from 0x%x to 0x%x\n", len\_to\_copy,

driver\_data->initial\_image\_address, dst\_pointer);

if (dst\_pointer != driver\_data->initial\_image\_address)

{

rom\_memcpy(dst\_pointer, driver\_data->initial\_image\_address, len\_to\_copy);

}

# Boot device

Boot device 是指boot image存储的媒介。

BootROM 会根据GPIO或者fuse的值，来确定是何种boot device，然后来进行boot

如下是RT1020 ROM对各个device的初始化参数

## FlexSPI Nor

driver\_data->device\_init = (device\_init\_f) &bl\_flexspi\_nor\_init;

driver\_data->device\_read\_data = NULL;

driver\_data->initial\_image\_size = (248U\*1024\*1024-4);

driver\_data->initial\_image\_address = (UINT32\*)0x60000000;

driver\_data->image\_offset = 0;

driver\_data->device\_page\_size = 1;

driver\_data->ivt\_offset = 0x1000;

driver\_data->use\_secondary\_image = FALSE;

driver\_data->device\_log\_entry = ROM\_LOG\_PRIM\_BOOTDEVICE\_FLEXSPI\_NOR; //0x00060008

## FlexSPI nand

driver\_data->device\_init = (device\_init\_f)&bl\_flexspi\_nand\_boot\_init;

driver\_data->device\_read\_data = (device\_read\_data\_f)&bl\_flexspi\_nand\_boot\_download\_firmware;

driver\_data->initial\_image\_size = 4096;

driver\_data->initial\_image\_address = IRAM\_FREE\_SPACE\_START;//0x20208000

driver\_data->image\_offset = 0;

driver\_data->device\_page\_size =2048;

driver\_data->ivt\_offset = 0x400;

driver\_data->use\_secondary\_image = FALSE;

driver\_data->device\_log\_entry = ROM\_LOG\_PRIM\_BOOTDEVICE\_FLEXSPI\_NAND;// 0x00060009

driver\_data->image\_offset会根据FCB里面指定的image重新赋值

nand的读取操作都是以页为单位

## EMMC and SD

driver\_data->device\_init = &card\_init;

driver\_data->device\_read\_data = &card\_data\_read;

driver\_data->initial\_image\_size = CARD\_IROM\_INIT\_IMAGE\_SIZE; //4096

driver\_data->initial\_image\_address = (UINT32\*)CARD\_IROM\_INIT\_IMAGE\_BASE;//0x20208000

driver\_data->image\_offset = 0;

driver\_data->device\_page\_size = CARD\_BLK\_LEN;//0x200

driver\_data->ivt\_offset = 0x400;

driver\_data->use\_secondary\_image = TRUE;

driver\_data->device\_log\_entry = ROM\_LOG\_PRIM\_BOOTDEVICE\_USDHC; //0x00060001

## SEMC nand

driver\_data->device\_init = (device\_init\_f)&bl\_semc\_nand\_boot\_init;

driver\_data->device\_read\_data = (device\_read\_data\_f)&bl\_semc\_nand\_boot\_download\_firmware;

driver\_data->initial\_image\_size = 4096;

driver\_data->initial\_image\_address = 0x20208000;

driver\_data->image\_offset = 0;

driver\_data->device\_page\_size = 2048;

driver\_data->ivt\_offset = 0x400;

driver\_data->use\_secondary\_image = FALSE;

driver\_data->device\_log\_entry = ROM\_LOG\_PRIM\_BOOTDEVICE\_SEMC\_NAND;// 0x0006000B

## SEMC Nor

driver\_data->device\_init = (device\_init\_f) &bl\_semc\_nor\_init;

driver\_data->device\_read\_data = NULL;

driver\_data->initial\_image\_size =16U\*1024\*1024;

driver\_data->initial\_image\_address = (UINT32\*)0x90000000;

driver\_data->image\_offset = 0;

driver\_data->device\_page\_size = 1;

driver\_data->ivt\_offset = 0x1000;

driver\_data->use\_secondary\_image = FALSE;

