AN4767
应用笔记

双存储区 STM32 微控制器的即时固件更新

简介

双存储区功能是多款 STM32 微控制器的通用特性。本文档旨在描述如何在客户应用中使用此功能。

本应用笔记中涉及的主要内容是现场升级，通过 X-CUBE-DBFU STM32Cube 扩展包介绍。

即时更新的主要优点是可以最大程度地缩短切换阶段的停机时间，即使在更新期间也可以执行要求严苛的实时任务。

尽管本文档仅直接描述了 STM32L0 系列 Cat5 器件、STM32L4 系列的入门系列和 USB OTG 器件，以及 STM32G4 系列 Cat.3 器件，但具有两个半独立存储器区的其他 STM32 微控制器也可以共享部分所述属性并采用类似的使用方式。

下列文件可在 *www.st.com* 获取，可作为参考：

- 参考手册 RM0367：“超低功耗 STM32L0x3 高级的基于 Arm® 的 32 位 MCU”
- 参考手册 RM0351：“STM32L4x5 和 STM32L4x6 高级的基于 Arm® 的 32 位 MCU”
- 参考手册 RM0440：“STM32G4xx 高级的基于 Arm® 的 32 位 MCU”
- AN2606 应用笔记：“STM32 微控制器系统存储器启动模式”
- AN4024 应用笔记：“STM32 安全固件升级（SFU）”
- AN4657 应用笔记：“使用 UART 进行 STM32L0 应用内编程”
- AN4894 应用笔记：“EEPROM 仿真技术和软件”

目录

表格索引

图片索引

1 定义

表 1. 缩略语列表

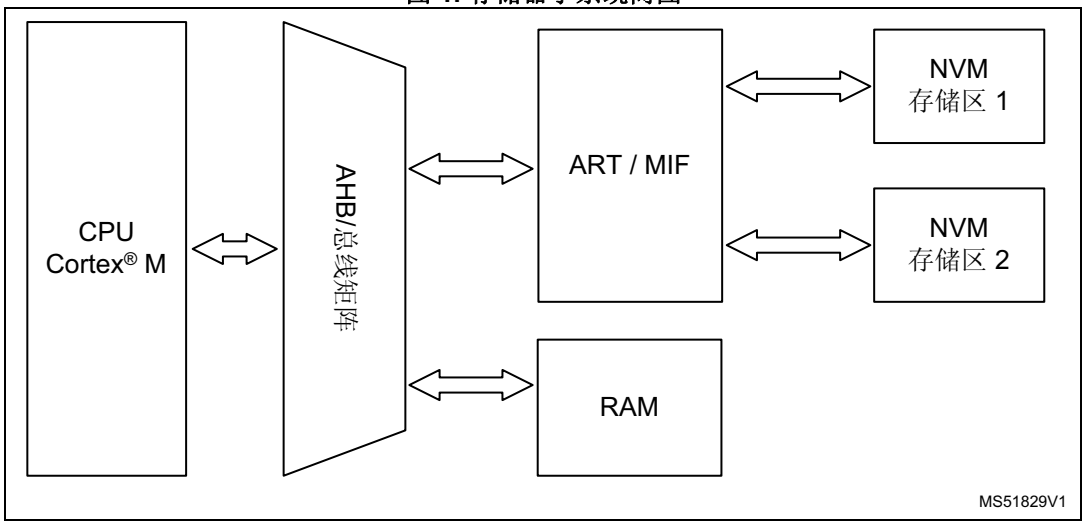
术语	说明
CPU	中央处理单元（MCU 的一部分）
EEPROM	电可擦除可编程只读存储器
ER	执行区域（链接器定义）
IAP	应用内编程
ISR	中断服务请求
IVT	中断向量表
LR	加载区域
MCU	微控制器
NMI	不可屏蔽中断
NVIC	嵌套向量中断控制器
NVM	非易失性存储器（EEPROM 或 Flash）
UART	通用异步收发器
USART	通用同步和异步收发器
VTOR	向量表偏移寄存器

