Application Note

在 TI TDA4x MCU R5F 上使用 eMMC FATFS 进行运行时代码覆盖

Johnny Kim

摘要

TI TDA4x 器件支持各种各样的嵌入式应用，这些应用可能需要的软件功能超出了驻留运行时环境的存储占用空间。运行时代码覆盖提供了一种机制，仅在需要时动态加载可执行代码，从而使多个软件模块可以共享一个共用执行区域。

本应用手册介绍了一种在 TI TDA4x MCU R5F 内核上使用 eMMC FATFS 的运行时代码覆盖方法。可执行有效载荷作为文件存储在 eMMC 用户数据区域 (UDA) 中，并在运行时动态加载到可重用的 SRAM 执行区域。与存储器映射的存储器件不同，eMMC 作为块存储器件被访问，不支持由 MCU R5F 内核直接执行代码。因此，可执行有效载荷必须通过 FATFS 层读取，加载到指定的 SRAM 覆盖区域中，并通过运行时覆盖机制执行。驻留运行时负责有效载荷管理、FATFS 访问、代码重定位、高速缓存维护和函数指针执行。

该实现方案通过使用单个 SRAM 覆盖槽从 eMMC FATFS 加载并执行多个有效载荷来验证。不同的有效载荷可反复重用同一执行区域，而无需为每个软件模块分配专用的存储器。该方法演示了一种在 TI TDA4x MCU R5F 上实现基于存储的可执行文件加载和运行时功能扩展的实用方法。

内容

商标

所有商标均为其各自所有者的财产。

1 简介

TI TDA4x 器件支持在 MCU R5F 内核上运行的各种嵌入式应用。随着软件复杂性的增加，除了驻留运行时环境提供的功能外，可能还需要诊断、维护实用程序、恢复服务和功能特定模块等其他功能。

常见的方法是将所有可执行代码静态链接到一个应用映像中。虽然该方法易于实现，但它会为每个软件组件永久分配存储，而无论该组件在运行时是否活跃使用。随着软件功能数量的增加，应用的存储占用空间也会增加。

运行时代码覆盖提供了一种替代方法，即仅在需要时加载可执行代码。无需将所有软件模块一直驻留在存储器中，可执行有效载荷可存储在外部存储器中，并在运行时动态加载到共享执行区域中。这使得多个软件模块能够重用同一存储区域，同时减少驻留存储占用空间。

本应用手册演示了一种在 TI TDA4x MCU R5F 内核上使用 eMMC FATFS 的运行时代码覆盖方法。可执行有效载荷作为文件存储在 eMMC 用户数据区域 (UDA) 中，并在请求执行时加载到可重用的 SRAM 执行区域。与存储器映射的存储器件不同，eMMC 作为块存储器件被访问，不支持由 MCU R5F 内核直接执行代码。因此，可执行有效载荷必须通过 FATFS 层读取，复制到可执行存储器中，并通过运行时覆盖机制执行。

本应用手册中介绍的实现验证了使用单个 SRAM 覆盖槽在运行时加载和执行多个有效载荷。本文档介绍了 TI TDA4x 器件的覆盖架构、有效载荷格式、软件实现、运行时执行流程以及验证程序。

2 运行时代码覆盖后台

2.1 TDA4x 的存储器架构

TI TDA4x 器件提供可供 MCU R5F 内核使用的多种存储器资源。其中包括片上 SRAM 区域、紧密耦合存储器 (TCM)、OCMC 存储器和外部 DDR 存储器。

对于许多软件架构，可执行代码存储在外部存储器中，并在系统初始化期间加载到 DDR 中。这种方法为复杂的软件应用和多个软件模块提供了足够的存储容量。

除了存储器资源外，TDA4x 器件还支持 eMMC、OSPI NOR 闪存和 SD 卡等各种存储器件。这些存储器件可用于存储软件映像、配置数据和应用特定内容。

2.2 静态代码分配的挑战

共用软件部署模型将所有可执行模块静态链接到一个应用映像中。在系统初始化期间，整个可执行映像会加载到存储器中，并在整个运行时内保持驻留状态。

这种方法虽然简化了软件管理，但它会为每个软件功能永久分配存储，而无论实际使用情况如何。随着软件模块数量的增加，驻留运行时环境的存储占用空间也会增加。

可能不需要持续驻留的软件组件示例包括

- 诊断功能
- 恢复实用程序
- 工厂测试功能
- 维修工具
- 功能特定应用模块

同时驻留所有此类组件可能会导致存储器利用率低下。

2.3 为什么选择运行时代码覆盖？

运行时代码覆盖是一项技术，该技术允许仅在需要执行时将可执行代码加载到存储器中。

可执行有效载荷存储在外部存储器中，并在运行时动态加载到共享执行区域中，而不是将所有软件模块一直驻留在存储器中。执行完成后，同一存储器区域可以被另一个有效载荷重用。当软件功能持续扩展，同时驻留运行时存储占用空间必须保持在可控范围内时，此方法特别有用。与在存储器映射模式下运行的 OSPI NOR 闪存不同，eMMC 作为块存储器件被访问，不支持由 MCU R5F 内核直接执行代码。因此，在执行之前，必须首先通过 FATFS 层读取存储在 eMMC 中的可执行代码，并将其复制到可执行存储器中。本应用手册中介绍的运行时代码覆盖方法使用 eMMC FATFS 作为有效载荷存储机制，并使用可重用的 SRAM 区域作为执行目标。这使得多个可执行模块能够共享一个共用的执行区域，同时保持较小的驻留运行时占用空间。

