**ARM 代碼燒錄方案與原理詳解 --- SWD/JTAG + Bootloader + OTA (ICP + ISP + IAP)**

流雲IoT

<https://blog.csdn.net/m0_37621078/article/details/106798909>

於 2020-06-22 21:03:48 發布

**文章目錄**

前言

一、ICP 與 JTAG / SWD

二、ISP 與 Boot Mode

三、IAP 與 Vector Table

四、Bootloader 與 OTA

更多文章：

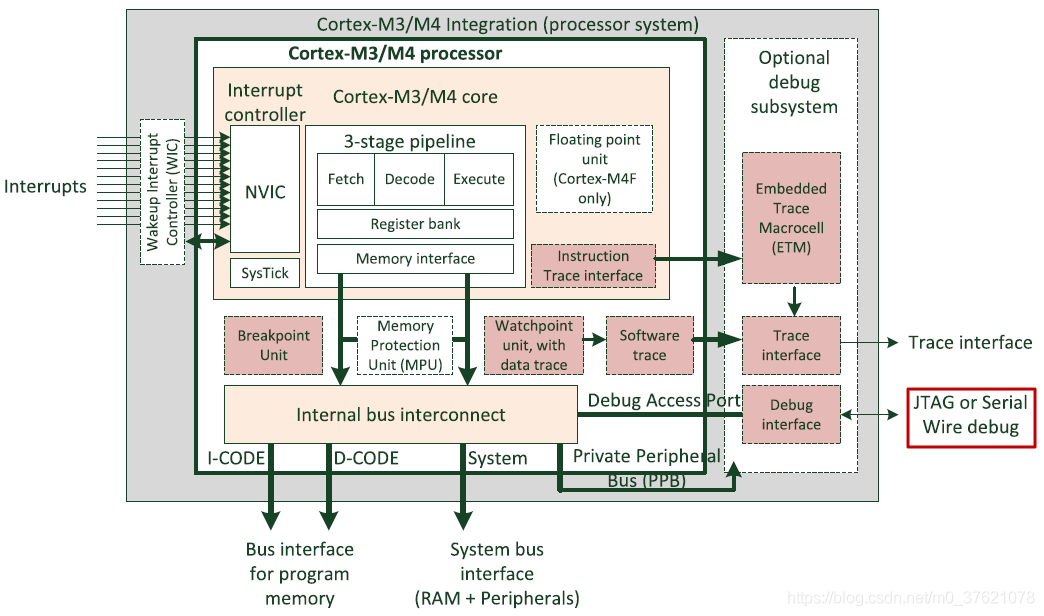
## 前言

我們常見的MCU / CPU 代碼燒錄方式主要有以下三種：

* **ICP(In Circuit Programing)**：在電路編程，可通過CPU的Debug Access Port 燒錄代碼，比如ARM Cortex的Debug Interface主要是SWD(Serial Wire Debug)或JTAG(Joint Test Action Group)；
* **ISP(In System Programing)**：在系統編程，可借助MCU廠商預置的Bootloader 實現通過板載UART或USB接口燒錄代碼，比如STM32存儲映射Code分區中的System memory可以預置廠商的Bootloader，讓MCU支持通過UART下載(不限於UART，具體由廠商預置Bootloader實現而定)；
* **IAP(In Applicating Programing)**：在應用編程，由開發者實現Bootloader功能，比如STM32存儲映射Code分區中的Flash本是存儲用戶應用程序的區間(上電從此處執行用戶代碼)，開發者可以將自己實現的Bootloader存放到Flash區間，MCU上電啟動先執行用戶的Bootloader代碼，該代碼可為用戶應用程序的下載、校驗、增量/補丁更新、升級、恢復等提供支持，如果用戶代碼提供了網絡訪問功能，IAP 還能通過無線網絡下載更新代碼，實現OTA空中升級功能。

## 一、ICP 與 JTAG / SWD

ICP 主要通過CPU的DAP(Debug Access Port) 燒錄代碼，下面以ARM Cortex-M3/M4 為例，展示Debug Interface 如下：