2 存储器子系统概述

STM32 微控制器基于 Arm®(a) 内核。

关于存储器子系统的详细介绍，请参见相关参考手册。*图 1* 为存储器子系统的极简图示，其中只显示了常见的组成部分。

图 1. 存储器子系统简图



STM32L0 系列使用 Cortex® M0+ 内核。STM32L4 和 STM32G4 系列使用 Cortex® M4 内核，并且数据总线与指令总线相互独立。这意味着，后两个系列器件中的 AHB 总线矩阵更为复杂。另一个区别是 STM32L0 基于 EEPROM 技术。设计应用程序时，务必要牢记 STM32L0 的数据 EEPROM 与程序 Flash 存储器共享存储区接口。

本文档涵盖的所有 MCU 均在 NVM 接口中采用了某种缓存机制，STM32L0 系列比较简单，基于 Cortex M4 的两个系列则采用了更为高级的 ART Accelerator™。

双存储区存储器可以被配置为一个支持连续寻址的大容量 NVM 块（少数情况例外，本文档不作详细介绍）。将 NVM 配置为两个并行块时则具有显著优势，最重要的是可以一边对其中一个存储区执行写操作，一边对另一个存储区执行读操作（和取指操作）。这一点在执行更新时尤为重要，因为无需停止从程序 NVM 中执行代码。

在设计使用双存储区器件的应用程序时，可以选择多种方法来使用程序存储器的后半部分。

a. Arm 是 Arm Limited（或其子公司）在美国和/或其他地区的注册商标。

3 即时更新

即时更新也称为即时现场升级，这是一种允许用户在不干扰器件正常工作的情况下修改代码和配置的过程，与简单的 IAP 解决方案相比具有更多优势：

- 使用双存储区时可以更新加载程序的代码
- 即使加载失败，原始代码也仍然能够正常运行（操作可为“原子操作”）
- 无需定义加载程序状态，器件始终能够加载代码。

本文档通过几个示例详细介绍了如何在几微秒内将基于原始代码的所有操作转换为基于更新代码的所有操作。本文档还介绍了使用重新定位的向量表从 RAM 执行 ISR 的情况。这是降低中断延迟的常见解决方案，但在现场升级时执行起来比较棘手。

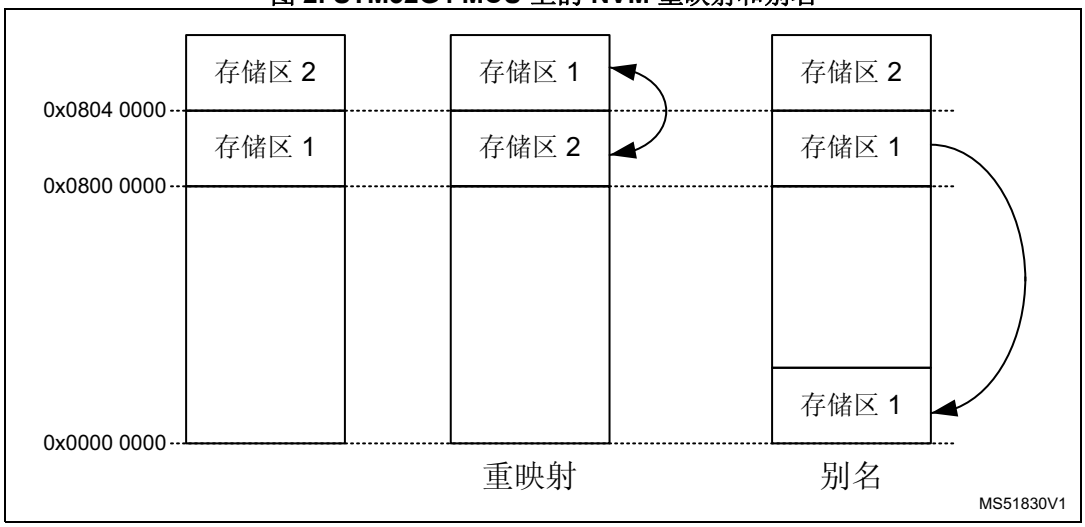
3.1 MCU 支持的特性

微控制器中实现了多种机制来确保固件正常执行，其中最重要的是能够一边对其中一个非易失性存储区进行写操作，一边从另一个非易失性存储区中执行代码。

3.1.1 存储器重映射切换

微控制器中有一个控制位（在 STM32L4 和 STM32G4 系列中标记为 FB_MODE，在 STM32L0 系列中标记为 UFB）可供用户代码访问。该位位于系统配置寄存器中，用于控制存储器映射和别名。此外，该位也用于双存储区自举机制，如果使用时足够谨慎，还可用于即时现场升级。

