**ROM固化试验与方案**

作者: 韦成、PeterLee

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 摘要 | 审阅 | 日期 |
| D0 | 初稿 | 黄正多、薛鹏宇、区伟权等 | 2022.03.09 |
| D1 | 1. 为试验详细报告增加图例 2. 修改实施方案及示意 3. 修正一些描述 |  |  |
| D1 | 新增具体实施报告 |  | 2022.03.30 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目录

[1. 背景与需求 4](#_Toc99542210)

[2. 运行空间示意 4](#_Toc99542211)

[3. Ceva栈拆分原理 5](#_Toc99542212)

[3.1. CEVA栈固化之VT表的试验 5](#_Toc99542213)

[3.2. 给固化原代码打Patch原理 6](#_Toc99542214)

[3.3. 试验详细报告 6](#_Toc99542215)

[3.4. 固化与Patch过程 14](#_Toc99542216)

[4. 实施方案 15](#_Toc99542217)

[4.1. 实施方案示意 15](#_Toc99542218)

[5. 风险评估 17](#_Toc99542219)

[6. 具体实施报告 18](#_Toc99542220)

[6.1. 参数信息 18](#_Toc99542221)

[6.2. 实际实现及使用说明 20](#_Toc99542222)

[6.3. 测试用例 21](#_Toc99542223)

[6.3.1. 自测 21](#_Toc99542224)

[6.3.2. 专业测试 22](#_Toc99542225)

[6.4. 后续工作 22](#_Toc99542226)

# 背景与需求

由于内存等资源紧张等原因提出ROM固化需求，固化中又以CEVA栈为重，CEVA栈主要会拆分为LL、Host、APP、Modules等库。其中CevaLL及C 库将固化到ROM中。固化引入的打补丁是重大难题。

# 运行空间示意

APP…

Ceva Host

Ceva LL

C库

RAM

Drivers

IST…

ROM

FLASH

# Ceva栈拆分原理

vt\_f1  
vt\_f2  
vt\_f3  
vt\_f4  
…

VT

f1  
f2  
f3  
f4  
…

patch\_f1  
patch\_f2  
patch \_f3  
patch \_f4  
…

ROM

RAM / FLASH

1. 主要拆分为两部分，会在SDK中编译成库CevaLL.a、CevaHost.a。当然也可以将Modules独立成库。
2. Modules中大部分归LL， 小部分归Host。
3. LL 与Host的函数相互调用，将转化为一个函数表VT(virtual Table ，虚表)，表指针vtPtr将以参数方法传递给LL初始化函数，LL库将用静态全局变量记住vtPtr。函数表就像C++的虚函数表，表内容可以存放于FLASH中。
4. 额外地，LL初始化函数也可接收其它参数，一些参数还可通过LL进一步设置基带。当然也可通过虚函数获取其它参数。内存分配、串口打印等函数，建议也放在VT中。Ceva的个别函数，如果需要重载，也可放入VT中，但存在级联风险，导出越多，风险越小。

# CEVA栈固化之VT表的试验

VT表原理本是很简单的，但试验起来非常困验，因为CEVA栈并没有这种设计，其LL层代码较大，修改起来工作量巨大，需要找到一种简单的方法，但寻找这些方法占用很长时间：

1. 寻找外部依赖函数，采用JT(jump Table, 跳转表)来代替VT，RISCV一条指令并不支持长跳转，非常麻烦。
2. 寻找编译器、链接器等支持，基本没有妥当的快捷方案。
3. 采用JT+VT方案，仍需要修改LL内部函数定义时的名称，修改量也很大，同时也会增大代码空间。
4. 全局变量方案，每个函数定义时被改名，原名称为全局变量指向该函数；原函数声明变为函数指针声明，这时调用者在代码级透明。这个其实回归到VT本身，它的技巧是利用全局变量可以在运行开始自动初始化函数地址，所有全局变量归段后自动形成VT表。

试验中重要发现是CEVA栈对补丁函数的调用层级，是树的关系，极其复杂，经沟通协商，**一致同意**导出CEVA栈固化的**全部**函数，由于没有寻找到妥善的解决方案，因此将有巨大编码工作。全部导出尽管增大VT,但降低了补丁风险。