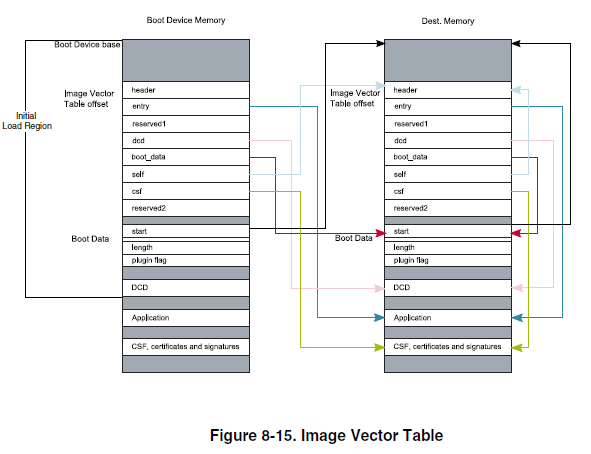
driver\_data->device\_log\_entry = ROM\_LOG\_PRIM\_BOOTDEVICE\_SEMC\_NOR;// 0x0006000A

# IVT

需要给application包装一个IVT，才能够boot。

ROM根据IVT找到application，并进行boot。

关于如何制作带IVT的image以及签名跟加密的image，请参照另一篇文档



# Boot flow

## Primary boot

Primary boot是ROM boot的第一道流程，如果primary boot成功后，就不会有recovery boot跟manufacturing boot了。

根据不同的device 还存在其他的boot 方式，比如nand中有redundant boot，SD中有secondary boot等等。

### FLEXSPI nand

Nand 的特性是按页写，按块擦。

#### Search count 、search stride

这两个值只能从fuse中获取，在FCB中定义的无效。

Search count是查找FCB跟DBBT最大次数

Search stride是查找FCB跟DBBT的页步长

FCB跟DBBT的查找都是在nand init时候做的，init成功后才进行数据的读取操作

#### FCB

FCB默认从page0开始查找。

如果search count设置为1,。则ROM再给一次机会查找FCB。查找的起始page为fuse中定义的search stride的长度

1 block = 64 page

1 page data size = 2048

1page total size = 4096

FCB 的结构如下

typedef struct \_firmware\_info

{

uint32\_t startPage;

uint32\_t pagesInFirmware;

}firmware\_info\_t;

typedef struct \_FlexSpiFCBStruct\_t

{

uint32\_t crcChecksum; //!< [0x000-0x003]

uint32\_t fingerprint; //!< [0x004-0x007]

uint32\_t version; //!< [0x008-0x00b]

uint32\_t DBBTSerachAreaStartPage; //!< [0x00c-0x00f]

uint16\_t searchStride; //!< [0x010-0x011] Not used by ROM

uint16\_t searchCount; //!< [0x012-0x013] Not used by ROM

uint32\_t firmwareCopies; //!< [0x014-0x017]

uint32\_t reserved0[10]; //!< [0x018-0x03f]

firmware\_info\_t firmwareTable[FIRMWARE\_MAX\_NUM];//!< [0x040-0x07f]

uint32\_t reserved1[32]; //!< [0x080-0x0ff];

flexspi\_nand\_config\_t config; //!< [0x100-0x2ff];

uint32\_t reserved2[64]; //!< [0x300-0x3ff];

}flexspi\_fcb\_t;

FCB指定了nand中image的存储位置

ROM会从page0 开始查找FCB，查找search count次，每次查找的步长为search stride。

status\_t bl\_flexspi\_fcb\_search(uint32\_t readCount, uint32\_t searchStride, uint32\_t \*buffer)

{

uint32\_t fcbStartPage = 0;

status\_t status;

uint32\_t retryCnt;

for (retryCnt = 0; retryCnt < readCount; retryCnt++)