图 2. STM32G4 MCU 上的 NVM 重映射和别名



根据标志设置，存储区 1 或存储区 2 被映射到以 0x0800 0000 为起始地址的区域，并在地址 0x0000 0000 处设置别名。由于操作不影响 PC 和其他 CPU 寄存器，因此当该位发生翻转时，CPU 只会简单地从另一个存储区获取下一条指令。示例代码未链接到别名地址范围，因为如果涉及系统存储器中的 ST 自举程序，别名地址将保持为 0x0000 0000。

正常情况下，两个存储区中的代码均链接到以 0x0800 0000 为起始地址的区域。

3.1.2 可重新定位的中断向量表

CPU 可以配置 IVT 的偏移量。因此，可以通过软件定义更多的 IVT，并根据需要在它们之间进行切换。

向量表偏移量的默认值为 0x0000 0000，通常指向程序存储器的别名。

务必要为向量表预留足够的空间，每个中断向量需要 4 字节。此外，偏移地址值也存在限制，即必须与 512 字节的倍数对齐。

3.1.3 用户选项字节中的 BFB2 标志

该标志本质上是用于触发在复位时尝试从存储区 2 自举。

当存储区 1 中不存在代码时，务必保持 BFB2 标志置 1，以确保在意外断电时也可保持安全。当 BFB2 置 1 后，将激活系统自举程序以评估存储区 2 中是否存在代码，以便尽可能对其进行控制（详见相关参考手册或 AN2606）。之后，固件必须检测到存储区 1 的代码被替换，且程序从存储区 2 运行。

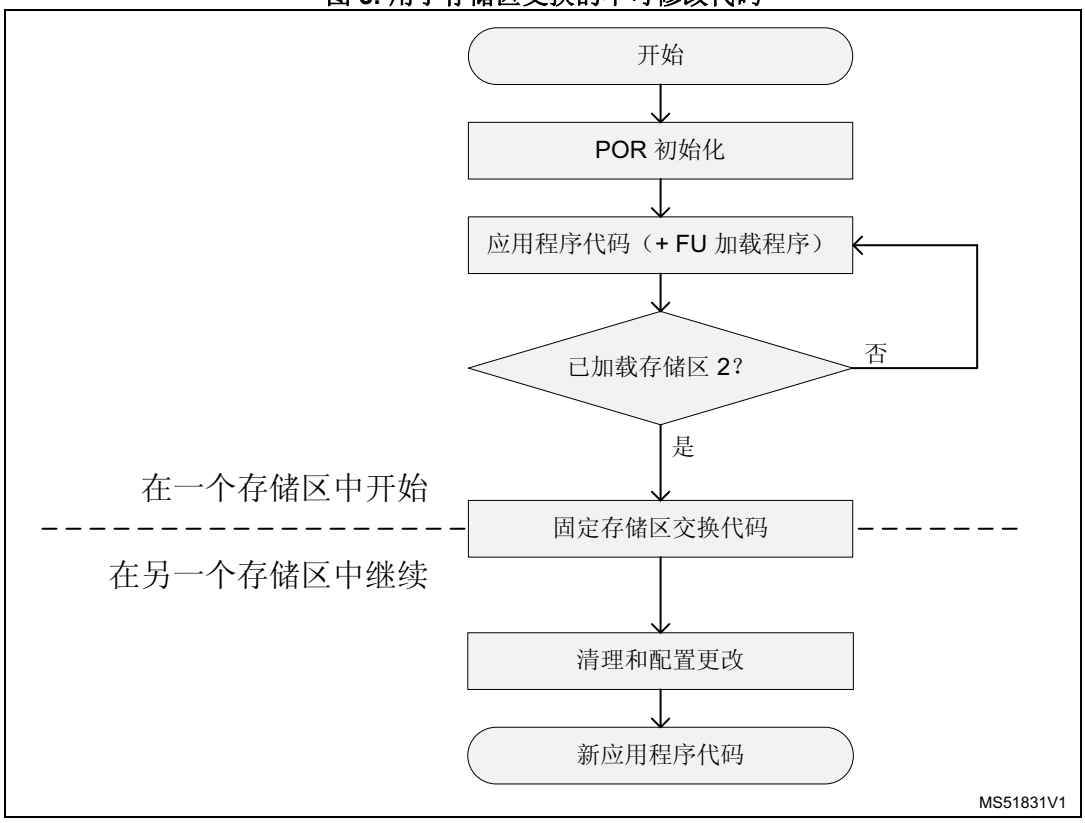
在该应用程序中，只有在从存储区 2 执行代码时发生意外断电的情况下，才使用 BFB2 作为故障安全机制。当存储区 1 不包含有效代码时，BFB2 必须保持置 1；当存储区 2 中的新代码复制到存储区 1 时，BFB2 清零。