# 给固化原代码打Patch原理

基本思路：

1. 把所有需要固化或固化代码直接调用到的非固化代码使用函数指针的形式调用，函数指针的地址需要固定到ram/flash，也就是说每次调用一个函数先从某个地方(可以是ram/flash但不是rom)取到这个函数的地址，再调用它，这样就可以做到轻松为每个rom函数打patch。每个函数指针占4byte，1000个函数只占4k空间，如果直接对rom代码打patch需要巨大空间。
2. 函数指针的名称使用原函数名，函数定义名称加前缀。使用rom函数sch\_prog\_push举例：定义指针sch\_prog\_push，原函数sch\_prog\_push的调用位置不做任何修改，sch\_prog\_push定义的位置名称加前缀为vt\_sch\_prog\_push。默认情况下使sch\_prog\_push = vt\_sch\_prog\_push。如果需要打patch， 只需要重新赋值sch\_prog\_push = path\_sch\_prog\_push ，理论上path\_sch\_prog\_push函数所在位置可以是ram/flash的任何地址。

# 试验详细报告

下面是详细测试：

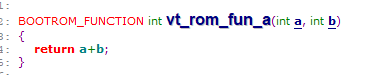
**测试case1**：直接给函数指针赋不同值1. 函数在rom，2. 函数在flash。并且在rom空间调用，两次rom空间的二进制指令是否有任何不同。

1. 定义变量

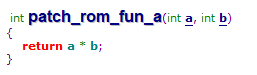




1. 定义rom空间对应函数如下：



1. 定义flash空间patch如下



1. 分别给函数指针分两次赋值，比较两次fw\_rom.bin的md5





1. 分别调用函数



结论：2次fw\_rom.bin的md5完全一样



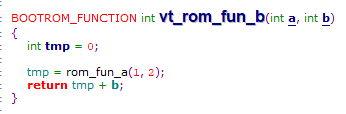
**测试case2:**函数a和b都在rom中，但b调用了a，b需要打patch。且b又被rom里的函数vt\_for\_patch调用，而函数vt\_for\_patch又被flash里的代码调用。这样是否影响fw\_rom.bin的md5。

1. a使用测试case1的，rom\_fun\_a。定义函数指针b

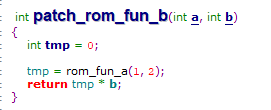




1. 定义函数b



1. 定义函数b的patch

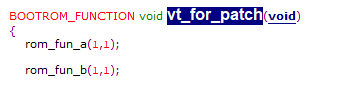


1. 给函数指针分别赋值





1. 在vt\_for\_patch调用a和b同时在main(main在flash中)中调用for\_patch



1. 比较打patch前后fw\_rom.bin的md5

结论：打patch前后的fw\_rom.bin的md5完全一样



测试case3:函数c在rom中，但函数c调用了在ram中的函数a，在ram函数a中调用了rom函数a。c需要打patch，patch\_c需要调用flash中的函数a。且函数patch\_c又被rom函数vt\_for\_patch调用，vt\_for\_patch又在flash代码被调用。