3 运行时代码覆盖方法

3.1 概述

本应用手册中介绍的运行时代码覆盖方法将软件分为两类：驻留运行时和瞬态覆盖有效载荷。

驻留运行时在整个系统执行过程中保持加载状态，并负责有效载荷管理、FATFS 访问、代码重定位、高速缓存维护和执行控制。覆盖有效载荷作为 eMMC FATFS 中的可执行包文件存储，仅在请求执行时加载。

所有有效载荷共享一个共用的 SRAM 执行区域，而不是为每个软件模块保留专用存储器。同一 SRAM 区域可以在运行时被不同的有效载荷反复重用。

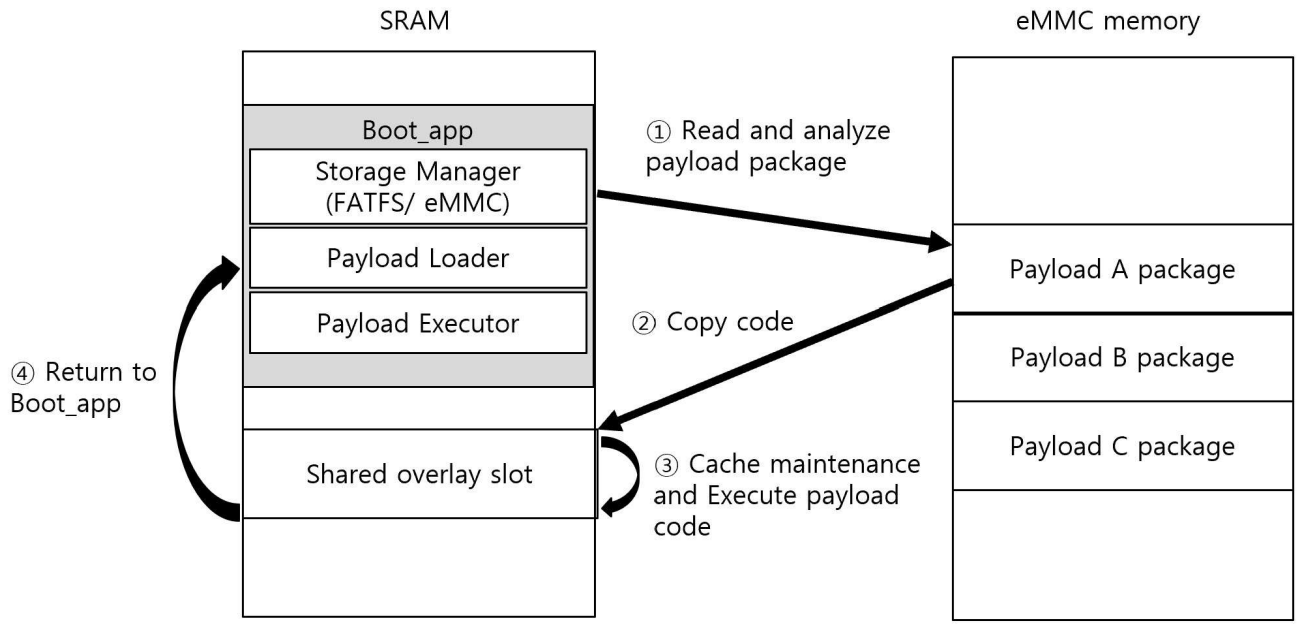

图 3-1. 运行时代码覆盖方法和控制流程

3.2 驻留运行时

驻留运行时在应用的整个生命周期内保持活动状态。在此实现中，驻留运行时集成到在 MCU R5F 内核上运行的 boot_app 应用中。

驻留运行时执行以下功能

- eMMC 初始化
- FATFS 访问
- 有效载荷文件管理
- 覆盖包解析
- 运行时映像加载
- 高速缓存维护
- 函数指针执行

驻留运行时不包含各个覆盖有效载荷的可执行功能。相反，它提供了动态加载和执行有效载荷所需的基础设施。

3.3 覆盖有效载荷包

不需要一直驻留的软件功能被打包为覆盖有效载荷。

本演示中使用了三个有效载荷

- payloadA.pkg
- payloadB.pkg
- payloadC.pkg

每个有效载荷包都包含可执行代码和一个入口点；在加载到 SRAM 覆盖区域后，驻留运行时可以调用该入口点。

有效载荷作为常规文件存储在 eMMC FATFS 分区中，可以独立加载和执行。

3.4 共享 SRAM 覆盖区域

保留用于覆盖执行的专用 SRAM 暂存区域。

示例配置

- 基址：0x41C70000
- 大小：16KB

在任何给定时间，只有一个有效载荷占用 SRAM 覆盖区域。请求新的有效载荷后，以前的有效载荷将被新加载的有效载荷替换。这样便可让多个可执行模块共享同一个执行存储器，而无需为每个模块单独分配存储器。