4 不可修改的代码

如前文所述，在存储区交换期间，由于会发生存储区重映射，因此 CPU 将从另一个存储区获取下一条指令。推荐的方法是创建一段代码（最好在 main() 函数内，其中堆栈位于最低位置），然后将这段代码设为不可修改。

这段不可修改的固定代码随后将负责切换本身。如果在切换点附近，两个存储区中的代码不一致，则代码执行可能会变得不可预测，甚至会崩溃。

图 3. 用于存储区交换的不可修改代码



这段代码实际上可以是整个项目中非常小的部分（越小越好）。由于这段代码通常不能通过即时更新进行修改，因此必须确保没有 bug。

这段固定的存储区交换代码执行以下操作：

1. 禁止中断
2. 禁止并清空缓存
3. 修改中断向量表偏移量
4. 切换存储区
5. 恢复缓存配置
6. 使能中断

请记住，同一段源代码生成的二进制代码不一定相同。最好生成一个将链接到所有未来版本的库，或者保存一个目标文件，（可能）甚至需要仔细编辑生成的二进制代码。

5 RAM 中的代码

如果认为不可修改代码这种解决方案对于预期的应用程序来说风险过高，那么可以改为在 RAM 中执行一个函数来进行存储区交换，具体来讲就是选择性地操作堆栈指针，然后跳转到 Flash 存储器代码中的一个安全入口点。

在本文提供的项目示例中，RAM 仅用于存储 ISR 代码。需要进行即时现场升级的应用程序通常都这样做，因为这类应用程序需要在确定的时间内以最短的延迟响应信号。Flash 存储器的速度比 RAM 慢，因此时间关键型 ISR 需放入易失性存储器中，以便在分支后能够无延迟地获取新指令（至少在 ISR 函数调用 Flash 存储器中的函数之前完成）。

虽然 SRAM 通常存在多个不同的区域，但是不能像 Flash 的存储区那样进行重映射。正常情况下，替换易失性存储器中的代码需要先禁止中断，然后复制替换代码，最后再重新使能中断，如果发生 NMI，可能还有致命崩溃的风险。另外，当更新 RAM 内容时，可以使用存放在 Flash 存储器中的临时中断代码，但由于存在延迟问题，速度上会受到一定的影响。

我们提供的示例中使用的是第三种解决方案。该解决方案在 RAM 中定义了两个分区，每个分区专用于一个存储区的 ISR 代码。二进制代码与 RAM 中每个 ISR 函数的两个相同副本链接。项目可以根据自身执行时所在的 Flash 存储区来选用对应的一个函数副本。这样可以在不影响正常功能的情况下更新另一个未使用的函数副本，就像编程 Flash 存储器中的另一个存储区时一样。不过，在两个向量表（分别包含 ISR 函数的两个不同副本的地址）之间切换时使用的是向量表偏移量，而不是使用重映射。这样可以降低 NMI 所带来的风险，但前提是 NMI 处理程序必须保持不变，并且未调用其他存储器。

向量表偏移量存入在 CMSIS 标准中定义为 SCB → VTOR 的 Arm 寄存器。该寄存器可供用户访问。

对于 RAM 中的非 ISR 函数，可以定义一个类似的引用表（前提是分支结构较为复杂，将代码存入 RAM 中仍更具优势）。我们提供的示例中未使用这类代码。

6 易失性数据

现场升级的目标是无需系统复位即可转换到新的代码版本。复位时，启动代码（通常默认由开发工具提供商提供）会先初始化 RAM，然后才将控制权交给用户代码。初始化阶段通常会擦除零初始化 RAM 区域的地址，然后向所谓的常量初始化区域复制默认值（存放在 NVM 中）。

