目录

[简介 1](#_Toc2168460)

[Key management 1](#_Toc2168461)

[Log buffer 1](#_Toc2168462)

[Mailbox 2](#_Toc2168463)

[Mailbox架构图 2](#_Toc2168464)

[Mailbox寄存器 2](#_Toc2168465)

[Debugger访问mailbox示意图 3](#_Toc2168466)

[ROM Mailbox Handler功能解析 4](#_Toc2168467)

[Debug Authentication 6](#_Toc2168468)

[原理概述 6](#_Toc2168469)

[实例演示 8](#_Toc2168470)

# 简介

i.MX RT600是跨界处理器系列产品，适用于32位沉浸式音频播放和视频用户界面应用，将高性能Cadence® Tensilica® Hi-Fi 4音频DSP内核与新一代Cortex-M33内核相结合。RT600系列跨界处理器旨在通过安全、功率优化的嵌入式处理器充分挖掘语音辅助终端节点的潜力。

i.MX RT600提供高达4.5MB的片上SRAM和多个高带宽接口，可访问片外闪存，包括带OTF解密引擎的Octal/Quad SPI接口。此外，该系列提供丰富的外设、高级安全功能，功耗极低，支持低功耗模式和快速唤醒功能。

* 支持多种外设：

1. Octal/Quad SPI nor flash
2. SD/EMMC
3. USB
4. 8个可配置通用串行接口模块（Flexcomm接口），可配置为SPI/I2C/I2S/UART

* 安全功能是ROM的一大亮点,

1. 支持ARM Trustzone
2. 支持物理不可克隆秘钥生成模块（PUF）
3. 支持Debug authentication
4. 支持认证boot，加密boot，在线解密OTFAD

* 产品的目标应用市场为工业控制中的机器学习设备，以及消费电子中的语音识别设备

ROM是产品上电后第一个执行的代码，它的功能是引导用户application的启动。

从用户的角度来看，ROM主要做了两点保护：

1.Boot的保护，非法的image不给boot

2.Debug的保护，用户配置禁用芯片debug功能后需要使用debug authentication来打开

本文介绍RT600的ROM的一些主要功能，以及如何使用这些功能

# ROM主要模块介绍

## Key management

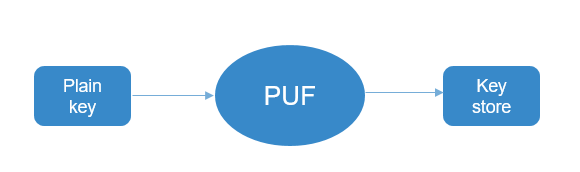
对秘钥的管理，目的就是让秘钥更加的安全。简言之就是让黑客无法获取正确的秘钥

RT600提供两种秘钥管理方式，PUF跟OTP

### PUF

PUF是一个第三方公司的IP，秘钥通过PUF处理后生成key store。解密时，通过PUF处理key store得出秘钥，从而实现解密。

这个过程中，秘钥的体现形式就是key store。由于PUF的安全性，黑客无法破解key store得到秘钥。



### OTP key management

OTP就是一块可烧写的fuse。秘钥经过不同的算法后生成加密的key，用于不同场景的加密。

ROM在hardware\_init的时候会打开OTP scramble，向fuse中烧写秘钥时OTP controller 会将写入的数值scramble然后后存放到fuse中。所以读取的master key永远是scramble后的数据.

秘钥会以明文的形式提供给AES engine，ROM的解密就是让AES engine从OTP读取plain key进行解密操作。

所以对于用户而言看到的永远是scramble后的数据，但是对于解密模块AES engine而言，他获取的是plain key

RT600的fuse中增加了SCRAMBLE\_INPUT的fuse，其目的是通过这个word的数据随机化scramble的算法。使用的时候应该先烧写这个fuse，reset后再烧写master key