3.5 运行时覆盖序列

运行时覆盖过程包含以下步骤。

1) 从 eMMC FATFS 打开有效载荷文件。

2) 读取并验证覆盖包标头。

3) 将可执行代码段复制到 SRAM 覆盖区域。

4) 对加载的代码映像执行高速缓存维护。

5) 计算运行时入口地址。

6) 通过函数指针执行有效载荷。

7) 将控制权交还给驻留运行时。

每当请求不同的有效载荷时，都会重复相同的序列。

4 运行时代码覆盖架构

4.1 软件架构

运行时代码覆盖框架包含四个主要软件组件。

- 驻留运行时
- FATFS 层
- 覆盖加载器
- 覆盖有效载荷

驻留运行时管理有效载荷请求并协调覆盖执行过程。FATFS 层为存储在 eMMC UDA 中的有效载荷映像提供文件访问。覆盖加载器负责解析有效载荷包，将可执行代码加载到 SRAM 中，并准备执行环境。覆盖有效载荷包含在运行时动态加载的可执行功能。这些组件协同工作，支持使用共享 SRAM 覆盖区域加载和执行基于存储的可执行文件。

4.2 覆盖包格式

每个可执行有效载荷都打包为覆盖映像，包含一个包标头，后跟可执行代码段。包标头提供运行时加载器所需的元数据，用于验证有效载荷、确定可执行代码的大小以及定位运行时入口点。当前实现采用以下包格式。

```
typedef struct BootApp_OverlayPkgHeader_s
{
    uint32_t magic;
    uint32_t totalSize;
    uint32_t codeSize;
    uint32_t entryOffset;
    uint32_t flags;
    uint32_t reserved[3];
} BootApp_OverlayPkgHeader;
```

4.3 存储器布局

可执行有效载荷存储在 eMMC FATFS 中，并在运行时复制到专用的 SRAM 执行区域中。

覆盖执行槽位于

- 基址：0x41C70000
- 大小：16KB

在任何给定时间，覆盖区域中仅驻留一个有效载荷。每当加载不同的有效载荷时，该区域就会被重用。

4.4 运行时映像加载

由于 eMMC 作为块存储器件被访问，因此无法从存储介质直接执行可执行代码。

请求有效载荷时，覆盖加载器从有效载荷包中读取可执行代码段，并将其复制到 SRAM 覆盖区域中。重定位后，使用 SRAM 基址和存储在包标头中的入口偏移量来计算运行时入口地址。然后从 SRAM 执行重定位的代码。

4.5 运行时执行

可执行代码重定位到 SRAM 后，会执行高速缓存维护以验证指令一致性。

运行时入口地址的计算方式如下

```
Entry Address = Overlay Base Address + Entry Offset
```

随后通过函数指针执行有效载荷。完成后，控制权将交还给驻留运行时，从而允许将另一个有效载荷加载到同一 SRAM 覆盖区域中。这种机制使多个可执行模块能够共享一个执行槽，同时保持较小的驻留运行时占用空间。

5 演示实现

5.1 软件组织

运行时代码覆盖演示在 TI Processor SDK RTOS 包中包含的“boot_app”示例内实现。

该实现包括

- 驻留运行时集成
- 基于 FATFS 的有效载荷访问
- 覆盖包加载器
- SRAM 覆盖管理器
- 有效载荷执行框架

覆盖有效载荷生成实用程序单独提供，用于创建供运行时加载的可执行有效载荷包。

5.2 覆盖 SRAM 配置

保留用于运行时覆盖执行的专用 SRAM 暂存区域。

当前演示使用

- 覆盖基址：0x41C70000
- 覆层大小：16KB

以下是示例链接器命令文件“emmc_overlay_payload.cmd”。

```
--entry_point=emmc_overlay_payload_entry
--stack_size=0x400
--heap_size=0x0

MEMORY
{
    SRAM_OVERLAY_SLOT (RWX) : origin = 0x41C70000, length = 0x00004000
}

SECTIONS
{
    .text       : {} palign(64) > SRAM_OVERLAY_SLOT
    .rodata     : {} palign(64) > SRAM_OVERLAY_SLOT
    .const      : {} palign(64) > SRAM_OVERLAY_SLOT
    .cinit      : {} palign(64) > SRAM_OVERLAY_SLOT
    .pinit      : {} palign(64) > SRAM_OVERLAY_SLOT
    .init_array : {} palign(64) > SRAM_OVERLAY_SLOT
    .data       : {} palign(64) > SRAM_OVERLAY_SLOT
    .bss        : {} palign(64) > SRAM_OVERLAY_SLOT
    .sysmem     : {} palign(64) > SRAM_OVERLAY_SLOT
    .stack      : {} palign(64) > SRAM_OVERLAY_SLOT
}
```