这些操作大多在“后台”进行，只有在需要初始化的全局变量数量过多的情况下，才会注意到启动时间较长。

使用即时更新时，开发人员必须对易失性数据结构的位置和内容加以管理。至少需要检查映射文件 (map file)，并查看 RAM 地址范围是否发生变化。

6.1 避免易失性数据结构发生变化

如果新的映射文件显示 RAM 中数据的位置发生严重不可取的意外变化，则可以通过多种方法将数据恢复原位。遗憾的是，没有一种通用的方法，必须根据使用的工具链来决定采取哪种办法。链接器决定着存储器中数据的位置，但所有的链接器控制均只适用于特定的产品。

虽然避免更改编译器和链接器的设置可以防止一部分修改，但更为可靠的解决方案是将源文件完全删除，然后用原始版本生成的目标文件代替。

新添加的数据可以单独移入之前未使用的存储器分区。如果删除了一个变量，导致地址移位，只需用一个未使用的相同大小的虚拟变量代替即可。

在任何情况下，都要参考链接器文档来确定是否有方法可以帮助简化补丁的开发。

6.2 管理对易失性数据结构的修改

关于修改，可以定义五种基本的严重性级别，详见*表 2*。

表 2. 修改严重性级别

严重性	情况	更新后的操作
0	RAM 的映射文件相同	无操作
1	删除了变量	无操作（但需要再次检查地址）
2	添加了新变量	按需进行初始化
3	修改了地址	移动数据
4	修改了结构	必须通过代码将旧数据转换为新格式

推荐的解决方案是在新代码中添加一个专用函数，即在更新配置完成加载后只执行一次的初始化程序。这段只运行一次的代码将在实际功能代码需要访问变量之前处理 RAM 内容。

本文档不涵盖动态分配堆存储器的情况。在即时升级的情况下，无法保证 RAM 堆的正常功能，最好避免使用。具体实现在实际使用中可能存在限制。

6.3 时序

加载过程完成后，不必立即执行存储区交换。加载的数据将在另一个存储区中保持可用状态，直到固件确定时机合适为止。

在实际的应用中，可能会存在多种周期长短不一的不同任务。在某些情况下，可以确定一个极少可能收到时间关键型中断的时隙。最高优先级的中断通常是最重要的，例如控制环。如果可以预测到该中断将在一段足够长的时间窗口内不会再次出现，那么这就是启动存储区交换的合适时机。

在 MCU 上运行的系统内部的标志可以是用于触发继续升级过程的信号。它既可能是一个简单的位，也可能包含一个时间戳，具体取决于软件系统的架构。

为了最大程度地缩短交换所需的时间，可设置最大系统时钟。

6.4 固件示例

X-CUBE-DBFU 固件包中包含一个简单的示例解决方案，专门与选定的 STM32 评估板搭配使用。它与 IAP 应用程序密切相关，但是具有重要区别。它只有一个固件，且两个存储区都使用相同的固件，既是加载程序也是应用程序。