{

TRACE("BootROM: Try to read FCB from page: %x\n", fcbStartPage);

status = bl\_flexspi\_nand\_read\_page(fcbStartPage, buffer, sizeof(s\_flexspiFcb));

if (status == kStatus\_Success)

{

// Do crc check here

// if crc check fails, switch to next FCB

if (flexspi\_nand\_fcb\_check())

{

TRACE("BootROM: FCB found at page %x\n", fcbStartPage);

return kStatus\_Success;

}

}

fcbStartPage += searchStride;

}

TRACE("BootROM: FCB not found\n");

return kStatus\_Fail;

}

FCB查找到之后，nand会根据FCB里面定义的config block对nand重新进行配置

flexspi\_nand\_config\_t config; //!< [0x100-0x2ff];

#### DBBT

如果DBBT中将FCB指定的image block标记为坏块，比如将block9标记为坏块。

那么driver\_data->image\_offset会变成10，因此ROM会从block 10查找image

typedef struct \_FlexspiDBBTStruct

{

uint32\_t crcChecksum; //!< [0x000-0x003]

uint32\_t fingerprint; //!< [0x003-0x007]

uint32\_t version; //!< [0x008-0x00b]

uint32\_t numberOfPages; //!< [0x00c-0x00f]

uint32\_t badBlockNumber; //!< [0x010-0x013]

uint32\_t reserved1[3]; //!< [0x014-0x01f]

uint32\_t badBlockTable[SERIAL\_NAND\_BAD\_BLOCKS\_MAX\_NUM]; //!< [0x020-0x41f]

}flexspi\_dbbt\_t;

查找DTTB , Start page是在FCB中定义的

// Search DBBT

memset(&s\_flexspiDBBT, 0, sizeof(s\_flexspiDBBT));

if (s\_flexspiFcb.DBBTSerachAreaStartPage)

{

dbbtSearchStride = searchStride;

dbbtSearchCount = searchCount;

status = bl\_flexspi\_dbbt\_search(s\_flexspiFcb.DBBTSerachAreaStartPage, dbbtSearchCount, dbbtSearchStride);

if (status != kStatus\_Success)

{

return false;

}

}

status\_t bl\_flexspi\_dbbt\_search(uint32\_t startPage, uint32\_t readCount, uint32\_t searchStride)

{

uint32\_t dbbtStartPage;

uint32\_t retryCnt;

status\_t status;

dbbtStartPage = startPage;

for (retryCnt = 0; retryCnt < readCount; retryCnt++)

{

TRACE("BootROM: SPI NAND read, blockId=%x\n", dbbtStartPage / s\_flexspiNandCfg.pagesPerBlock);

TRACE("BootROM: SPI NAND read, pageId=%x\n", dbbtStartPage);

status = bl\_flexspi\_nand\_read\_page(dbbtStartPage, (uint32\_t\*)&s\_flexspiDBBT, sizeof(s\_flexspiDBBT));

if (status == kStatus\_Success)

{

if (flexspi\_nand\_dbbt\_check())

{

TRACE("BootROM: DBBT found at page %x\n", dbbtStartPage);

return kStatus\_Success;

}

}

dbbtStartPage += searchStride;

}

TRACE("BootROM: DBBT not found\n");

return kStatus\_Fail;

}

#### Boot flow

Nand根据FCB跟persist寄存器，找到合法的image后。会将image的前4K数据读取到0x20208000处，然后从IVT里面将取出boot data。根据boot data里面指定的地址，将application拷贝到应该运行的地址。然后boot application

#### Redundant boot

如果FCB中指定了多个image，也就是firmwareTable中制定了image多个存储位置。

配合persist[31:30] 0x400f8044 寄存器，ROM回去找可以boot 的image。默认persist 寄存器的值是0

也就是说，如果firmwareTable[0]里面存放了合法的image，且image存放的地方ROM可以读取，那么persist此时的值为0，image boot起来。

如果firmwareTable[0]对应的page是坏块，或者里面存放了不能boot的image。那么persist register会变成1，同时会从firmwareTable[1]中找image