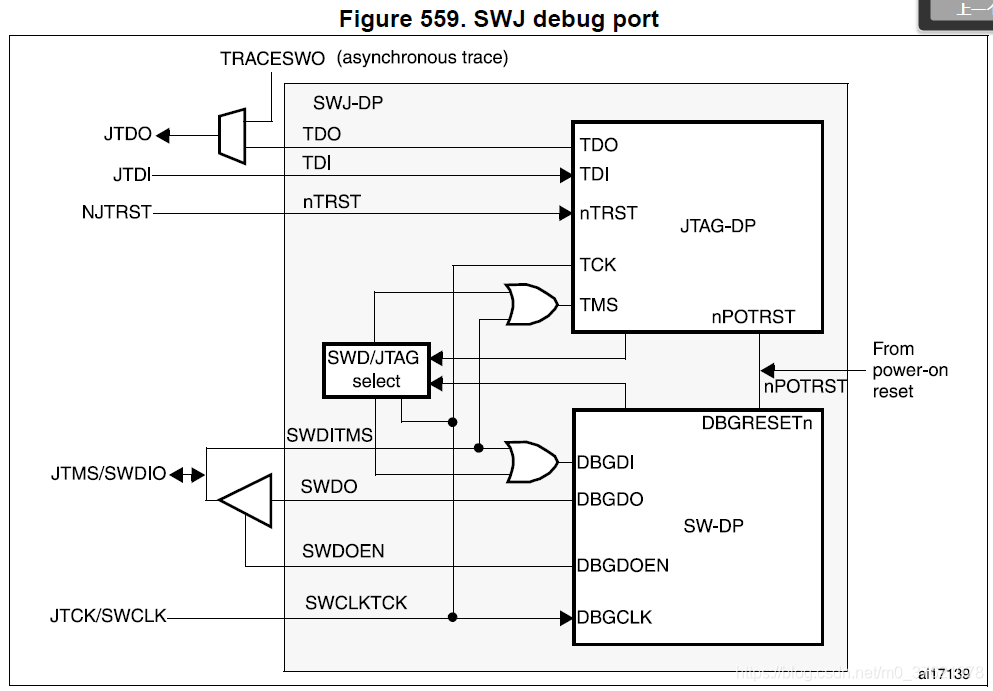


ARM Cortex 內部包含了CoreSight 調試架構，CoreSight 包括調試接口協議、調試總線協議、對調試組件的控制、安全特性、跟蹤接口等。以前的ARM 處理器都提供JTAG 接口，通過它來控制對寄存器和存儲器的訪問，CoreSight 則通過DAP(Debug Access Port) 來控制處理器上的總線邏輯。從上圖也可以看出，CoreSight 除了提供Debug Interface，還提供了Trace Interface，Trace Interface 主要用來觀察數據中間值的變化、跟蹤指令的執行狀態等，只能單向讀取跟蹤數據，屬於非侵入式調試，不能用於燒錄代碼，因此本文不對此做過多介紹。

ARM Cortex 目前支持兩種Debug Interface：

* **SW-DP (Serial Wire - Debug Port)**：新提供的串行線調試接口SWD，只需要兩條信號線(SWCLK、SWDIO)；
* **JTAG-DP (Joint Test Action Group - Debug Port)**：廣為使用的聯合調試接口JTAG，至少需要四根信號線(JTCK、JTMS、JTDI、JTDO，JTRST 為可選信號線)；
* **SWJ-DP (SW + JTAG - DP)**：支持SW-DP 與 JTAG-DP 兩種接口協議。

STM32 同時支持SW-DP 和 JTAG-DP，也即支持的是SWJ-DP 調試接口協議，STM32-L4 的SWJ Debug Port 如下圖所示：



從上圖中可以看出，JTAG-DP 與SW-DP 兩個接口都需要提供Clock 驅動信號，二者的數據傳輸引腳不同，SW-DP 接口使用半雙工的SWDIO 引腳傳輸數據，JTAG-DP 接口使用兩個單工的JTDI 和JTDO 引腳組成全雙工通信。JTAP-DP 除了時鐘引腳和數據傳輸引腳外，還有個重要的引腳JTMS 測試模式選擇引腳，可以控制TAP(Test Access Port) 狀態機的切換，JTRST 主要用來對TAP 狀態機進行復位，為可選引腳(因此有四線JTAG 與五線JTAG 之分)，JTMS 也可以實現TAP 狀態機的復位。JTAG-DP 與SW-DP 引腳的功能描述如下：

一張含有 桌 的圖片

自動產生的描述

早期的MCU 是可插拔的，向其內燒錄代碼時常需要將其拔下來，借助燒錄器或仿真調試器通過Debug Interface 將代碼燒錄進MCU 後，再把芯片插回到電路板上。後來MCU 直接銲接在電路板上，電路板上直接引出了Debug Interface，比如常見的JTAG 和 SWD 接口，可以借助JLINK 或 STLINK 等仿真器，通過JTAG 或 SWD 接口直接將代碼燒錄到MCU 中，省去了MCU 插拔操作，方便了代碼燒錄和調試操作。

不同的IC 廠家會自己定義自家產品專屬的JTAG 接口，來下載和調試程序。嵌入式系統中常用的有20pin、14pin、10pin 等JTAG 接口類型，不同類型的JTAG 接口都包含前面提到的五個引腳，另外包括VRef 和 GND 等引腳，它們的不同之處主要在於GND引腳的數量、一些用戶自定義引腳、保留引腳等。一般JTAG 接口同時支持 SWD 接口，但有些主板受空間限制，只提供了SWD 接口，只需要四個引腳，大大節約了空間。

一張含有 桌 的圖片

自動產生的描述

IC 廠家提供的部分開發板上直接集成了 JLINK 或STLINK 模塊，相當於將JLINK / STLINK 仿真器內置到開發板上了(比如STM32L4 Pandora 或 NRF52 DK)，不再需要額外的仿真器(也不再需要板載JTAG 或 SWD 接口)，直接通過USB 線連接電腦，就可以方便的進行代碼燒錄和調試，對於手頭沒有仿真器的開發者更加友好，畢竟正版仿真調試器價格還是有點貴的。

## 二、ISP 與 Boot Mode

ISP 可以通過USB接口(可以是UART協議或USB協議等)直接燒錄代碼，ARM Cortex 提供的Debug Interface 並不支持UART 或USB 協議，這就需要IC 廠商預置UART 協議或USB 協議代碼，讓MCU / CPU 支持UART 通訊或USB 通訊。UART 協議比較簡單、傳輸速率比較低，比如STM32 這類MCU 支持UART 協議燒錄代碼；USB 協議相對複雜、傳輸速率比較高，比如高通驍龍這類CPU 支持USB 協議燒錄代碼。

