参考資料

Application Note

TI TDA4x マイコン R5F 上の eMMC FATFS を使用したランタイム コード オーバーレイ

Johnny Kim

概要

TI の TDA4x デバイスは、常駐ランタイム環境のメモリ フットプリントを上回るソフトウェア機能を必要とする、幅広い組込みアプリケーションをサポートしています。ランタイム コード オーバーレイは、必要なときにのみ実行可能コードを動的にロードするメカニズムを提供し、複数のソフトウェア モジュールが共通の実行領域を共有できるようにします。

このアプリケーション ノートは、TI TDA4x マイコン R5F コア上で eMMC FATFS を使用しているランタイム コード オーバーレイ手法について説明します。実行可能ペイロードは eMMC User Data Area (UDA) にファイルとして格納され、実行時、再利用可能な SRAM 実行領域に動的にロードされます。メモリ マップト ストレージ デバイスとは異なり、eMMC はブロック ストレージ デバイスとしてアクセスされ、マイコン R5F コアによる直接コード実行をサポートしていません。したがって、実行可能ペイロードは、FATFS レイヤを介して読み取られ、指定された SRAM オーバーレイ領域にロードされ、ランタイム オーバーレイ メカニズムによって実行される必要があります。常駐ランタイムは、ペイロード管理、FATFS アクセス、コード再配置、キャッシュ メンテナンス、関数ポインタの実行を担当します。

単一の SRAM オーバーレイ スロットを使用して eMMC FATFS から複数のペイロードをロードし、実行することにより、この実装が検証されます。ソフトウェア モジュールごとに専用のメモリを割り当てずに、同じ実行領域を異なるペイロードで繰り返し再利用できます。この手法は、TI TDA4x マイコン R5F で、実行可能ロードとランタイム機能を拡張するための実用的なアプローチを示します。

目次

1 概要 3
2 ランタイム コード オーバーレイのバックグラウンド 4
2.1 TDA4x のメモリ アーキテクチャ 4
2.2 静的コード割り当てに関する課題 4
2.3 ランタイム コード オーバーレイを使用する理由 4
3 ランタイム コード オーバーレイ手法 5
3.1 概要 5
3.2 常駐ランタイム 5
3.3 オーバーレイ ペイロード パッケージ 6
3.4 共有 SRAM オーバーレイ領域 6
3.5 ランタイム オーバーレイ シーケンス 6
4 ランタイム コード オーバーレイ アーキテクチャ 7
4.1 ソフトウェア アーキテクチャ 7
4.2 オーバーレイ パッケージ形式 7
4.3 メモリ レイアウト 7
4.4 ランタイム イメージのロード 7
4.5 ランタイム実行 8
5 デモの実装 8
5.1 ソフトウェア構成 8
5.2 オーバーレイ SRAM 構成 8
5.3 ペイロードの生成 9
5.4 ペイロードのロードと実行 12
5.5 ビルド構成 13

6 ランタイム コード オーバーレイ検証 13
6.1 PayloadA の実行 13
6.2 PayloadB の実行 14
6.3 PayloadC の実行 14
6.4 共有 SRAM オーバーレイ スロットの再利用 14
6.5 ランタイム検証を完了する 16
7 まとめ 16
8 参考資料 16

商標

すべての商標は、それぞれの所有者に帰属します。

1 概要

TI の TDA4x デバイス群は、マイコン R5F コア上で動作する幅広い組込みアプリケーションをサポートしています。ソフトウェアがより複雑になるにつれ、常駐のランタイム環境で提供される機能にとどまらず、診断、メンテナンス ユーティリティ、リカバリ サービス、機能固有のモジュールなどの追加機能が必要になる場合があります。

すべての実行可能コードを 1 つのアプリケーション イメージに静的にリンクするのが一般的なアプローチです。実装は簡単ですが、実行時にアクティブに使用されているかどうかにかかわらず、このアプローチはすべてのソフトウェア コンポーネントにメモリを永続的に割り当てます。ソフトウェア機能の数が増えると、アプリケーションのメモリ フットプリントも増加します。

ランタイム コード オーバーレイは、必要なときにのみ実行可能コードをロードすることにより、代替アプローチを提供します。すべてのソフトウェア モジュールをメモリに保存するのではなく、実行可能ペイロードを外部ストレージに格納し、実行時に共有実行領域に動的にロードできます。こうすることで、複数のソフトウェア モジュールで同じメモリ領域を再利用しながら、常駐メモリ フットプリントを減らせます。