覆盖区域用作所有有效载荷的临时执行槽。一次只有一个有效载荷占用覆盖区域。加载新的有效载荷时，之前的有效载荷将被覆盖。

5.3 有效载荷生成

可执行有效载荷作为独立的覆盖包生成。

下面是一个示例“package_overlay.py”脚本，用于从“bin”文件生成覆盖包。

```
#!/usr/bin/env python3
import struct
import sys
from pathlib import Path

# Must match BootApp_OverlayPkgHeader / boot_app side checks
BOOTAPP_OVERLAY_MAGIC = 0x4F56524C  # 'OVRL'

# Header layout (32 bytes)
#   uint32_t magic;
#   uint32_t totalSize;
#   uint32_t codeSize;
#   uint32_t entryOffset;
#   uint32_t flags;
#   uint32_t reserved[3];
HEADER_FMT = "<8I"
HEADER_SIZE = struct.calcsize(HEADER_FMT)

def make_pkg(bin_path: Path,
             out_path: Path,
             entry_offset: int = 0,
             flags: int = 0,
             prepad: int = 0) -> None:
    code = bin_path.read_bytes()

    if len(code) == 0:
        raise ValueError(f"Empty binary: {bin_path}")
    if entry_offset < 0:
        raise ValueError(f"entry_offset must be >= 0: {entry_offset}")
    if prepad < 0:
        raise ValueError(f"prepad must be >= 0: {prepad}")

    # For XIP packages, TI objcopy emits the binary without preserving the
    # absolute load address. If the linker aligns .text from header_end
    # 0x...20 to 0x...40, the raw binary still starts at file offset 0.
    # Add explicit padding after the OVRL header so the first binary byte is
    # physically placed at the same address used by the linker.
    payload = (b"\x00" * prepad) + code
    code_size = len(payload)
    total_size = HEADER_SIZE + code_size

    header = struct.pack(
        HEADER_FMT,
        BOOTAPP_OVERLAY_MAGIC,  # magic
        total_size,             # totalSize
        code_size,              # codeSize, including any XIP prepad
        entry_offset,           # entryOffset from first byte after header
        flags,                  # flags
        0, 0, 0                 # reserved[3]
    )

    out_path.write_bytes(header + payload)

    print(f"Created {out_path}")
    print(f"  totalSize   : {total_size}")
    print(f"  codeSize    : {code_size}")
    print(f"  entryOffset : {entry_offset}")
    print(f"  flags       : {flags}")
    print(f"  prepad      : {prepad}")

def main() -> int:
    if len(sys.argv) not in (3, 4, 5, 6):
        print("Usage: package_overlay.py <input.bin> <output.pkg> [entry_offset] [flags] [prepad]")
        return 1

    in_path = Path(sys.argv[1])
    out_path = Path(sys.argv[2])

    entry_offset = int(sys.argv[3], 0) if len(sys.argv) >= 4 else 0
    flags = int(sys.argv[4], 0) if len(sys.argv) >= 5 else 0
    prepad = int(sys.argv[5], 0) if len(sys.argv) >= 6 else 0
```

```
    make_pkg(in_path, out_path, entry_offset, flags, prepad)
    return 0

if __name__ == "__main__":
    raise SystemExit(main())
```

下面是一个示例 shell 脚本“build_payloads.sh”，用于生成有效载荷 payloadA.pkg、payloadB.pkg 和 payloadC.pkg。

```
#!/usr/bin/env bash
set -e

SCRIPT_DIR=$(cd "$(dirname "$0")" && pwd)
OUT_DIR=${OUT_DIR:-$SCRIPT_DIR/out}
TIARMCLANG=${TIARMCLANG:-tiarmclang}
TIARMLNK=${TIARMLNK:-tiarmlnk}
TIARMOBJCOPY=${TIARMOBJCOPY:-tiarmobjcopy}
SLOT_ADDR=${SLOT_ADDR:-0x41C70000}

mkdir -p "$OUT_DIR"

prepare_linker_cmd() {
    local addr=$1
    local out_cmd=$2
    sed "s/0x41C70000/${addr}/g" "$SCRIPT_DIR/emmc_overlay_payload.cmd" > "$out_cmd"
}

build_one() {
    local tag=$1
    local src=$2
    local obj="$OUT_DIR/${tag}.o"
    local elf="$OUT_DIR/${tag}.elf"
    local bin="$OUT_DIR/${tag}.bin"
    local pkg="$OUT_DIR/payload${tag^^}.pkg"
    local lnk="$OUT_DIR/${tag}.cmd"

    echo "[build] ${tag^^}: slot=${SLOT_ADDR} src=${src}"
    prepare_linker_cmd "$SLOT_ADDR" "$lnk"

    $TIARMCLANG -mcpu=cortex-r5 -mthumb -Oz -ffreestanding -fno-builtin -nostdlib \
        -c -I"$SCRIPT_DIR/../include" "$SCRIPT_DIR/$src" -o "$obj"
    $TIARMLNK -o "$elf" "$obj" -c "$lnk" --cinit_compression=off
    $TIARMOBJCOPY -O binary "$elf" "$bin"
    python3 "$SCRIPT_DIR/package_overlay.py" "$bin" "$pkg" 0x0 0x0 0x0
}

build_one a emmc_overlay_payload_a.c
build_one b emmc_overlay_payload_b.c
build_one c emmc_overlay_payload_c.c

cat <<MSG

Done. Copy these files to the FAT partition root used by boot_app:
  $OUT_DIR/payloadA.pkg -> 0:/payloadA.pkg
  $OUT_DIR/payloadB.pkg -> 0:/payloadB.pkg
  $OUT_DIR/payloadC.pkg -> 0:/payloadC.pkg
MSG
```