以STM32 為例，先看看有哪些啟動模式：

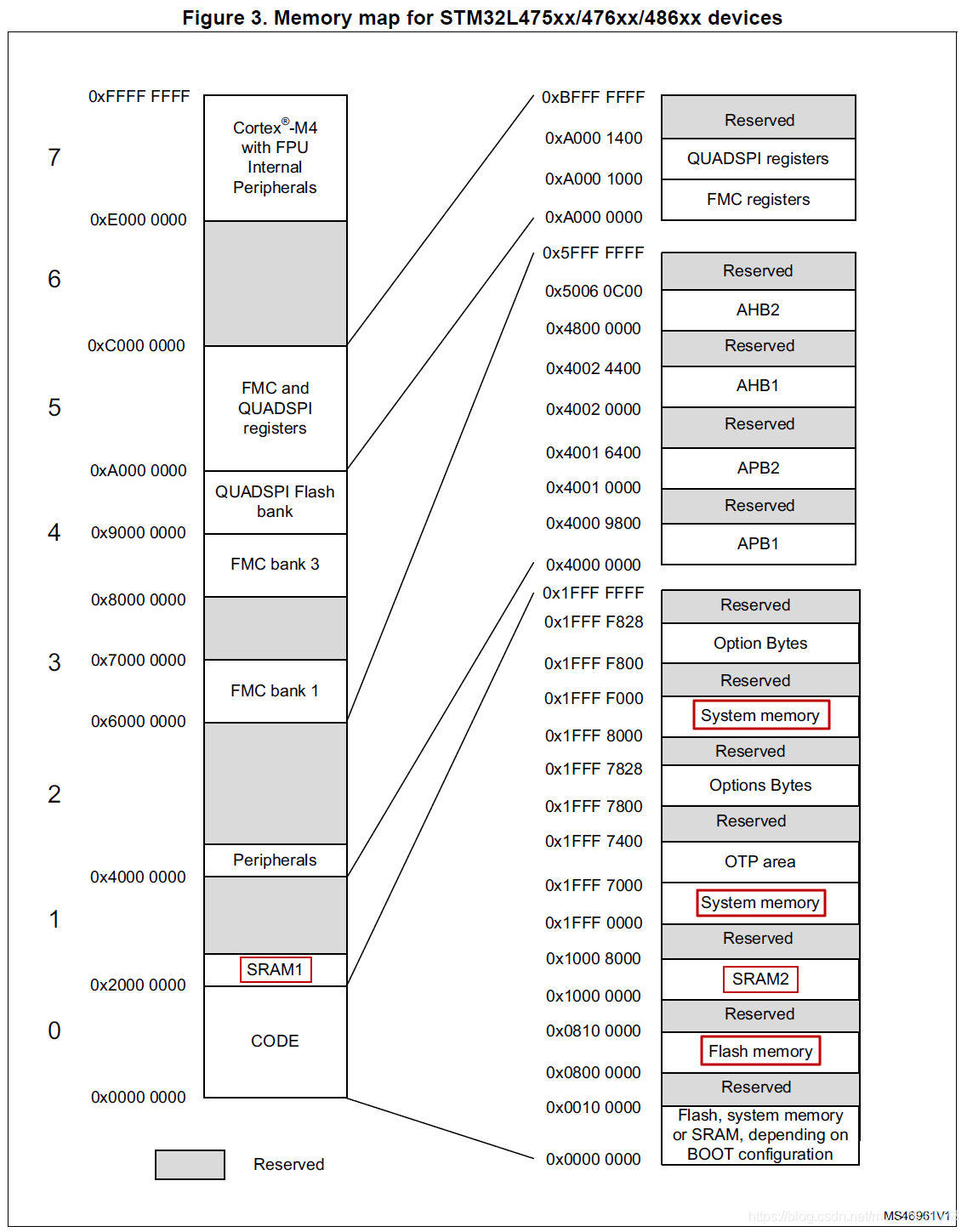
一張含有 桌 的圖片

自動產生的描述

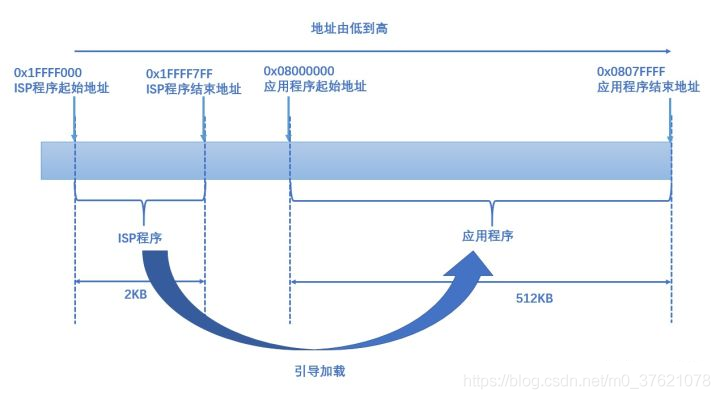
STM32 有三種Boot modes 開始執行代碼：

* **Main Flash memory**：STM32內置的Flash，一般我們使用JTAG / SWD 燒錄代碼都是直接燒錄到這個存儲區間，重啟後也直接從這裡的代碼開始執行；
* **System memory**：這種模式啟動的程序功能是由廠家設置的，STM32 在出廠時由ST 在這個存儲區間內部預置了一段BootLoader(也即ISP 程序)，這段程序出廠後無法修改。廠家提供的BootLoader 一般支持UART 協議，可以讓我們直接通過串口將程序代碼燒錄到Main Flash memory 中；
* **SRAM1**：這個啟動模式一般用於程序調試，假如我只修改了代碼中一個很小的地方，重新燒錄到Flash比較費時，可以從STM32 內存中啟動代碼，進行快速的程序調試，等程序調試完成後，再將程序燒錄到Main Flash memory 中。

Main Flash memory、System memory、SRAM1 的存儲區間分佈如下圖所示(在博文：存儲管理與虛擬內存中也有介紹)：



從STM32 的啟動模式可以看出，ISP 通過USB 接口直接燒錄代碼，主要是靠System memory 中被廠家預置的Bootloader 代碼提供的UART 協議實現的，要想使用System memory 中的ISP 代碼，需要將BOOT0 設置為高電平、BOOT1 設置為低電平，然後按下復位鍵，MCU 從System memory 啟動BootLoader，靠ISP 代碼中提供的UART 協議，用戶可以通過串口將程序代碼燒錄到Main Flash memory 中。通過串口燒錄代碼完成後，需要將BOOT0 設置為低電平，然後按下復位鍵，MCU 就可以從剛才燒錄進去的代碼處(也即Main Flash memory)開始執行了。



借助ISP 代碼通過USB 接口燒錄程序的方法需要切換Boot modes，也即需要切換BOOT0 與BOOT1 引腳的電平，常使用跳線帽來切換這兩個BOOT 引腳的電平。但手動插拔跳線帽略嫌麻煩，能否不手動操作，直接通過軟件方式控制兩個BOOT 引腳的電平呢？