#### 1.2.1 使用master key生成不通用途的encrypted master key

1. *HMAC\_SHA256 key128*:  This is used for image header authentication in LoadToRam images.
   * AES256(MASTERKEY256, 00000000\_00000000\_00000000\_00000000);
2. *Encrypted boot image key256*: Key used to decrypt encrypted images during boot.
   * AES256(MASTERKEY256, 01000000\_00000000\_00000000\_00000000\_02000000\_00000000\_00000000\_00000000);
3. *SB2 KEK key* or FW update Key: Key used as key encryption key to handle SB2 file (update capsule).
   * AES256(MASTERKEY256, 03000000\_00000000\_00000000\_00000000\_04000000\_00000000\_00000000\_00000000);
4. OTFAD\_KEK128 (supports both per-device AES key and pre-shared AES key)
   * AES256\_ENCRYPT (MASTERKEY256, OTFAD\_key[127:0] fuse );

## ROM hiding

ROM hiding 字面意思就是将ROM的一些重要代码隐藏起来，所谓的隐藏就是不能读跟执行。

ROM在编译的时候会将需要隐藏的代码存放到ROM的前16KB处，即0x13000000-0x13004000.

#### 2.1 ROM如何隐藏代码：

SYSCON->BOOT\_LOCK\_ADDR\_DP = 0x4000/0x4 - 1; //0x4000\_0484配置lock region的大小

SYSCON->LockOut = 0xc33ca55a; //0x4000\_0488使能lock

ROM执行hide的流程：

上电后，ROM执行，如果运行到ISP里，不会hide ROM。此时debug无法使用，无法读取ROM 前16K内容。

何时hide ROM，有如下几种情况

1. Mailbox enable debug

2. Boot

3. 进入FA mode

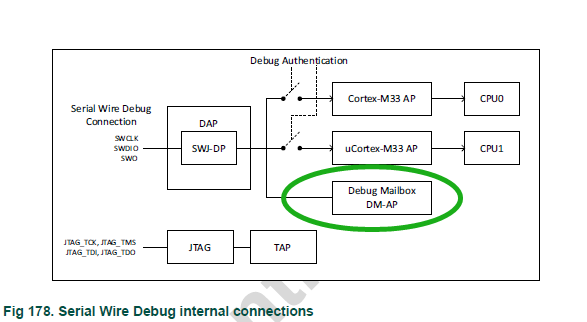
4. Execute

5. Debug authentication成功

## Mailbox

Mailbox是NXP LPC系列产品新推的一项功能。Debugger可以通过mailbox与ROM通信，实现Flash Erase、Enter ISP mode、Set FA mode、Debug authentication等功能。

### 3.1 Mailbox架构图

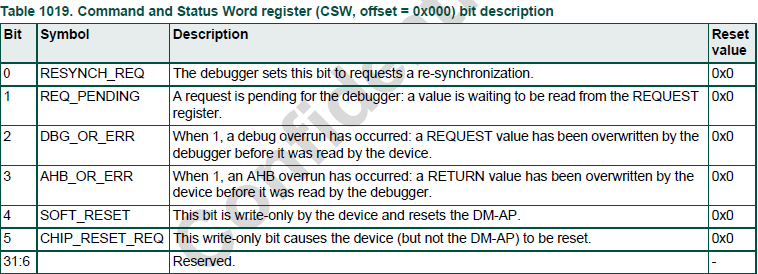


### 3.2 Mailbox寄存器

Mailbox有三个寄存器

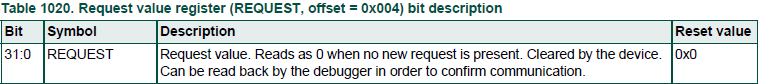
#### 3.2.1 CSW(Command status and Word register)