每个包均包含

- 覆盖包标头
- 可执行代码段
- 运行时入口点信息

使用以下命令可生成有效载荷包

```
SLOT_ADDR=0x41C70000 ./build_payloads.sh
```

生成的包文件随后被复制到 eMMC FATFS 分区。

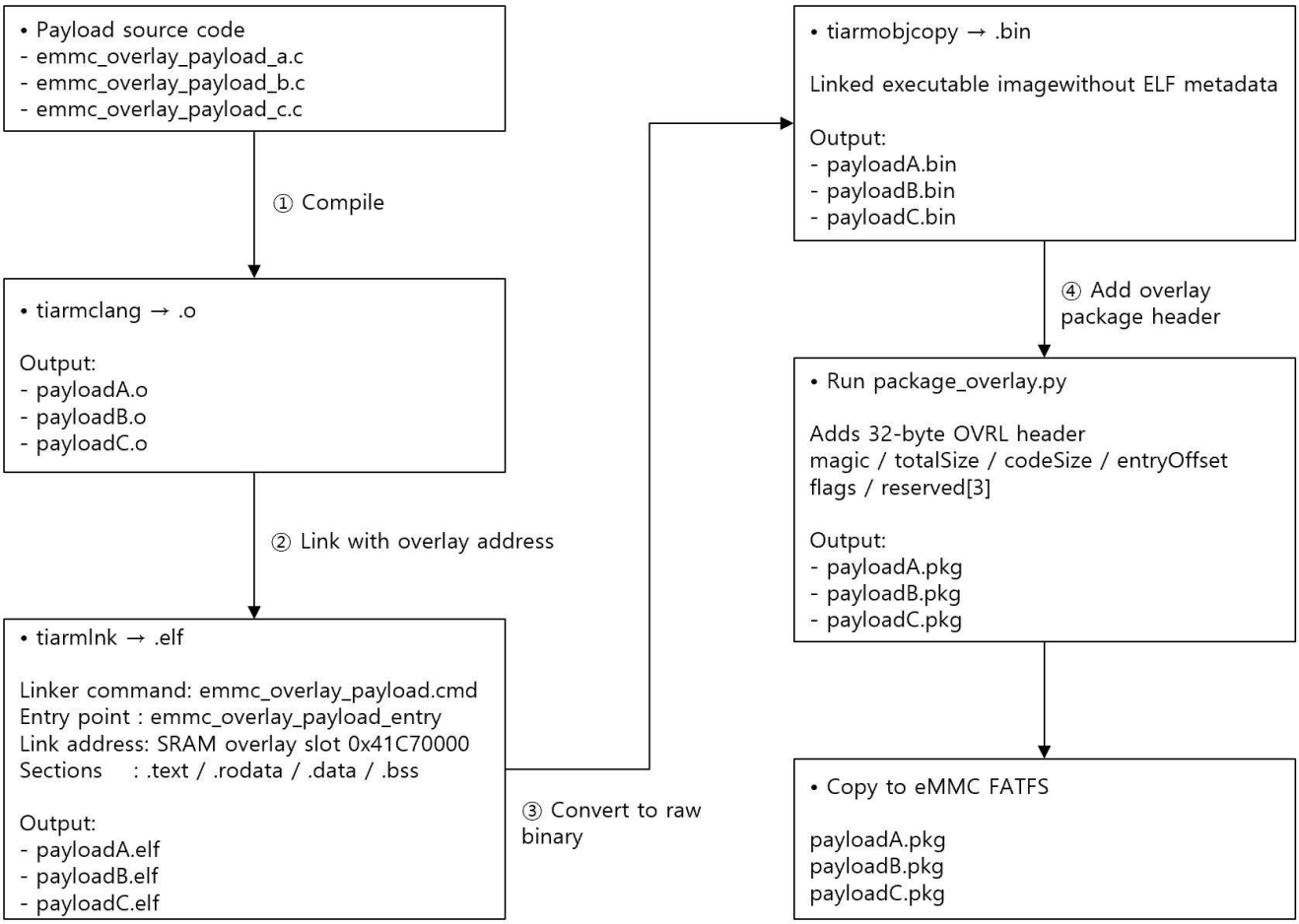

图 5-1. 覆盖有效载荷包生成流程

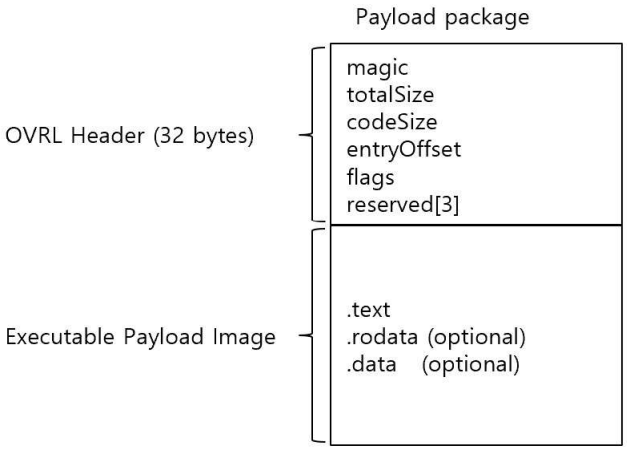

图 5-2. 覆盖包格式

5.4 有效载荷加载和执行

运行时覆盖框架将可执行有效载荷从 eMMC FATFS 加载到共享的 SRAM 覆盖槽中。加载过程由 BootApp_emmcOverlayLoadAndRun() 函数实现。

加载器首先读取并验证覆盖包标头。然后，清除共享 SRAM 覆盖槽，并将可执行有效载荷映像从包文件复制到覆盖执行区域。

有效载荷映像加载到 SRAM 后，由 BootApp_emmcOverlayExecute() 函数执行。执行完成后，有效载荷会将控制权交还给驻留的 boot_app 运行时。然后，相同的 SRAM 覆盖槽可重复用于另一个有效载荷包。

下面是一个与有效载荷加载和执行相关的代码片段。

```
int32_t BootApp_emmcOverlayLoadAndRun(const char *payloadFile,
                                      BootApp_EmmcOverlayCtx *ctx)
{
    BootApp_EmmcOverlayPkgHeader hdr;
    int32_t status;
    void *slotBase = (void *)((uintptr_t)BOOTAPP_EMMC_OVERLAY_SLOT_BASE);

    if ((payloadFile == NULL) || (ctx == NULL))
    {
        return CSL_EBADARGS;
    }

    memset(&hdr, 0, sizeof(hdr));
    status = BootApp_emmcOverlayReadFile(payloadFile, 0U, &hdr, sizeof(hdr));
    if (status != CSL_PASS)
    {
        return status;
    }

    status = BootApp_emmcOverlayValidateHeader(payloadFile, &hdr);
    if (status != CSL_PASS)
    {
        return status;
    }

    memset(slotBase, 0, BOOTAPP_EMMC_OVERLAY_SLOT_SIZE);
    CacheP_wbInv(slotBase, BOOTAPP_EMMC_OVERLAY_SLOT_SIZE);

    status = BootApp_emmcOverlayReadFile(payloadFile,
                                         (uint32_t)sizeof(hdr),
                                         slotBase,
                                         hdr.codeSize);
    if (status != CSL_PASS)
    {
        return status;
    }