正點原子設計的STM32 開發板借用串口的兩個流控引腳DTR、RTS 信號，分別控制STM32 的復位(需要復位才能切換啟動模式)和BOOT0 (從Main Flash memory 啟動與從System memory 啟動主要看BOOT0 電平高低，跟BOOT1 電平無關)，配合上位機軟件FlyMcu，就可以通過軟件控制流控引腳DTR 和RTS 的信號電平，實現通過軟件控制RESET 和 BOOT0 電平高低的效果。這樣就不再需要用戶來回拔插跳線帽，直接通過軟件比如FlyMcu 自動控制RESET 與BOOT0 的電平高低，從而實現一鍵燒錄功能，對開發者燒錄代碼更友好。

一張含有 文字 的圖片

自動產生的描述

上圖FlyMcu 設置“DTR的低電平復位，RTS高電平進BootLoader” 和“編程後執行”，在右側Log 數據中可以看到MCU 復位、選擇進入Bootloader、燒錄程序代碼的過程，待燒錄完成後還會看到MCU 復位、選擇進入Main Flash memory 開始執行代碼的過程。

ISP 使用哪種接口燒錄程序代碼取決於廠商預置的ISP 代碼支持哪種通訊協議，UART 算是最簡單且的比較常用的通訊協議，如果廠商的ISP 代碼支持，ISP 還可以使用SPI、CAN、IIC、USB 等通訊協議燒錄代碼到Main Flash memory 存儲區間。廠商的ISP 代碼是怎麼預置到System memory 中的呢？答案正是靠前面介紹的處理器Debug Interface(比如ARM 的JTAG / SWD 接口) 實現ISP 代碼燒錄的。

如果ISP 指的是MCU / CPU 不用脫離電路板，可以直接通過電路板上預留的接口燒錄或擦除程序代碼的話，前面介紹ICP 時提到的直接在電路板上預留JTAG / SWD 接口、甚至直接將JLINK / STLINK 仿真模塊預置到開發板上只留出USB接口，都可以通過電路板上預留的接口燒錄或擦除程序代碼，雖然仍是通過Debug Interface 燒錄代碼的，卻可以稱之為ISP。

## 三、IAP 與 Vector Table

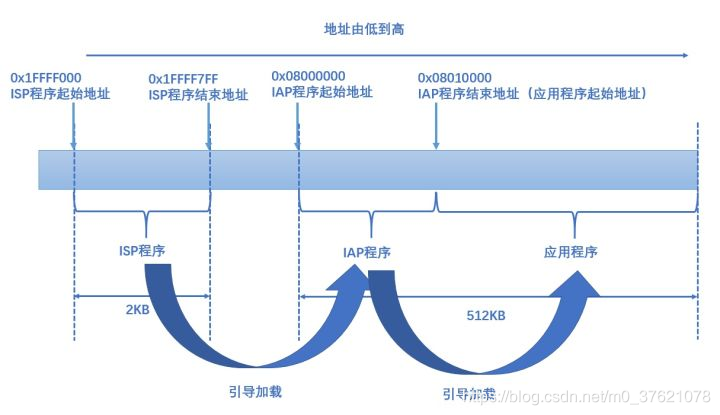
IAP 可以在系統運行過程中完成內部程序代碼的更新，由Main Flash memory 存儲區間內運行的程序自身實現更新代碼的下載、校驗、燒錄等工作。IAP 並不是通過Debug Interface 完成更新代碼燒錄的，自然也需要類似ISP 代碼的Bootloader 完成這些工作，看起來IAP 就相當於將ISP 中廠商預置到System memory 中的Bootloader 搬移到Main Flash memory 中了，MCU / CPU 不再需要切換BOOT Mode，直接從Main Flash memory 啟動即可。

將Bootloader 搬移到Main Flash memory 中，最明顯的好處就是Bootloader 的功能可以由開發者自由實現(ISP 中廠家預置到System memory 的Bootloader 出廠後不可修改)，如果存儲資源足夠，可以增加很多功能，比如增量更新、空中升級、數據線燒錄、SD卡燒錄、系統恢復等。

由於Bootloader 主要是在代碼燒錄和更新過程中用到，並不涉及處理業務邏輯，需要將Bootloader 和用戶的Application 分割開來，將二者分割最簡單的方式就是將Bootloader 與Application 存放在不同的存儲區間，而且Bootloader 應該放置到Main Flash memory 開始位置，便於上電從Bootloader 代碼開始執行，Application 放置到Bootloader 後面。舉例說明，比如Bootloader 佔用存儲空間64KB，存放在Main Flash memory 初始地址0x0800 0000 開始的一段存儲區間內，將 Application 存放在0x0800 0000 向後偏移Bootloader 佔用的64KB 距離處的0x0801 0000 開始的一段存儲區間內，同時要在Bootloader 中設置要跳轉到Application 代碼的起始地址0x0801 0000，這裡的地址只是舉了個例子，Application 存放的起始地址可由開發者設定，並能被Bootloader 識別並跳轉過去即可。

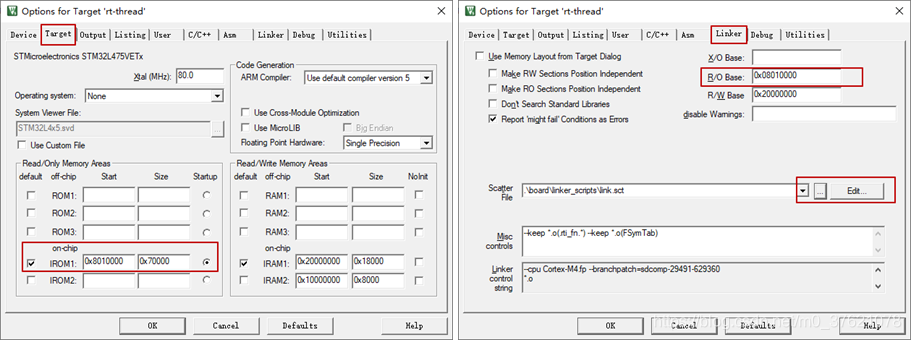
IAP 中由開發者實現的Bootloader (也稱IAP 程序)是怎樣燒錄到Main Flash memory 存儲區間的呢？答案自然是前面介紹的ICP 或 ISP 方式，也即通過JTAG /SWD 接口或者ISP 程序將用戶的Bootloader 燒錄到Main Flash memory 內。由於MCU / CPU 上電後默認從Main Flash memory 開始執行，也即從開發者實現的Bootloader 代碼處開始執行，處理業務邏輯的Application 代碼如何燒錄到對應存儲區間呢？