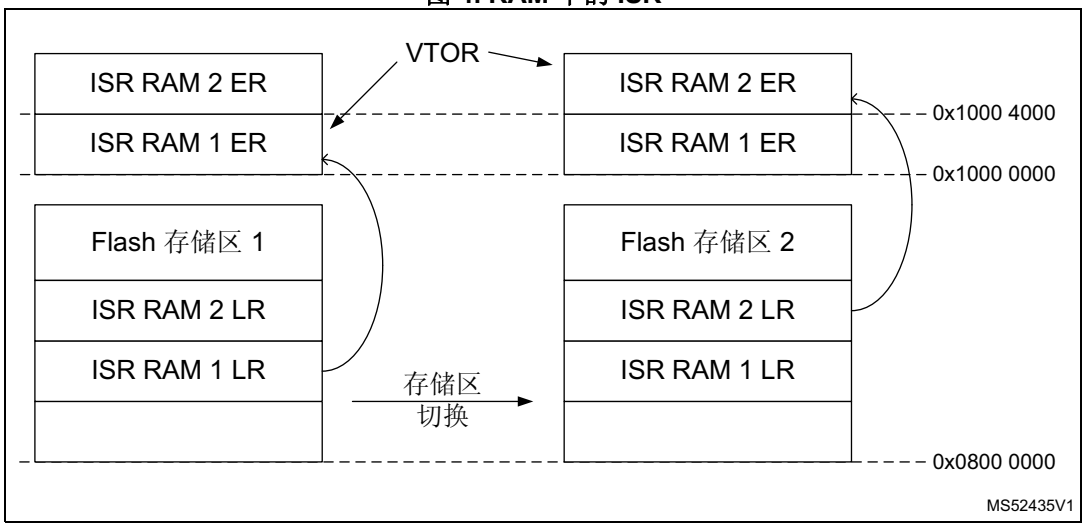
6.4.1 示例解决方案架构

本示例解决了其中一项挑战，即从 RAM 执行中断服务程序。不同系列产品的 RAM 配置各不相同（STM32L0 系列配有一个 RAM 块，STM32L4 系列配有两个不同的 RAM 区域，STM32G4 系列配有三个不同的 RAM 区域）。这并不影响示例的功能，因为 RAM 不支持双存储区。这意味着不能通过硬件重映射交换两个 RAM 地址范围。

本示例使用的是*第 5 节：RAM 中的代码*所述的解决方案。另一个表在每个特定项目的 stm32xxxx_it2.c 源文件中声明。当使用本示例作为后续开发的起点时，务必要确保中断程序代码的两个副本能够持续更新。与不可修改的代码部分不同，两个副本的二进制代码不必完全相同，只需功能相同即可（除非有意让产品在从存储区 2 运行时呈现不同的行为）。

代码中执行的第一项操作是设置向量表。在本示例中，这项操作发生在 main() 函数中，但也可以选择在 system_stm32xxxx.c 中修改 SystemInit()。当然，首先必须初始化实际的向量表，因为它保存在易失性存储器中，然后相应地修改 VTOR 值。

图 4. RAM 中的 ISR



6.4.2 硬件设置

使用 RS-232 串行线，将 USART2 端口连接到 PC（推荐使用作为虚拟 COM 端口的通用 USB 适配器）。使用支持 YMODEM 协议的终端应用程序。Tera Term 是 Windows® HyperTerminal 的替代产品，其性能可靠且免费提供。

已使用 Tera Term 版本 4.84 测试了示例功能。

配置通信速度为 115200 Bd，8 个数据位，无奇偶校验，1 个停止位，接着启动开发板。

6.4.3 示例操作

采用终端软件提供的用户界面易于操作。示例固件首先显示标题和菜单。按 PC 键盘上的数字即可选择选项。

1. 将文件下载到另一个存储区

   选择第一个菜单选项，即表示开始将二进制文件下载到另一个存储区中。擦除和重写另一个存储区先前的内容。文件大小受存储器大小的限制。选择要下载的文件，解密并写入该文件。使用 YMODEM 协议继续进行通信。

2. 擦除另一个存储区的内容

   此选项使另一个存储区的内容无效。

3. 重写另一个存储区的内容

   来自活动存储区的代码被复制到相对的存储器存储区。

4. 检查另一个存储区的完整性

   此选项启动对另一个存储区内容的检查。对于 L0 系列，示例固件将代码 CRC 完整性值存储在 EEPROM 数据存储器中。对于其他器件，检查内容仅限于简单存在性检查。

5. 切换存储区

   如果在另一个存储区似乎存在功能代码，则重写向量表并切换存储区（如*第 3.1.1 节：存储器重映射切换*所述）。

6. 切换系统存储区

   对选项字节进行编程，以修改 BFB2 位的值（从存储区 2 自举，如*第 3.1.3 节：用户选项字节中的 BFB2 标志*所述）。

本文所述的即时升级序列可以使用序列 *1*, *4*, *5*, *6* (BFB2 ON) 和 *3*, *4*, *5*, *6* (BFB2 OFF) 来描述。模拟失效状态时包含选项 *2*（擦除）。

6.4.4 加密选项

固件升级文件中包含的 IP 对于所有者而言可能是宝贵的信息，因此所有者可能会担心代码保密性问题，这一点在情理之中。结合完整性保护功能，即使是对称加密也可以防止引入伪造或欺诈性固件。

多个 ST 微控制器包括可选的 AES 硬件加速器外设选项。此时，可以使用 STM32HG484、STM32L486 和 STM32L083 分别代替评估板上的 STM32G474、STM32L476 和 STM32L073，以便使用加密功能。

6.4.5 加密二进制码

此示例中使用的加密方案是在计数器 (CTR) 模式下使用明文 AES。

这种方法的优点是不需要填充，只加密有效负载。虽然这与现场升级案例无关，但值得一提的是，计数器模式还能够避免传播通信错误（仅在通信过程中损坏的字节会在解密消息中被破坏）。