1. 定义函数指针c

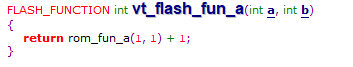




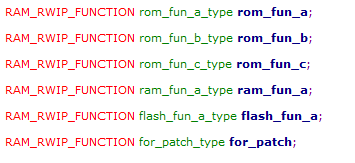
1. 定义ram函数a，和rom函数c，flash函数a



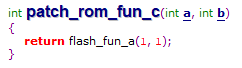




1. 定义函数指针



1. 定义函数c的patch



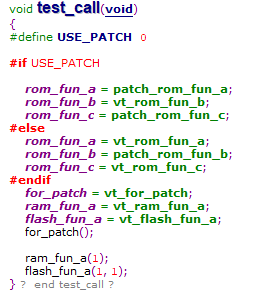
赋值并且分别测试打patch前后的fw\_rom.bin的md5

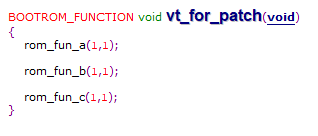
结论：打patch前后fw\_rom.bin的md5完全一样。



**测试case4**: 以上的case放在一起测试。

把USE\_PATCH分别改成1和0，rom和patch顺序穿插。





结论：前后的fw\_rom.bin的md5完全一样。





# 固化与Patch过程

以全局变量方案为例，解释开发过程及工作原理。

1. 分离CEVA栈为固化部分与非固化部分，独立成库。修改要固化库的代码
2. 编译成固化库如CevaLL，注意固化库仅需一次性编译。固化后，不再需要修改代码，也不需要编译。
3. 将CevaLL与其它固化的库如C库，打包成一个文件如ROM.bin，固化到ROM。内部用到的符号（如常数等）要全部固化到ROM中。如果链接在Flash中，则可能会漂移， ROM寻址错误。
4. 打补丁的代码，写入非固化库。比如替换函数如F()，将原VT.F设为New\_F()就可以。
5. 编译并与不再修改的固化库链接，形成固化文件ROM2.bin
6. 检查MD5，即比较文件。
   1. 由于CEVA固化库与其它库之间相互访问，都通过VT表时(相互引用的所有符号，包括全局变量、只读常数、函数等)，那么理论上无需考虑CEVA库自己的MD5是否一致。（这理论需进一步验证），当然可能存在漏洞，MD5检查更安全。
   2. 整个ROM.bin是需要检查的，除非它只含Ceva固化库。

# 实施方案

1. 经对比试验,在小机里可行的方案为JT+VT方案。Ceva的导出函数被加上rom\_前缀,如rom\_rwip\_init，JT即跳转表，实现一个个导出函数，如rwip\_init。跳转表参考一个个.c文件来分块（而每个.c仅导出非静态符号，如可extern的函数，这样可以减小VT表的大小并降低编码量），VT表是一个个全局变量，其指向一个个函数，其在运行初始化时被自动赋值。然后则在非固化代码中修改、保存以打补丁。
2. 外部引用，目前发现的外部引用有30多个，包括C库函数、Module中的变量等。除了C库因为已经固化，Module的引用也需要存放入结构体中，并非固化代码在启动时赋值。为了安全起见，固化版本确定时，将列表对外部所有函数的引用，以供全局ROM化参考。
3. 考虑到性能影响，将一个一个.c文件地实现VT表项并测试。比一口气做出整个VT表来说，方便测试分析。
4. 实现真实的补丁并测试。
5. CEVA栈的拆分细节，以及相关Feature选项，目前没有精确方案。但先据目前大致安排，做出一个整个可运行的VT表，然后再调整。
6. 先期由于并没有真正的拆分库，即ROM固化部分维持现有方案（即由BOOTROM\_FUNCTION等对函数的归段来决定），而VT中所谓的rom\_函数事实上也可能存在RAM、FLASH中。但这并不影响JT 或VT工常工作。当拆分库后，不再需要这些修饰了，而归段设置将放在LD script中(如 .rom={ Ceva.a(.text\*, .rodata\*) })，而且也较为简洁。当完成库的拆分，建立SDK不再每次编译CEVA栈，而是直接使用专人维护编译好的CEVA库进行链接，但仍可以保留源码或全新编译的选项。
7. 长远看，为了减小ROM补丁，还应当选用最新的CEVA版本，并做好充分测试。同时也需要对新版本做优化，以提升性能并节省功耗等。

# 实施方案示意

这个具体实施方案是将跳转表JT放入ROM中，虚表VT放入RAM中，以获得取最高性能。缺点是导出大量的符号（即VT表项，一个个函数指针，是全局变量）。但优点是只需在每个.c文件中指定VT函数。而JT与VT也仅需在一个文件(JumpTable.c)中集中实现，形成真正的跳转表及虚表。无需修改头文件，大大加快了编译调度速度。长远看，通过修改VT\_EXPORT\_FUNC宏就可以重定义虚表到Flash或ROM中以便对比测试。当Flash有足够的性能支持虚表时，跳转表也可放入Flash中，这时所有的VT表项都可声明为static，这时就无需忧虑导出符号太多了。

//仅示意一个函数 rw\_init

//Compiler.h 全局，ROM与FLASH都包含 ////////////////////////////////////////////

#ifdef CFG\_ROM\_VT

#define ROM\_VT\_FUNC(func) rom\_##func

…

#endif

// rwip.c 仅存在ROM内 ////////////////////////////////////////////

…

void ROM\_VT\_FUNC(rwip\_init)(uint32\_t error)