Application 代碼既可以使用前面介紹的JTAG /SWD 接口或者ISP 程序燒錄到Main Flash memory 內指定地址區間(記得修改默認的燒錄地址，為用戶Bootloader 預留出空間，如果從默認的Main Flash memory 起始地址開始燒錄，將會覆蓋用戶的Bootloader 代碼，運行時由於Application 代碼修改了中斷向量表偏移地址，不燒錄到設定地址會導致指令執行錯誤)，也可以使用IAP 代碼燒錄到Main Flash memory 內指定地址區間。燒錄Application 代碼使用哪種接口，就要在Bootloader 中實現相應的通訊協議，如果ISP 代碼中不支持想要的通訊協議，可以在IAP 代碼中添加相應的通訊協議，然後借助IAP 代碼將Application 代碼燒錄到Main Flash memory 內指定地址區間。



IAP 燒錄的應用程序代碼在IAP 代碼後面，也即默認的Main Flash memory 起始地址向後偏移一段距離之後(假如偏移量64KB，Application 代碼應存儲在0x0801 0000 起始的地址區間內)。如果通過IAP 程序來燒錄Application 代碼，可以在IAP 程序中控制Application 代碼應被存儲的區間地址，Application 代碼文件可以使用bin 格式的純數據文件以減小文件大小；如果使用ISP代碼或JTAG/SWD 接口燒錄Application 代碼，燒錄文件一般使用包含地址信息的Hex格式，以便將代碼數據正確燒錄到指定地址區間。

以Keil MDK 開發環境為例，編譯生成hex 文件或者借助JLINK / STLINK 仿真器直接燒錄Application 代碼到指定的地址區間(也即0x0800 0000 + Offset 起始的地址區間)，相比直接燒錄到Main Flash memory 起始的地址空間應該修改哪些配置呢？答案是修改ROM 地址區間(一般Bootloader 與Application 並不同時運行，可以不用修改RAM 地址區間)，Keil MDK 主要在Target 和Linker 兩個地方修改ROM 起始地址和區間大小參數。以STM32 L475為例，Main Flash memory 的起始地址Start 為0x0800 0000，區間大小Size 為0x80000(也即512KB)，現在IAP 代碼存儲在0x0800 0000 處，佔用地址區間長度為0x10000(也即64KB)，Application 代碼應存儲在IAP 代碼之後，也即起始地址Start 為0x0801 0000，剩餘地址空間大小為0x70000，地址區間配置界面如下圖示：

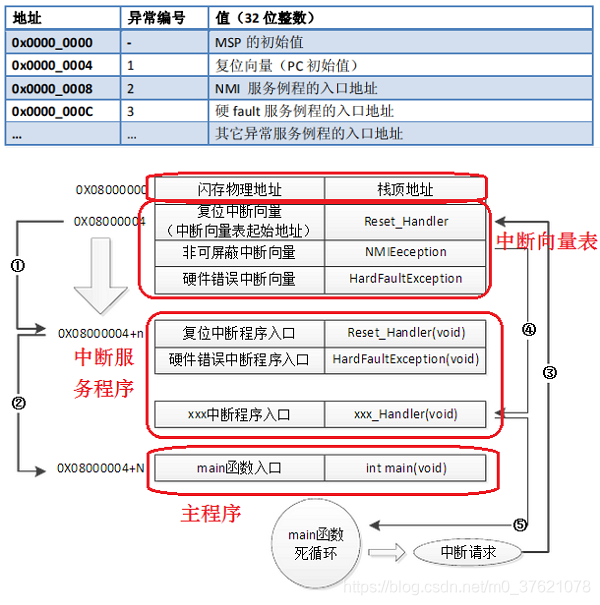


點擊上圖Linker 界面的“Edit” 按鈕，即可編輯鏈接腳本文件“link.sct”，按照Target 地址區間配置修改鏈接腳本中的ROM 地址區間如下：

|  |
| --- |
| // .\link.sct  ; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  ; \*\*\* Scatter-Loading Description File generated by uVision \*\*\*  ; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  LR\_IROM1 0x08010000 0x00070000 { ; load region size\_region  ER\_IROM1 0x08010000 0x00070000 { ; load address = execution address  \*.o (RESET, +First)  \*(InRoot$$Sections)  .ANY (+RO)  }  RW\_IRAM1 0x20000000 0x00018000 { ; RW data  .ANY (+RW +ZI)  }  } |

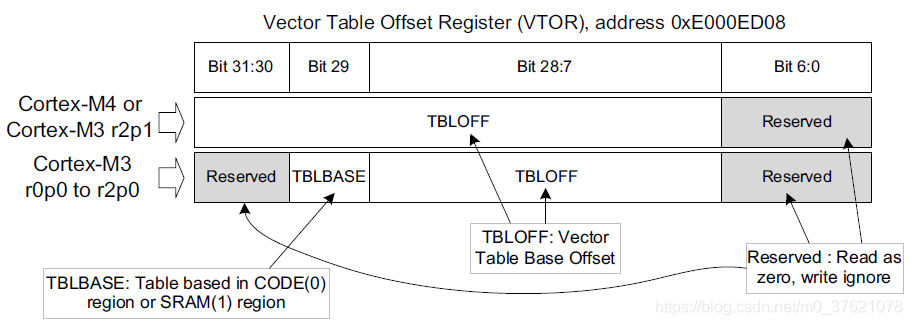
將Application 代碼燒錄到指定的位置(也即IAP 代碼之後)還不足以保證代碼的正常運行，首先遇到的問題是MCU/CPU 怎麼找到並跳轉至Application 代碼處開始執行？MCU / CPU 運行過程中怎麼找到並跳轉至中斷、異常等服務函數的入口地址？