    UART_printf("emmc_overlay: loaded %s code=%u entryOffset=%u slot=0x%x\r\n",
                payloadFile,
                hdr.codeSize,
                hdr.entryOffset,
                (uint32_t)((uintptr_t)slotBase));

    status = BootApp_emmcOverlayExecute(slotBase, hdr.codeSize, hdr.entryOffset, ctx);
    return status;
}

static void BootApp_emmcOverlaySyncForExec(void *addr, uint32_t size)
{
    CacheP_wbInv(addr, size);
    CSL_armR5CacheInvalidateAllIcache();
    CSL_armR5Dsb();
    CSL_armR5Isb();
}

int32_t BootApp_emmcOverlayExecute(void *slotBase,
                                   uint32_t codeSize,
                                   uint32_t entryOffset,
                                   BootApp_EmmcOverlayCtx *ctx)
{
    uintptr_t entryAddr;
    BootApp_EmmcOverlayEntry entry;
```

```
    if ((slotBase == NULL) || (ctx == NULL) || (codeSize == 0U) || (entryOffset >= codeSize))
    {
        return CSL_EBADARGS;
    }

    BootApp_emmcOverlaySyncForExec(slotBase, codeSize);

    entryAddr = ((uintptr_t)slotBase) + ((uintptr_t)entryOffset);

    /* R5F executes Thumb code. Force bit[0] for function pointer call. */
    entryAddr |= (uintptr_t)0x1U;
    entry = (BootApp_EmmcOverlayEntry)entryAddr;

    UART_printf("emmc_overlay: execute entry=0x%x codeSize=%u entryOffset=%u\r\n",
                (uint32_t)entryAddr,
                codeSize,
                entryOffset);

    entry(ctx);

    CSL_armR5Dsb();
    CSL_armR5Isb();

    return CSL_PASS;
}
```

5.5 构建配置

可以通过专用的构建选项来启用运行时覆盖演示。

当前实现中使用以下构建命令

```
make -s BOARD=j721s2_evm \
CORE=mcu1_0 \
BUILD_PROFILE=release \
BOOTMODE=mmcsd \
OVERLAY_EMMC_TEST=yes \
OVERLAY_EMMC_UDA=yes \
boot_app_mmcsd
```