此外，无需开发专用加密实用程序，任何公开可用的加密库都能够生成计数器模式暂存器，并在暂存器和数据之间进行 XOR。

如果用户不确定如何在计算机上准备输入，则必须遵循以下步骤：

1. 下载开源库 Crypto++® (www.cryptopp.com)
2. 使用您偏好的编译器来编译 cryptest.exe。
3. 使用 ./cryptest.exe v（v 选项：验证套件）来测试库
4. 如果所有测试都通过，将显示消息“所有测试通过！”
5. 生成二进制文件并将其放在 Cryptopp 库下
6. 使用参数 cryptest.exe ae <HexKey> <HexIV> Project.bin Firmware.aes 调用 cryptest.exe。

项目中包含一个批文件，旨在消除对密钥格式的任何疑问。

cryptest.exe 将生成一个 .aes 文件，后者将用于具有 AES 外设的器件。

6.4.6 配置解密

在项目中添加或取消注释 #define ENCRYPT。这仅适用于具有 AES 外设的器件（在本例中为 STM32L083/STM32L486/STM32G484）。

将占位符密钥和初始化向量替换为 main.c 中的任何其他值，即可定制此项目。

6.4.7 简单解决方案限制

本示例中使用的基本对称加密解决方案存在一些局限性。

它对现场的所有器件使用相同的密钥。单密钥的广泛使用会导致加密系统更易于受到攻击。它不仅让攻击者有更多机会获取密钥，而且一旦密钥泄露，所有器件都处于公开状态。使用代码的某些属性（例如版本）导出密钥，可以在一定程度上减少此影响。

此外，身份验证仅依赖于知道密钥的这一事实，这并不能完全减少提供非正版代码的可能性。

事实上，代码的前几个字节（由初始堆栈指针和向量表组成）大部分是可预测的（“已知的明文攻击”），这可能导致会削弱加密属性。在加密之前向“明文”添加随机序列（“salt”），在解密后忽略“salt”，即可轻松解决这一弱点。

6.4.8 其他加密选项

保护代码不仅涉及加密算法，而且涉及复杂的加密系统，包括在所有产品生命阶段（从设计和开发，到制造和维修，再到停用和回收）实施的安全要求。

还必须认真考虑身份验证角色和完整性保护。

意法半导体能够提供顶级嵌入式安全解决方案。

6.5 其他实施选项

本节介绍了实施现场升级机制时常用的各种技术，但在 X-CUBE-DBFU 示例中没有使用，旨在保持简单性和通用性。

6.5.1 现场升级文件

在我们提供的示例中，加载到系统的文件是一个普通的二进制文件，顶多经过加密处理。正常情况下，不推荐使用这种文件。文件应至少标记版本和产品标识，以防止将不兼容的代码加载到产品中，避免意外降级，或者仅仅为了增添便利性。

现场升级包通常不仅包含代码，还包含数据。可能有的部分包含更新后的新配置，有的部分包含新添加的 RAM 变量的初始化值。

在更为复杂的系统中，有些数据应在更新成功后发送到 PCB 上的其他芯片，这类数据也可以包含在同一个现场升级包中。

即时更新过程也可能识别出应立即复制到 RAM 的代码部分，以及有关更新后操作的代码部分的跳转点，这部分代码只执行一次，负责切换和清理。

6.5.2 NVM 中的数据

使用双存储区的另一个普遍原因是可以使用另一个存储区来存储数据，这通常使用 EEPROM 仿真算法来实现。本文档中所述的即时升级机制可以与 AN4894 中所述的 EEPROM 仿真结合使用。

7 结论

如果使用得当，双存储区特性能够提供适用于广泛应用的众多优点。

虽然本应用笔记中所提供的示例和说明都比较基础，但可作为实际项目的功能构件。

务必要彻底测试最终的解决方案，以确保即时更新代码稳定可靠。

8 版本历史

表 3. 文档版本历史

日期	版本	变化说明
2016 年 9 月 29 日	1	初始版本。
2016 年 10 月 31 日	2	更新了文档标题。
2019 年 5 月 15 日	3	引入了 STM32G4 系列。 重新编排了文档内容，所有章节都有变化，包括标题。