ARM Cortex 使用中斷向量表來管理main函數、中斷/異常服務函數的入口地址，每個獨立的程序代碼都有自己的中斷向量表，某程序所屬的中斷向量表位於該程序代碼的起始地址處(可參考博文：系統啟動和中斷管理)。ARM Cortex 中斷向量表的前幾項及中斷服務程序跳轉過程如下圖示：



中斷向量表的前兩項分別保存了MSP與PC的初始值，處理器剛上電復位後，硬件會自動根據向量表偏移地址(查詢VTOR向量表偏移量寄存器)找到中斷向量表，硬件會自動從向量表首地址(初始地址0x0000 0000，設置BOOT Modes從Main Flash memory啟動時則映射到0x0800 0000，設置BOOT Modes從System memory啟動時則映射到0x1FFF 0000)處讀取數據並賦給棧指針SP，然後自動從向量表第二項地址處(向量表每一項佔用4 字節，也即向量表首地址 + 4 字節處)讀取數據賦給程序計數器PC，開始執行PC地址上存儲的指令，PC初始值指向復位向量，所以剛上電後首先執行的是復位指令。

中斷向量表中的異常 / 中斷向量也都是基於中斷向量表首地址偏移一定字節得到的，我們只需要讓MCU / CPU 能正確找到中斷向量表的首地址就可以讓Application 代碼正確執行，中斷向量表首地址是通過查詢VTOR 寄存器獲得的，自然也可以通過設置VTOR(Vector Table Offset Register)讓MCU / CPU 能找到Application 的中斷向量表，VTOR 寄存器結構如下：



ARM Cortex M3與M4的VTOR 寄存器結構還有些區別，以M4 為例，要求VTOR 的bit 6:0為0(由於向量表起始地址要求對齊到2 的整次冪，比如MCU 共使用了31 個系統異常與中斷向量，也即佔用31\*4 = 124 字節，對齊到2 的整次冪就是27 = 128 = 0x80 字節，VTOR 低7 位就是0 了)，只有bit 31:7 為向量表偏移值TBLOFF，我們可是使用掩碼將bit 6:0 設置為0，設置SCB->VTOR 寄存器的代碼如下：

|  |
| --- |
| /\* 將中斷向量表起始地址重新設置為 app 分區的起始地址 \*/  #define NVIC\_VTOR\_MASK 0xFFFFFF80  #define APP\_START\_ADDR 0x08010000  /\* 根據應用設置向量表 \*/  SCB->VTOR = APP\_START\_ADDR & NVIC\_VTOR\_MASK; |

對於STM32L475 芯片，一共有98 個異常與中斷向量，中斷向量表佔用空間 98\*4 = 392 字節，對齊到2 的整次冪為 29 = 512 = 0x200 字節，因此對於STM32L475 的Application 代碼及其中斷向量表起始地址應設置為0x200 的整數倍，也可以將STM32L475 的NVIC\_VTOR\_MASK 設置為0xFFFF FE00，每個芯片支持的中斷向量數不同，因此需要根據情況設置。

為什麼我們不使用IAP 代碼時不需要設置VTOR 寄存器呢？還記得前面介紹的BOOT Mode 吧，我們設置BOOT Modes從Main Flash memory啟動時，MCU 硬件會自動從Main Flash memory 起始地址處讀取中斷向量表的值；設置BOOT Modes從System memory啟動時，MCU 硬件會自動從System memory 起始地址處讀取中斷向量表的值。當我們使用IAP 代碼管理Application 代碼的燒錄、升級時，MCU 硬件不會到Application 代碼起始地址處讀取中斷向量表，默認依然從Main Flash memory 起始地址處(也即IAP 代碼起始地址)讀取中斷向量表。這就需要我們在Application 代碼中通過設置VTOR 寄存器的值，告訴MCU 應該到哪去讀取Application 程序的中斷向量表，將Application 代碼的起始地址設置到VTOR 寄存器，MCU 就可以通過讀取VTOR 的值找到Application 程序的中斷向量表，Application 代碼就可以正確執行了。

## 四、Bootloader 與 OTA

ISP 與IAP 燒錄代碼都用到了Bootloader 程序，不同的是ISP 的Bootloader 是廠家預置的，功能比較簡單且固定，IAP 的Bootloader 是開發者自己實現的，功能比較靈活且豐富。那麼，Bootloader 都能實現哪些功能呢？

Bootloader 實際上就是一段代碼，跟我們編寫裸機程序差不多，ISP 代碼比較簡單，以UART Bootloader 為例，主要實現UART 驅動，可以通過UART 通訊協議傳輸數據，當然也要能夠解析出HEX 文件中的地址信息，將UART 接收到的數據寫入到特定的地址區間。

IC 廠商預置的ISP 代碼一般只實現了通過常用的UART 或USB 等通訊協議燒錄Application 代碼的功能，有些場景我們需要使用其它的通訊接口比如CAN 通訊實現代碼燒錄與更新，這就需要我們自己實現IAP 代碼，由IAP 代碼實現通過CAN 通訊協議燒錄Application 代碼的功能。如果你對手機進行過刷機操作，應該瞭解過fastboot 刷機功能，fastboot 的功能實際就是由IAP 代碼實現的，如果要使用fastboot 功能需要先通過ISP 代碼提供的USB 通訊協議向CPU 中刷入Bootloader 代碼(一般Bootloader 代碼與Application 代碼一起通過USB 通訊協議燒錄進CPU 中)。

IAP 代碼可以實現ISP 代碼的功能，但開發者既然選擇自己實現Bootloader，IAP 代碼就不只是具有燒錄Application 代碼的功能，更多的是為了升級代碼，甚至是空中升級代碼。