{

…

}

…

#ifdef CFG\_ROM\_VT

#define VT\_IMPORT\_PTR(var, TYPE) extern TYPE var[]; TYPE\* const rom\_vt\_##var = var;

#define VT\_IMPORT\_FUNC(func) extern void func(); \

void(\*pfn\_##func)() = &func; \

void vt\_##func() { pfn\_##func(); }

#define VT\_EXPORT\_FUNC(func) extern void rom\_##func(); \

void(\*pfn\_##func)() = &rom\_##func; \

void func() { pfn\_##func(); }

#if 0

VT\_IMPORT\_PTR(DebugE256SecretKey, const char)

VT\_IMPORT\_PTR(sw\_to\_hw, char)

…

VT\_IMPORT\_FUNC(memcmp)

…

VT\_IMPORT\_FUNC(aes\_encrypt)

…

#endif

// rwip.c

VT\_EXPORT\_FUNC(rwip\_rand\_init)

VT\_EXPORT\_FUNC(rwip\_init)

VT\_EXPORT\_FUNC(rwip\_reset)

VT\_EXPORT\_FUNC(rwip\_schedule)

// rwip\_driver.c

VT\_EXPORT\_FUNC(rwip\_time\_get)

…

#endif // CFG\_ROM\_VT

# 风险评估

1. 导出ROM的全部函数已经大降低了补丁风险，但有重大、特别修改也难以胜任：
   1. 需要修改某些参数，这些参数被即时编译至.txt段，甚至没有符号。
   2. 数组尺寸变化、结构体调整（如调整位置、Pack特性等）、函数接口调整。先要弄好
   3. 非常直观地，代码改动太多，导致补丁太大，并且很乱。
2. 当固化风险巨大时，存在不可抗力，只好全部归段到非固化代码中，或者另造芯片，或采用新栈

# 具体实施报告

至22年3月底，在SDK平台上，已成功移植CEVA栈1.1至TO2上，并通过真实的补丁验证

# 参数信息

|  |  |  |
| --- | --- | --- |
| 栏位 | 值 | 说明 |
| ROM大小 | 约119Kbytes  0-1d9f0 | 1. 含4KB bootroom，其它(约114K) 2. 含C库部分函数(或同名函数)，如memcpy、strlen等(192B) 3. 含LL库所有代码、只数读据，但不包含CEVA栈其它内容。含跳转表代码   (.text\* 52K, .rodata\* 2096B)  留有数K冗余，可在其后追加C库或高性能函数(自行解决补丁) |
| Flash大小 | 约278KB  0x20000000  ~ | 含CEVA栈Host\Modules，及系统其它组件 |
| 内存 .rwip  (ROM配套) | 4552B  VMA 0x10000000  LMA 0x20000198 | 1. 含LL层所有(.data\*, .bss\*. COMMON) 2. 含虚表VT(sdata.\*, 1664B, 约411表项)   可在其后追高性能函数或变量、缓冲区 |
| 内存 其它 | 约118K | .data\* 53K  .bass\*, 62K  .ram\_func\* 2856B  .retdata\* 48B  .func\_ret\* 288B |
| CEVA Features | 地址(PA, RA, PRA)  广告(含PA)  连接传数  加解密  不支持 Features:  CTE/AOA  ISO | (-DCFG\_APP)  (-DCFG\_APP\_PRF)  (-DCFG\_APP\_SEC)  (-DCFG\_APP\_HID)  (-DCFG\_APP\_DIS)  (-DCFG\_PRF\_BASS)  (-DCFG\_BLE\_EMB)  #(-DCFG\_CONLESS\_CTE\_RX)  #(-DCFG\_CONLESS\_CTE\_TX)  (-DCFG\_BLE\_ISO\_HW\_SUPPORT\_NOT\_PRESENT)  (-DCFG\_BLE\_HOST)  (-DCFG\_CON=10)  (-DCFG\_HL\_MSG\_API)  (-DCFG\_GATT\_CLI)  (-DCFG\_HOST\_KEY\_MGR)  (-DCFG\_MAX\_LE\_MTU=2048)  (-DCFG\_ATT\_VAL\_MAX=2048)  (-DCFG\_L2CAP\_COC\_CHAN\_PER\_CON\_NB=10)  (-DCFG\_L2CAP\_CHAN\_IN\_ENV\_NB=20)  (-DCFG\_PRF\_DISS)  ( -DCFG\_PRF\_BASS)  (-DCFG\_PRF\_HOGPD)  (-DCFG\_PRF)  (-DCFG\_NB\_PRF=10)  #(-DCFG\_AHITL)  (-DCFG\_NVDS)  (-DCFG\_NVDS\_RAM)  (-DCFG\_RF\_XRC443)  (-DCFG\_MODEM\_LR)  (-DCFG\_STATIC)  (-DCFG\_ALLROLES)  (-DCFG\_LE\_PWR\_CTRL)  #(-DCFG\_AOD)  (-DCFG\_ISO\_CON=3)  (-DCFG\_RAL=3)  (-DCFG\_ACT=12)  (-DCFG\_GAIA)  (-DCFG\_GPIO)  (-DCFG\_ECC\_P256\_SUPPORT)  #(-DCFG\_DBG) |
|  |  |  |