依次类推，给三次机会。

### FLEXSPI nor

Nor boot会从0x60000000处读取0x200数据进行nor初始化。

然后根据boot data找到application的start address。

#### 非XIP boot

ROM从bootdata里面取出start address跟device初始化的地址进行对比，如果不相等的话就会将image拷贝到bootdata里面指定的地址，然后boot

if (dst\_pointer != driver\_data->initial\_image\_address)

{

rom\_memcpy(dst\_pointer, driver\_data->initial\_image\_address, len\_to\_copy);

}

#### XIP boot

如果两个地址相等，就直接从nor进行boot

### SD/MMC

#### Normal boot

制作好带IVT的image烧写到SD或者EMMC中driver\_data->image\_offset处，ROM会读取前4K数据到0x20208000中，然后根据IVT中boot data找到image的运行地址，然后将application拷贝到运行地址执行。

#### Secondary boot

如果在driver\_data->image\_offset + 0x200处存在如下的结构体

typedef struct

{

UINT32 chipNum; /\* Chip Select, ROM does not use it \*/

UINT32 driveType; /\* Always system drive, ROM does not use it \*/

UINT32 tag; /\* Tag, must be 0x584D2E69 (i.MX) \*/

UINT32 firstSectorNumber; /\* Block address of secondary image, block size is 512B \*/

UINT32 sectorCount; /\* Not used by ROM \*/

} SECONDARY\_IMG\_TBL\_T;

然后将normal boot的image存放到driver\_data->image\_offset + 0x200\* firstSectorNumber处。此时normal boot失败，persist register就会变成1，同时根据如上的结构体找到image，进行boot。

总而言之，secondary boot就是将image的存储地址通过一个结构体指向了另一个地方而已。

举个例子，以mScale EMMC为例：（driver\_data->image\_offset为0x8000）

SD\EMMC 一个block的长度为0x200

Tag ： 0x00112233

FirstSectorNumber ：

EMMC的image start address = 0x8000 IVT offset = 0x400

Table address = 0x200

设定secondary start address为0x9000，则firstSectorNumber为：

(0x9000-0x8000) /0x200 = 8

将制作好的table烧写到0x8200， image烧写到0x9000.reset板子后persist register变为1，secondary image就boot成功了

#### Fast boot

以EMMC为例，首先EMMC必须支持fast boot

然后gpio或者fuse将emmc设置为fast boot模式，然后对EMMC进行bootbus 跟part config的配置，使EMMC跟fuse或者GPIO的配置对应。此时的boot 就叫做fast boot。

Fast boot， 不同板子image的烧写地址可能不一样。

以mScale为例，如下是fast boot的一个case

##### fast boot, 8 bit ddr, high speed, acknowledge open

1. using Uboot to configure the boot fuse

fuse prog –y 1 3 100020e4

fuse prog –y 2 2 1

2. choose EMMC, erase the first 0x200 block(1 block = 0x200 bytes.)

mmc dev 0x0 0x0

mmc erase 0x0 0x200

3. open x mode for transmitting the demo

loadx

4. using UART to transmit demo to DDR in x mode.

click transmit -> transmit file (using x mode) -> choose demo ->click transmit

5. write demo from DDR to EMMC 0x400 (demo is start from IVT so we must write the demo to 0x8400 , block = 0x400/0x200=0x2)

mmc write 0x40080000 0x2 0x200

6. configure EMMC to meet the fuse setting

mmc partconf 0 1 7 0

mmc bootbus 0 2 0 2

7. set boot mode from fuse and re-power the board, demo boot successfully.