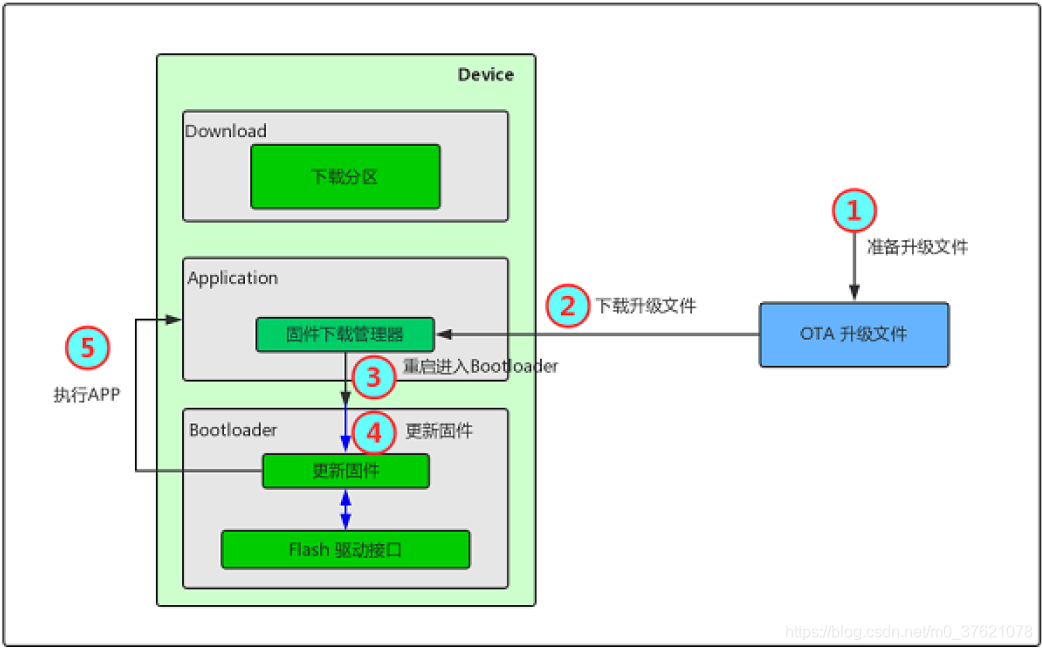
我們應該已經習慣升級系統版本或者給系統打補丁解決Bug等，不管是版本升級還是打補丁，都涉及到在系統運行過程中，重新燒錄部分代碼數據。如果沒有IAP 代碼，我們只能借助ISP 代碼使用第三方設備和燒錄工具，完全覆蓋先前的代碼，這對普通用戶是極不友好的。我們借助IAP 代碼就可以不依賴第三方設備與燒錄工具，僅靠自身就可以完成版本升級或打補丁的功能，不管是PC 還是手機操作系統也都是依靠系統開發商提供的Bootloader 實現系統版本或補丁升級的。

IAP 代碼怎麼實現系統版本升級的呢？回想我們在PC 或手機上升級應用軟件，都是先將軟件安裝包下載到本地存儲介質中，然後再運行軟件安裝包將軟件執行代碼複製到系統安裝目錄下，也即需要佔用存儲介質中一定的存儲空間來暫存軟件安裝包。升級Application 程序版本的過程與此類似，也是將Application 代碼文件暫存到某個存儲介質中(可以是RAM、ROM、SD卡、U盤等存儲介質)，Bootloader 會先校驗Application 代碼文件的有效性，通過校驗後再將Application 代碼從存儲介質搬移到特定的存儲區間(比如Main Flash memory 中IAP 代碼後的存儲區間)。

如果我們開發的產品不具備網絡連接能力，也只能先用電腦下載好固件升級包，暫存到SD卡或者U盤中，再插到我們的產品上。在IAP 代碼中可以實現SD卡通訊協議或USB 通訊協議，支持從SD卡或U盤讀取固件升級包，校驗通過後搬移到Main Flash memory 的特定存儲區間，實現Application 版本升級的目的。

隨著物聯網設備的普及，開發的產品越來越多支持網絡訪問能力，我們可以將Application 版本升級過程設計的更簡單。直接借助產品的網絡訪問能力，從特定服務器將系統升級包下載到本地存儲介質特定分區內(可以是RAM、片內Flash、片外Flash、SD卡等本地存儲介質)，校驗通過後再將代碼搬移到Main Flash memory 的特定存儲區間。如果系統升級包是通過無線網絡(比如移動蜂窩網、WLAN、WPAN等無線協議)下載的，這種通過無線網絡下載升級包，並通過Bootloader 實現版本升級的技術稱為OTA(Over-the-air programming)空中升級或空中編程技術。

要具備OTA 空中升級能力，只有負責固件升級包校驗升級的Bootloader 分區和負責業務邏輯的Application 分區是不夠的，還需要有暫存固件升級包的Downloader 分區。每次MCU / CPU 上電啟動都從Bootloader 開始執行代碼，Bootloader 會到Downloader 分區檢查是否有可升級的固件包，如果沒有則直接跳轉到Application 代碼執行，如果有則對固件升級包校驗通過後，將新的固件代碼搬移到Application 分區，完成Application 代碼更新後，再跳轉到新的Application 代碼執行。