Debugger可以通过这个寄存器指示ROM进到mailbox handler，ROM、Debugger可以通过这个寄存器获知通信状况。



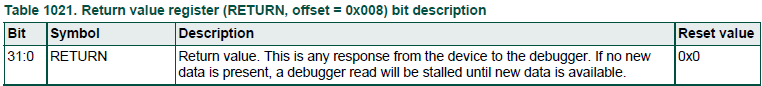
#### 3.2.2 Request register

Debugger向这个寄存器写入数据，ROM从该寄存器中获取数据执行相应的操作。



#### 3.2.3 Return register

ROM向此寄存器写入数据，debugger可以从该寄存器获取ROM的信息。

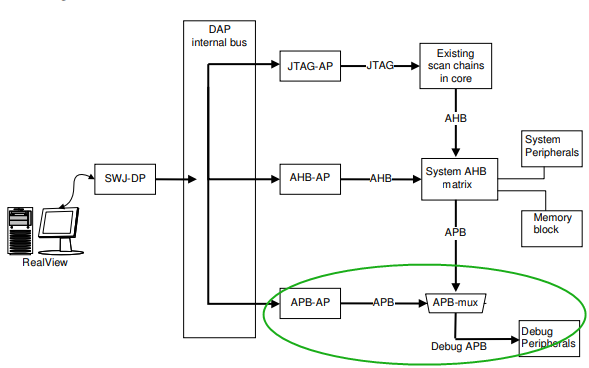


### 3.3 Debugger访问mailbox示意图

Debugger访问APB-AP，实现跟Mailbox的通信

AP: Access Port。 绿色圈内所示连接到Mailbox

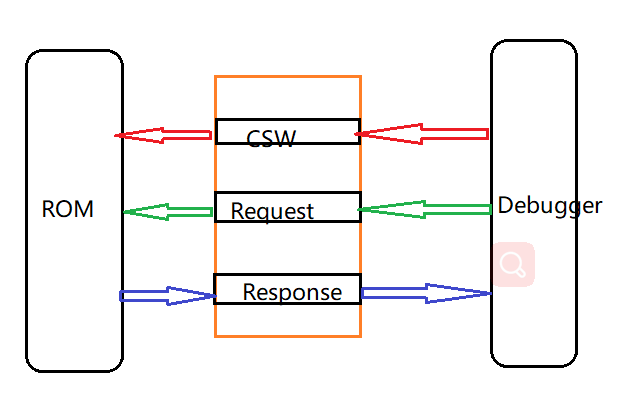
DP: Debug Port。 就是我们的Jlink



### 3.4 ROM Mailbox Handler功能解析

#### 3.4.1 Request/Response机制

Mailbox基于Request/Response机制，如下图：



1. Debugger向CSW写入0x21，申请re-sync同时reset
2. ROM检测CSW re-sync被置1，进入mailbox handler，向response寄存器写入0x0.等待debugger的请求。
3. Debugger向request寄存器写入数据
4. ROM获取request寄存器中的数据，根据不同的指令进行相应的处理(比如擦除flash)，向response寄存器中写入数据。
5. Debugger可以根据response寄存器中的数据判断指令是否执行成功

#### 3.4.2 ROM支持的mailbox指令

* START\_DBG\_MB - 0x0001
* GET\_CRP\_LEVEL - 0x0002, Deprecated ans return 3
* ERASE\_FLASH - 0x0003,
* EXIT\_DBG\_MB - 0x0004,
* ENTER\_ISP\_MODE - 0x0005,
* SET\_FA\_MODE - 0x0006,
* DBG\_AUTH\_START - 0x0010
* DBG\_AUTH\_RESP - 0x0011

#### 3.4.3 Debugger访问Mailbox流程

以进入ISPmode为例

1. Debugger选择APB-AP，访问mailbox
2. Debugger向CSW写入0x21，申请re-sync同时reset
3. ROM检测到re-sync信号，reset mailbox，进入mailbox handler，等待debugger的指令
4. Debugger发送0x10005至request寄存器(如上如示ENTER\_ISP\_MODE = 0x5，高16bit为1表示通知ROM接下来还要发送一个参数，ROM根据这个参数使用相应的peripheral进入ISP)
5. ROM读取request寄存器中数据，发现需要一个参数，向reponse寄存器中写入0x1A5A5。告知debugger其已经获知需要一个参数
6. Debugger向request寄存器发送0x7(Use UART\SPI\I2C进入ISP mode)
7. ROM检测SOCU是否使能了ISPCMDENROM，如果是则向response写入0x0然后根据参数0x7进入ISP mode。否则向response写入ERR\_DM\_UNKNOWN\_CMD