这些选项支持基于 eMMC FATFS 的运行时覆盖框架和相关的演示代码。

6 运行时代码覆盖验证

本节通过使用单个 SRAM 覆盖区域从 eMMC FATFS 加载并执行多个有效载荷来验证运行时代码覆盖框架。

验证侧重于以下目标

- 基于 FATFS 的有效载荷加载
- 运行时映像加载
- 基于 SRAM 的有效载荷执行
- 函数指针调用
- 重用共享 SRAM 覆盖槽

所有测试均使用位于地址 0x41C70000 的同一覆盖区域执行。

6.1 PayloadA 执行

第一个验证步骤从 eMMC FATFS 加载 payloadA.pkg，并将可执行代码复制到 SRAM 覆盖槽中。

示例日志

```
emmc_overlay: load/run payloadA file=0:/payloadA.pkg
emmc_overlay: loaded 0:/payloadA.pkg code=64 entryOffset=0 slot=0x41c70000
emmc_overlay: execute entry=0x41c70001
payloadA PASS
```

结果证实该有效载荷已成功完成加载、重定位，并从 SRAM 执行。

6.2 PayloadB 执行

第二个验证步骤使用相同的 SRAM 覆盖槽加载 payloadB.pkg。

示例日志

```
emmc_overlay: load/run payloadB file=0:/payloadB.pkg
emmc_overlay: loaded 0:/payloadB.pkg code=64 entryOffset=0 slot=0x41c70000
emmc_overlay: execute entry=0x41c70001
payloadB PASS
```

结果证实了可以使用相同的覆盖区域加载并执行不同的有效载荷。

6.3 PayloadC 执行

第三个验证步骤使用 payloadC.pkg 重复该过程。

示例日志

```
emmc_overlay: load/run payloadC file=0:/payloadC.pkg
emmc_overlay: loaded 0:/payloadC.pkg code=64 entryOffset=0 slot=0x41c70000
emmc_overlay: execute entry=0x41c70001
payloadC PASS
```

结果证实了第三个独立有效载荷在运行时成功执行。

6.4 共享 SRAM 覆盖槽重用

运行时代码覆盖框架的一个关键目标是由多个有效载荷重用单个 SRAM 执行区域，下面是一个代码片段。

```
int32_t BootApp_emmcOverlayTest(void)
{
    int32_t status;
    uint32_t i;
    static const BootApp_EmmcOverlayTestItem testItems[] =
    {
        { "payloadA", BOOTAPP_EMMC_OVERLAY_PAYLOAD_A_FILE, 0U },
        { "payloadB", BOOTAPP_EMMC_OVERLAY_PAYLOAD_B_FILE, 1U },
        { "payloadC", BOOTAPP_EMMC_OVERLAY_PAYLOAD_C_FILE, 2U },
    };

    UART_printf("emmc_overlay: ================================\r\n");
    UART_printf("emmc_overlay: eMMC/FATFS code overlay demo start\r\n");
    UART_printf("emmc_overlay: SRAM scratch slot base=0x%x size=%u\r\n",
                (uint32_t)BOOTAPP_EMMC_OVERLAY_SLOT_BASE,
                (uint32_t)BOOTAPP_EMMC_OVERLAY_SLOT_SIZE);

    status = BootApp_emmcOverlayStorageOpen();
    if (status != CSL_PASS)
    {
        UART_printf("emmc_overlay: storage open FAIL\r\n");
        UART_printf("emmc_overlay: ================================\r\n");
        return status;
    }

    for (i = 0U; i < (sizeof(testItems) / sizeof(testItems[0])); i++)
    {
        status = BootApp_emmcOverlayRunOne(&testItems[i]);
        if (status != CSL_PASS)
        {
            break;
        }
    }

    BootApp_emmcOverlayStorageClose();

    if (status == CSL_PASS)
    {
        UART_printf("emmc_overlay: eMMC/FATFS code overlay demo PASS\r\n");
    }
    else
    {
        UART_printf("emmc_overlay: eMMC/FATFS code overlay demo FAIL\r\n");
```

```
    }

    UART_printf("emmc_overlay: ================================\r\n");
    return status;
}
```