## Plugin boot

Plugin boot是指IVT中plugin的flag打开后。上述ROM支持的device中有一个能够正常boot的image，这个image的功能是指定一个APP的base address，同时指定IVT offset跟length。

ROM从如上指定的base 跟IVT offset进行image的认证，然后boot。

总结下来，我们需要两个image：

1. 存储于ROM支持的device中的一个能够boot的image，这个image的功能需要遵循一定的[规则](#_plugin_function)。目的是告诉ROM一个新的base address，新的IVT offset，image length

boot\_data = (BOOT\_DATA\_T\*)data\_ptr->boot\_data;

if (PLUGIN\_IMAGE(boot\_data))

{

hab\_image\_entry = pu\_irom\_run\_plugin((plugin\_download\_f)hab\_image\_entry);

}

2. ROM 从1中指定的base address处找APP，认证后进行boot

static void\* pu\_irom\_run\_plugin(plugin\_download\_f plugin\_download)

{

ENABLE\_STACK\_PROTECTION();

void\* hab\_image\_entry = NULL;

void \*start = NULL;

UINT32 bytes = 0;

UINT32 ivt\_offset = 0;

hab\_status\_t hab\_status = HAB\_FAILURE;

hab\_status = hab\_rvt.exit();

/\*

\* Search for HAB\_FAILURE event in log buffer.

\*/

check\_pass\_fail();

if(hab\_status != HAB\_SUCCESS)

{

boot\_failure\_indicator();

/\* As stated earlier, the norm is if entry or exit fails we should go into failsafe mode, control

will not return back instead this will result in ROM going into serial

downloader mode \*/

hab\_rvt.failsafe();

}

/\* Add plugin image call log entry \*/

pu\_irom\_log\_and\_data\_add(ROM\_LOG\_PLUGIN\_IMAGE\_CALL, (UINT32)plugin\_download);

// Disable MPU in order to allow plugin image to be executed.

pu\_irom\_mpu\_disable();

if (plugin\_download(&start, &bytes, &ivt\_offset) == TRUE)

{

// Renable MPU after plugin being executed.

pu\_irom\_mpu\_setup();

/\* Add plugin image call pass log entry \*/

pu\_irom\_log\_add(ROM\_LOG\_PLUGIN\_IMAGE\_PASS);

/\* Initialize hab\_status to failure before its second use \*/

hab\_status = HAB\_FAILURE;

/\* Call HAB entry \*/

hab\_status = hab\_rvt.entry();

/\*

\* Search for HAB\_FAILURE event in log buffer.

\*/

check\_pass\_fail();

if(hab\_status != HAB\_SUCCESS)

{

boot\_failure\_indicator();

/\* As stated earlier, the norm is if entry or exit fails we should go into failsafe mode, control

will not return back instead this will result in ROM going into serial

downloader mode \*/

hab\_rvt.failsafe();

}

hab\_image\_entry = hab\_rvt.authenticate\_image(HAB\_CID\_ROM, ivt\_offset, &start, &bytes, NULL);

/\*

\* Search for HAB\_FAILURE event in log buffer.

\*/

check\_pass\_fail();

/\* Add authentication status log entry \*/

pu\_irom\_log\_and\_data\_add(ROM\_LOG\_AUTHENTICATION\_STATUS,

(UINT32)((pu\_irom\_error\_status.reserved<<24) |

(pu\_irom\_error\_status.context <<16) |

(pu\_irom\_error\_status.reason<<8) |

(pu\_irom\_error\_status.status)));

}

else

{

/\* Add plugin image call fail log entry \*/

pu\_irom\_log\_add(ROM\_LOG\_PLUGIN\_IMAGE\_FAIL);

}

return hab\_image\_entry;

### plugin function

typedef BOOLEAN (\*plugin\_download\_f)(void \*\*start, size\_t \*bytes, UINT32 \*ivt\_offset);

## Recovery boot

如果板子支持EEPROM，则存在此种boot，目前没有涉及到

## Manufacturing boot

如上都boot fail后，板子会默认从SD卡boot。

## Serial download

如果boot失败或者boot mode设置为serial download模式，会进入serial download模式。

在这个模式下，我们可以通过SDP host tool跟ROM进行通信。

SDP tool支持对寄存器的读写，以及可以烧写一个image到RAM中，并且boot这个image。如果板子处于HAB close模式，则image也必须为签名的image。