## Debug Authentication

### 4.1原理概述

Debug authentication是RT600推出的一项关于debug保护的技术，其目的是控制对芯片debug的权限。当用户烧写fuse开启debug authentication功能后，用户通过debugger发送正确的认证信息，device（ROM）通过认证才会打开debug。

Debug authentication实现的硬件基础是[mailbox](#_Mailbox)，debugger跟device（ROM）可以通过mailbox这个IP进行信息交互。

#### 4.1.1 Debug authentication流程

1. Debugger向CSW寄存器写入0x21请求ROM进入mailbox handler
2. Debugger发送0x1初始化mailbox handler
3. Debugger 发送0x10请求开始debug authentication
4. ROM接收到0x10命令后，从设备里获取信息填入DAC后逐条发给debugger
5. Debugger接受到DAC后，将DAC内容与本地端数据进行比对（本地端数据是事先准备好进行debug authentication的数据，这些数据需要跟芯片内烧写的数据一致）。如果不同则fail
6. Debugger端构造DC，使用芯片内某个公钥对应的私钥进行签名，签名信息存放于DC的末尾。DC\_sign = SIGN（rotPriv，HASH(version::…::rotpub)）。（进行HASH的对象为除了DC的signature以外的全部数据）

Debugger端构造DAR，使用DCK\_Priv对DAR进行签名，签名信息存放于DAR结构体的末尾。DAR\_sign = SIGN(dckPriv, HASH（DC，AB， DAC[CV]）。如果使用2048长度的RSA公钥，DAR数据长度为0x12C

1. Debugger发送0x012C0011命令（2048 key length），ROM检测到此命令后获知debugger即将发送0x12C长度的数据并申请对数据进行认证
2. ROM逐条接收DAR，比对DC中的各个参数与device中是否一致。使用rotPub对DC进行认证，使用dckPub对DAR进行认证。全部通过后打开debug。
3. Debugger发送0x4申请退出mailbox，debugger就可以连上板子进行调试了

#### 4.1.2 DAC (debug authentication challenge)

Debugger端发送0x10请求认证后。ROM 发给debugger的数据，数据结构体如下，

长度为0x1A（fuse中设置的key length=2048，文中关于debug authentication都是以2048长度key为例）

CV是调用TRGN生成的随机数，参与DAR signature的构造。

其余的信息从板子中获取

typedef struct \_\_attribute\_\_ ((packed)) debug\_auth\_challenge {

debug\_auth\_version\_t version; //1 word

uint32\_t socc; //1

uint32\_t device\_uuid[4]; //4

debug\_rotid\_t rotid; //1 + 32/4 = 9

//revoke fuse的值，RoTK fuse值

uint32\_t cc\_socu\_pinned; // 1

uint32\_t cc\_socu\_default; //1

uint32\_t cc\_vu; // 1

uint8\_t challengeVector[32]; //32/4 = 8 also called CV

} debug\_auth\_challenge\_t;

#### 4.1.3 DAR (debug authentication response)

Debugger端发送0x012C0011后，debugger需要传送给device的数据，数据结构体如下。

使用2048长度的key，response结构体大小为0x12C

typedef struct \_\_attribute\_\_ ((packed)) debug\_auth\_response\_2048 {

debug\_credential\_2048\_t credential;

//1+1+4+32+65+1+1+1+65+64 = 235, also called DC

uint32\_t auth\_beacon; // 1 also called AB

uint8\_t signature[256]; // 256/4 = 64

//SIG(DCK\_Priv, HASH(DC::AB::DAC[CV]))

} debug\_auth\_response\_2048\_t;