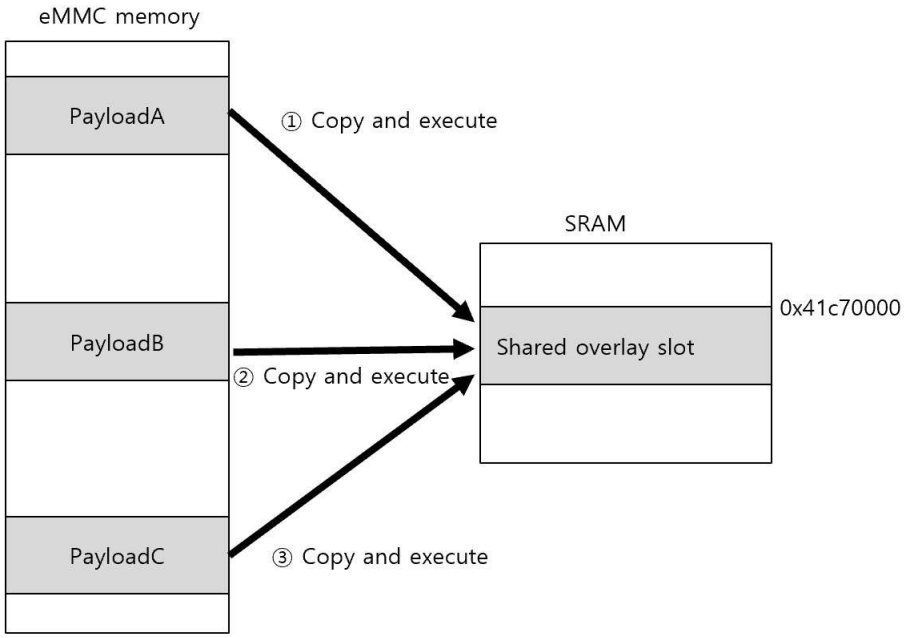

图 6-1. 覆盖槽重用

PayloadA、PayloadB 和 PayloadC 会反复重用同一个 SRAM 覆盖槽。不会为每个有效载荷分配专用执行存储器。

这表明多个软件模块可通过运行时加载和重定位共享一个共用执行区域。

6.5 完整运行时验证

最终日志证实了成功执行完整运行时覆盖序列。

```
SBL Revision: 01.00.10.01 (Sep  4 2025 - 14:31:42)
TIFS  ver: 11.1.8--v11.01.08 (Fancy Rat)
Starting Sciserver..... PASSED
MCU R5F App started at 1370 usecs
emmc_overlay: ================================
emmc_overlay: eMMC/FATFS code overlay demo start
emmc_overlay: SRAM scratch slot base=0x41c70000 size=16384
emmc_overlay: storage target=eMMC UDA volume=0 drvInst=0
emmc_overlay: MMCSD config ready target=eMMC UDA drvInst=0 busWidth=8
emmc_overlay: FATFS_open PASS volume=0 handle=0x41cb06a0
emmc_overlay: load/run payloadA file=0:/payloadA.pkg
emmc_overlay: loaded 0:/payloadA.pkg code=64 entryOffset=0 slot=0x41c70000
emmc_overlay: execute entry=0x41c70001 codeSize=64 entryOffset=0
emmc_overlay: payloadA PASS
emmc_overlay: load/run payloadB file=0:/payloadB.pkg
emmc_overlay: loaded 0:/payloadB.pkg code=64 entryOffset=0 slot=0x41c70000
emmc_overlay: execute entry=0x41c70001 codeSize=64 entryOffset=0
emmc_overlay: payloadB PASS
emmc_overlay: load/run payloadC file=0:/payloadC.pkg
emmc_overlay: loaded 0:/payloadC.pkg code=64 entryOffset=0 slot=0x41c70000
emmc_overlay: execute entry=0x41c70001 codeSize=64 entryOffset=0
emmc_overlay: payloadC PASS
emmc_overlay: FATFS_close done
emmc_overlay: eMMC/FATFS code overlay demo PASS
emmc_overlay: ================================
Loading BootImage
```

结果验证了存储在 eMMC FATFS 中的可执行有效载荷可以动态加载到 SRAM 中并使用共享覆盖槽执行。

7 总结

本应用手册介绍了一种在 TI TDA4x MCU R5F 内核上使用 eMMC FATFS 的运行时代码覆盖方法。

该实现将可执行有效载荷作为文件存储在 eMMC 用户数据区域 (UDA) 中，并在运行时将其动态加载到可重用的 SRAM 执行区域。由于 eMMC 作为块存储器件被访问，不支持直接执行代码，因此有效载荷通过 FATFS 层读取，加载到 SRAM 覆盖区域中，然后通过运行时覆盖机制执行。

演示的框架包含驻留运行时、覆盖加载器和可执行有效载荷包。使用该框架，从 eMMC FATFS 成功加载并执行多个有效载荷，同时共享单个 SRAM 覆盖槽。

验证结果证实了基于 FATFS 的有效载荷访问、运行时映像加载、基于 SRAM 的执行、函数指针调用和对同一执行区域的反复重用。该方法提供了一种在 TI TDA4x MCU R5F 平台上实现基于存储的可执行文件加载和运行时功能扩展的实用方法。

8 参考资料

- 德州仪器 (TI)，TDA4VE，产品页面。
- 德州仪器 (TI)，*J721S2、TDA4AL、TDA4VL、TDA4VE、AM68A 技术参考手册*，技术参考手册。
- 德州仪器 (TI)，*TDA4VE TDA4AL TDA4VL Jacinto™ 处理器，器件版本 1.0*，数据表。
- 德州仪器 (TI)，PROCESSOR-SDK-J721S2，产品页面。

重要通知和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、与某特定用途的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他安全、安保法规或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的相关应用。严禁以其他方式对这些资源进行复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。对于因您对这些资源的使用而对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，您将全额赔偿，TI 对此概不负责。

TI 提供的产品受 TI 销售条款)、TI 通用质量指南 或 ti.com 上其他适用条款或 TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。 除非德州仪器 (TI) 明确将某产品指定为定制产品或客户特定产品，否则其产品均为按确定价格收入目录的标准通用器件。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

最后更新日期：2025 年 10 月