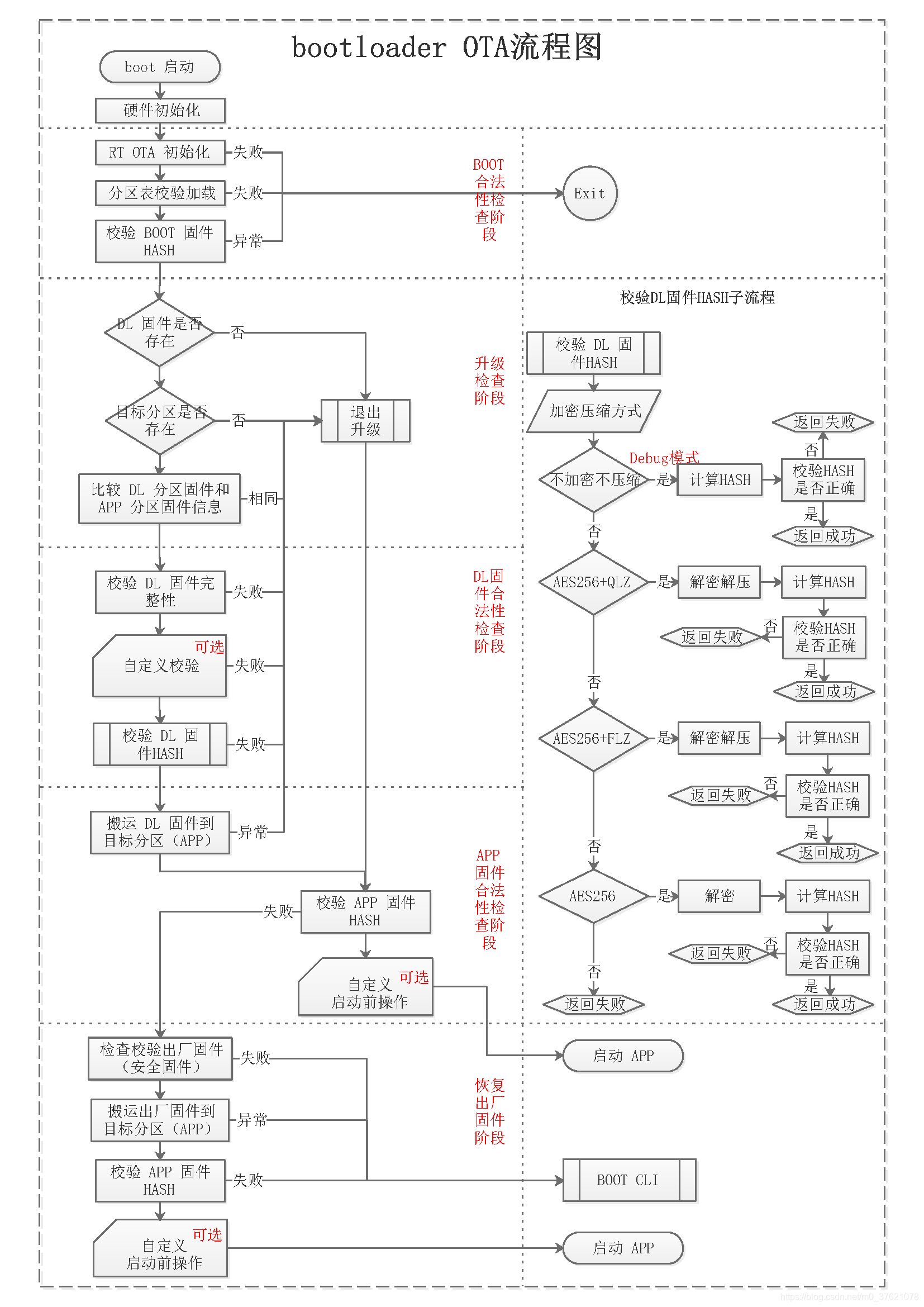


為了增強系統更新包網絡傳輸的安全性，防止被網絡攻擊者篡改，可以對通過網絡下載的固件升級包先進行加密認證(包括數據加密、數據完整性校驗、客戶端與服務器身份校驗等)，Bootloader 就要能對固件升級包進行解密和認證校驗。為了減少網絡傳輸開銷(也即減少消耗的網絡流量)，可以對固件升級包進行壓縮後傳輸，Bootloader 就要能夠對固件升級包進行解壓縮。

有時候我們只是想升級Application 的部分代碼，如果將全部代碼下載後升級比較耗費資源且效率較低，這時候就需要Bootloader 提供只更新部分代碼的能力。將更新代碼做成差分升級包(相當於補丁文件)，通過網絡將差分升級包下載到Downloader 分區，Bootloader 對其解壓縮、解密、校驗通過後，將要更新的那部分代碼搬移到Application 特定的存儲區間內(差分升級包包含了要更新代碼的地址信息，Bootloader 可以解析出地址信息，並將更新代碼搬移到目標存儲區間)。差分升級包一般比整個固件升級包佔用的存儲空間小得多，可以極大節省存儲空間和網絡流量，也為整個升級過程節約了不少時間，效率更高。

使用OTA 空中編程技術對產品進行固件升級對用戶是比較友好的，只要能訪問網絡，就可以實現一鍵升級功能。但Application 代碼運行過程中有可能會出現一些預料之外的Bug，比如某些重要文件被刪除或篡改、被網絡攻擊、中病毒、垃圾過多耗盡資源等，這就需要具備一鍵恢復能力。PC 和手機上都提供了recovery 恢復功能，該功能是怎麼實現的呢？原理跟升級類似，可以再劃分出一個recovery 分區，將出廠的軟件版本壓縮後存放到該分區內，如果系統運行出現了錯誤，可以將recovery 分區內的代碼搬移到Application 分區，相當於實現了恢復出廠設置的功能。recovery 分區內也不一定一直存儲出廠版本，可以同步更新為前一個比較穩定的系統版本，使用recovery 恢復功能就相當於回退到了前一個穩定版本，即便系統因為中病毒或網絡攻擊等特殊情況，也能恢復過來繼續正常使用。

Bootloader 除了具備固件升級和固件恢復外，還應針對升級過程中突然中斷(比如斷電了)的情況妥善處理，儘量減少返廠的機會。Bootloader 該怎麼應對升級突然被打斷的情況呢？正常情況下RAM 中的數據斷電後就消失了，我們可以借鑑斷點續傳技術的實現方案，Bootloader 在升級代碼過程中，在ROM 中同步記錄升級進度與狀態，可以讓MCU 再次上電從Bootloader 啟動時，從ROM 中讀取升級進度與狀態，繼續完成先前的升級過程。



更多文章：

[《IOT-OS之RT-Thread(十七)— 如何使用HTTP協議實現OTA空中升級》](https://blog.csdn.net/m0_37621078/article/details/105442358)

[《STM32之CubeL4(一)—HAL詳解與CubeMX使用》](https://blog.csdn.net/m0_37621078/article/details/100084448)

[《RTOS之UCOS(六)—系統啟動與固件移植》](https://blog.csdn.net/m0_37621078/article/details/92799772)