##### 4.1.3.1 DC(debug credential)

typedef struct \_\_attribute\_\_ ((packed)) debug\_credential\_2048 {

debug\_auth\_version\_t version; //1

uint32\_t socc; // 1

uint32\_t device\_uuid[4]; // 4

uint8\_t rot\_meta[128]; // 128/4=32

//此值为四个根公钥的hash，再hash一次就是存储在fuse中的Rotk

ROM接收到DAR的数据后会对此值进行一次hash，然后跟板子中的Rotk进行对比

rsa\_public\_key\_2048\_t dckPub; // 256/4+1 = 65

uint32\_t cc\_socu; //1

uint32\_t cc\_vu; //1

uint32\_t credential\_beacon; //1

rsa\_public\_key\_2048\_t rotPub;//256/4+1 = 65

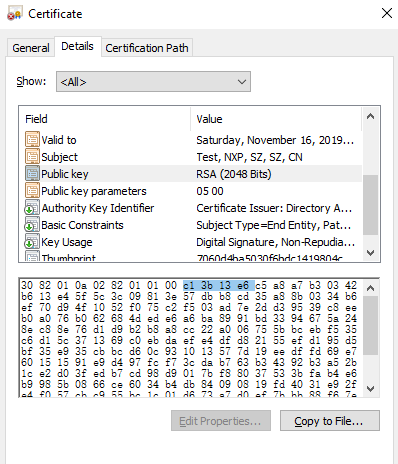
//debugger会根据revoke的情况从0-3就近选择可用的一个根公钥，可以从私钥对应的证书中查看公钥信息。

ROM会对此值进行一次hash，在rot\_meta中查找。如果存在则证明公钥是正确的

uint8\_t signature[256]; //256/4=64

//signature is calculated by SIG(rotPriv, HASH(version::socc: …::rotPub)) 使用rotPub对应的私钥对此结构体中除了签名部分以外的数据进行签名

} debug\_credential\_2048\_t;



证书中的public key。从第10个byte开始，到倒数第六个byte结束

typedef struct \_\_attribute\_\_ ((packed)) rsa\_public\_key\_2048 {

uint8\_t modulus[256];

uint32\_t exponent;

} rsa\_public\_key\_2048\_t;

### 4.2实例演示

有关debug authentication的原理跟示例都是使用2K长度的秘钥。

#### 4.2.1 使能Debug authentication

1. 配置RT600进入ISP mode，使用blhost.exe上位机进行如下的配置
2. 配置SOCU相关的fuse，打开debug authentication

blhost -p com10 -- efuse-program-once 0x5f 00000055

blhost -p com10 -- efuse-program-once 0x64 00000000

blhost -p com10 -- efuse-program-once 0x68 ffffffaa

1. 烧写Rotk0-3，Rotk用于认证

blhost -p com10 -- efuse-program-once 0x78 24371f32

blhost -p com10 -- efuse-program-once 0x79 b30bd1b5

blhost -p com10 -- efuse-program-once 0x7a 3b70e2e6

blhost -p com10 -- efuse-program-once 0x7b 1e1319d5

blhost -p com10 -- efuse-program-once 0x7c 89e54def

blhost -p com10 -- efuse-program-once 0x7d f0d588b2

blhost -p com10 -- efuse-program-once 0x7e 99f239d3

blhost -p com10 -- efuse-program-once 0x7f 2f3f75ff

1. 烧写boot image到QSPI，设置boot mode并reset。Boot成功后Jlink就连不上device了

Attention：

1. 此处没有烧写RSA4K\_EN的fuse，默认使用公钥长度为2K，烧写的Rotk都是2K长度。
2. 没有配置revoke，默认四个公钥都enable。对DC的签名以及认证使用Rotk0

#### 4.2.2 使用Debug authentication打开debug

想要打开debug，首先我们需要一个可以跟mailbox通信的tool，就是tool.exe

1. 准备烧写到板子中的Rotk对应的四个根私钥
2. 准备一个2K长度的DCK的私钥
3. 创建一个名为debug\_auth.yml的文件，内容如下

socc: 0x0000

uuid: "00000000000000000000000000000000"

cc\_socu: 0x3FF

cc\_vu: 0x00000000

cred\_beacon: 0

auth\_beacon: 0

rot:

- ./rotk0\_cert0\_2048.pem

- ./rotk1\_cert0\_2048.pem

- ./rotk2\_cert0\_2048.pem

- ./rotk3\_cert0\_2048.pem

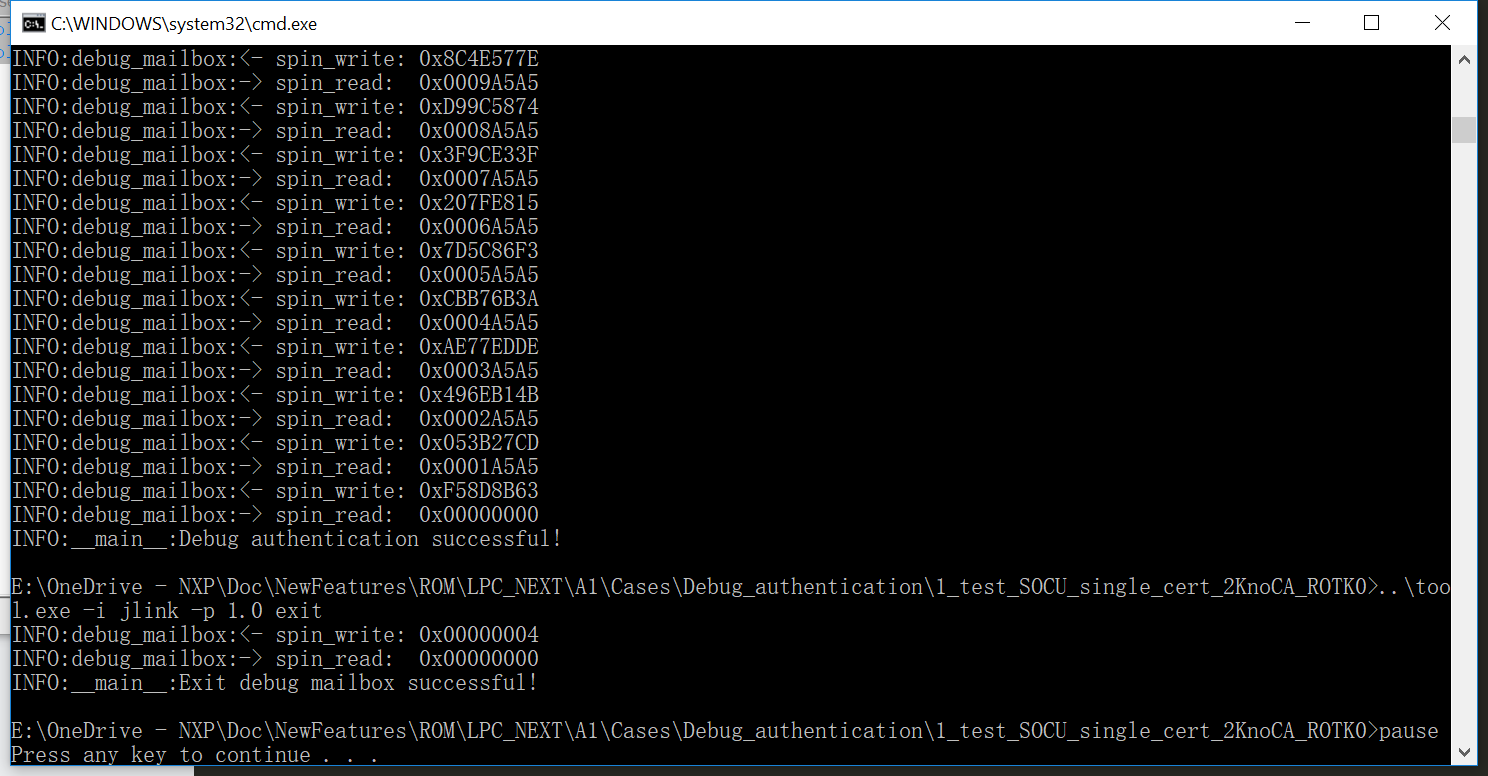
dck: ../dck\_2048.pem

1. Jlink接到板子上，使用如下的命令进行Debug authentication

tool.exe -i jlink -p 1.0 auth -c 0\_debug\_auth.yml

tool.exe -i jlink -p 1.0 exit

1. 成功后会有如下图示的log出现，至此Jlink就可以连上设备了。



## Log buffer

ROM在boot的过程中会记录一些log，这些log存放在全局变量中。

通过查阅log可以获知boot过程中的详细信息，比如从哪个device boot，以什么方式boot等

Log 定义于.noinit段中。

#define LOG\_ADD\_ENTRY bl\_log\_add\_entry

static log\_context\_t s\_logCtx BL\_SECTION(".noinit");