# 实际实现及使用说明

1. components\bt\CMakeLists.txt, 主要实现将CEVA栈分离成ll库, 其它归为host库。分库的主要作用是可将rodata(尤其是字符串)通过ld script固化到ROM。该文件中 CFG\_ROM\_VT 可开关固化功能。
2. components\bt\vtPatch目录
   1. JumpTable.c，即跳转表，ld 入 ll库.
   2. vtPatchs.c，即补丁程序，既使不打补丁，但只要开启了固化功能，就要调用其my\_patch()，因为要配置LL用到的外部符号。Ld 入 Host库
   3. 补丁，\*\_vtpatch.c，放在对应的固化.c文件同目录下，编译时，脚本会自动将它从LL库中剔出，转存入Host库中。因此打补丁时，需更新Host库及my\_patch()函数。
   4. LL升级。LL升级即意味着ROM变更，需要重新编译，导出、分析符号，工作量非常大。平时不宜编译LL库，尤其是现在LL库包含了复杂头文件，甚至与平台相关，平台一点改动都可能引起LL的指令、符号引用变化。

# 测试用例

# 自测

1. 功能测试
   1. 固化功能开关对比测试，广告、连接、配对、加密
   2. 补丁开关对比，广告、连接、配对、加密
      1. 关闭补丁，编译连接，将ROM.bin、Flash.bin等保存至其它目录。烧录，运行并测试。
      2. 打补丁，编译连接，只烧写Flash.bin，运行并测试。
   3. 调整LD脚本（或代码中调整某个符号大小），以移动非固化符号（要据规则），对比测试。

ROM

次引导区

可移动区

Flash

.rwip…

可移动区

RAM

1. 性能测试。固化与补丁插入了较多的跳转，这肯定会影响性能。但目前还没有好方法及专用仪器来分析性能。另外目前关闭了非ROM部分的函数优化，全都放入Flash中，也影响了性能。

# 专业测试

专业测试需要专业测试人员设计用例，通过专业仪器、规模测试，发现功能、性能、极限问题。

# 后续工作

近期

1. CEVA Features落实。
2. 整理与优化。目前只是完成移植与部分自测，VT表、LD Script等可进一步整理、调优。
3. 团队支持，如改进编译脚本，自动化MD5检查等
4. 进一步自测分析。团队可设计、实施专业测试用例。
5. 疑点跟进与验证，如CEVA部分寄存器访问、时钟相关修改，暂时还不能成功移植，尽管并不存在于ROM中。
6. 简单的NO ROM试验。将ROM代码归入Flash中运行分析，得出初步判断。当优化到一定程度，NO ROM方案也许可以初步运行。

中期

1. 代码、性能简单优化。没有ROM的代码全部Flash，但需要将中断、频繁调用、核心的函数归段、迁移至内存
2. CEVA栈真正库化及维护升级。CEVA栈将脱离平台单独编译，理论上，库只与指令集与编译器有关，与平台有关符号引用，将完全出导出，也即不会包含任何平台头文件及路径，平台只需链接相关库即可。为ROM打一次补丁时，则提供新的Host库给平台。
3. 基于协议分析可进行CODE PATH优化，据运行路径及图进行代码归段，以提升性能。也能为NO ROM提供支持。