s\_logCtx 0x10008528 0x108 Data Lc bl\_log.o [1] //不同版本的芯片，这个全局变量地址不一样，此处是A0芯片的地址

log组成如下：

MAKE\_LOG\_ENTRY(kLog\_Startup, kLog\_SubState\_Init, kLog\_Status\_Pass, 0),

#define MAKE\_LOG\_ENTRY(state, substate, status, entries) \

(((state) << 24) | ((substate) << 16) | ((status) << 8) | (entries))

每种状态的定义值如下：

typedef enum \_log\_boot\_state

{

kLog\_Startup = 0,

kLog\_HardwareInit,

kLog\_ROMPatch,

kLog\_SecureBoot,

kLog\_BootMode,

kLog\_MasterBoot,

kLog\_PassiveBoot,

kLog\_RecoveryBoot,

kLog\_IspBoot,

kLog\_BootDevice,

kLog\_ImgLoad,

kLog\_Auth, //0xB

kLog\_Jump,

kLog\_Fatal = 0xFF,

} log\_boot\_flow\_t;

----------------------------------------------------------------------------------------------

//!@brief Log Sub state definitions

//! 0x00-0x1F common sub states

//! 0x20-0x2F Image Load sub states

//! 0x30-0x6F Image authentication sub states

//! 0x7F-0xDF Reserved

//! 0xE0-0xFF Device specific sub states

typedef enum \_log\_substate

{

//!@ Commmon Sub States, 0x00-0x1F

kLog\_SubState\_Common = 0x00,

kLog\_SubState\_Init = 0x01,

kLog\_SubState\_Call = 0x02,

kLog\_SubState\_DeInit = 0x03,

kLog\_SubState\_Source = 0x04,

kLog\_SubState\_Check = 0x05,

} log\_substate\_t;

//!@brief Image load Substate defintions

typedef enum \_log\_ImgLoad\_substate

{

//!@ Image Load substate:0x20-0x2F

kLog\_SubState\_InitialLoad = 0x20,

kLog\_SubState\_RemainingLoad = 0x21,

} log\_ImgLoad\_substate\_t;

//!@brief Image authentication Substate defintions

typedef enum \_log\_auth\_substate

{

//!@ Image authentication Substats: 0x30-0x6F

kLog\_AuthState\_CheckSecureState = 0x30,

kLog\_AuthState\_ImageTypeCheck,

kLog\_AuthState\_ReadKeyStore,

kLog\_AuthState\_ReadMacKey,

kLog\_AuthState\_VerifyHMAC,

kLog\_AuthState\_ImageEntryCheck,

kLog\_AuthState\_Authenticate,

kLog\_AuthState\_CrcCheck,

} log\_auth\_substate\_t;

----------------------------------------------------------------------------------------------

typedef enum \_log\_status

{

//@!Status defintions, the value 0x00-0x2F is flow specific

kLog\_Status\_Pass = 0xf0,

kLog\_Status\_Fail = 0xf3,

kLog\_Status\_Fatal = 0xf5,

kLog\_Status\_Invalid = 0xfc,

kLog\_Status\_Default = 0xff,

kLog\_Status\_Disabled = 0xdc,

kLog\_Status\_Enabled = 0xea,

kLog\_Status\_MasterBoot = 0x01,

kLog\_Status\_PassiveBoot = 0x02,

kLog\_Status\_RecoveryBoot = 0x03,

kLog\_Status\_IspBoot = 0x04,

} log\_status\_t;