このアプリケーション ノートは、TI TDA4x マイコン R5F コア上で eMMC FATFS を使用しているランタイム コード オーバーレイ手法を示します。実行可能なペイロードは eMMC User Data Area (UDA) にファイルとして格納され、実行が要求されると再利用可能な SRAM 実行領域にロードされます。メモリ マップト ストレージ デバイスとは異なり、eMMC はブロック ストレージ デバイスとしてアクセスされ、マイコン R5F コアによる直接コード実行をサポートしていません。したがって、実行可能ペイロードは、FATFS レイヤを介して読み取られ、実行可能メモリにコピーされ、ランタイム オーバーレイ メカニズムを介して実行される必要があります。

このアプリケーション ノートに記載されている実装は、単一の SRAM オーバーレイ スロットにより複数のペイロードのランタイム ロードと実行を検証します。この資料では、テキサス インスツルメンツの TDA4x デバイス群のオーバーレイ アーキテクチャ、ペイロード形式、ソフトウェア実装、ランタイム実行フロー、検証手順について説明します。

2 ランタイム コード オーバーレイのバックグラウンド

2.1 TDA4x のメモリ アーキテクチャ

TI の TDA4x デバイスは、マイコン R5F コアで使用できる複数のメモリ リソースを提供しています。これには、オンチップ SRAM 領域、密結合メモリ (TCM)、OCMC メモリ、外部 DDR メモリが含まれます。

多くのソフトウェア アーキテクチャでは、実行可能コードは外部メモリに格納され、システムの初期化時に DDR にロードされます。このアプローチでは、複雑なソフトウェア アプリケーションと複数のソフトウェア モジュール用に十分なメモリ容量を提供します。

メモリ リソースに加えて、TDA4x デバイスは eMMC、OSPI NOR フラッシュ、SD カードなどのさまざまなストレージ デバイスをサポートしています。これらのストレージ デバイスは、ソフトウェア イメージ、構成データ、およびアプリケーション固有のコンテンツを保存するために使用できます。

2.2 静的コード割り当てに関する課題

共通のソフトウェア配置モデルは、すべての実行可能モジュールを単一のアプリケーション イメージに静的にリンクします。システムの初期化中に、実行可能イメージ全体がメモリにロードされ、実行中は常駐します。

この方法ではソフトウェア管理が簡素化されますが、実際の使用状況に関係なく、すべてのソフトウェア機能にメモリを永続的に割り当てます。ソフトウェア モジュールの数が増加すると、常駐ランタイム環境のメモリ フットプリントも増加します。

継続的に必要とされないソフトウェア コンポーネントの例には、次のようなものがあります

- 診断機能
- リカバリ ユーティリティ
- 工場出荷時のテスト機能
- サービス ツール
- 機能固有のアプリケーション モジュール

このようなコンポーネントをすべて同時に常駐させると、メモリ使用率が低下する可能性があります。

2.3 ランタイム コード オーバーレイを使用する理由

ランタイム コード オーバーレイは、実行が必要な場合にのみ実行コードをメモリにロードできるようにする手法です。

すべてのソフトウェア モジュールをメモリに保存するのではなく、実行可能ペイロードを外部ストレージに格納し、実行時に共有実行領域に動的にロードします。実行が完了すると、同じメモリ領域を別のペイロードで再利用できます。このアプローチは、常駐ランタイム メモリ フットプリントを制御したままソフトウェア機能が拡張され続ける場合に特に役立ちます。メモリ マップ モードで動作する OSPI NOR フラッシュとは異なり、eMMC はブロック ストレージ デバイスとしてアクセスされ、マイコン R5F コアによる直接コード実行をサポートしていません。したがって、eMMC に格納されている実行可能コードは、まず FATFS レイヤを介して読み取られ、実行前に実行可能メモリにコピーする必要があります。このアプリケーション ノートで説明しているランタイム コード オーバーレイ手法では、ペイロード ストレージ メカニズムとして eMMC FATFS を使用し、実行ターゲットとして再使用可能な SRAM 領域を使用します。これにより、常駐ランタイム フットプリントを小さく抑えながら、複数の実行可能モジュールで共通の実行領域を共有できます。

3 ランタイム コード オーバーレイ手法

3.1 概要

このアプリケーション ノートで紹介するランタイムコード オーバーレイ手法により、ソフトウェアは常駐ランタイム ペイロードと過渡オーバーレイ ペイロードの 2 つのカテゴリに分けられます。

常駐ランタイムはシステム実行中にロードされたままで、ペイロード管理、FATFS アクセス、コード再配置、キャッシュ メンテナンス、実行制御を担当します。オーバーレイ ペイロードは eMMC FATFS に実行可能パッケージ ファイルとして格納され、実行が要求された場合にのみロードされます。

ソフトウェア モジュールそれぞれに専用メモリを予約するのではなく、すべてのペイロードが共通の SRAM 実行領域を共有します。実行時、異なるペイロードが同じ SRAM 領域を繰り返し利用できます。

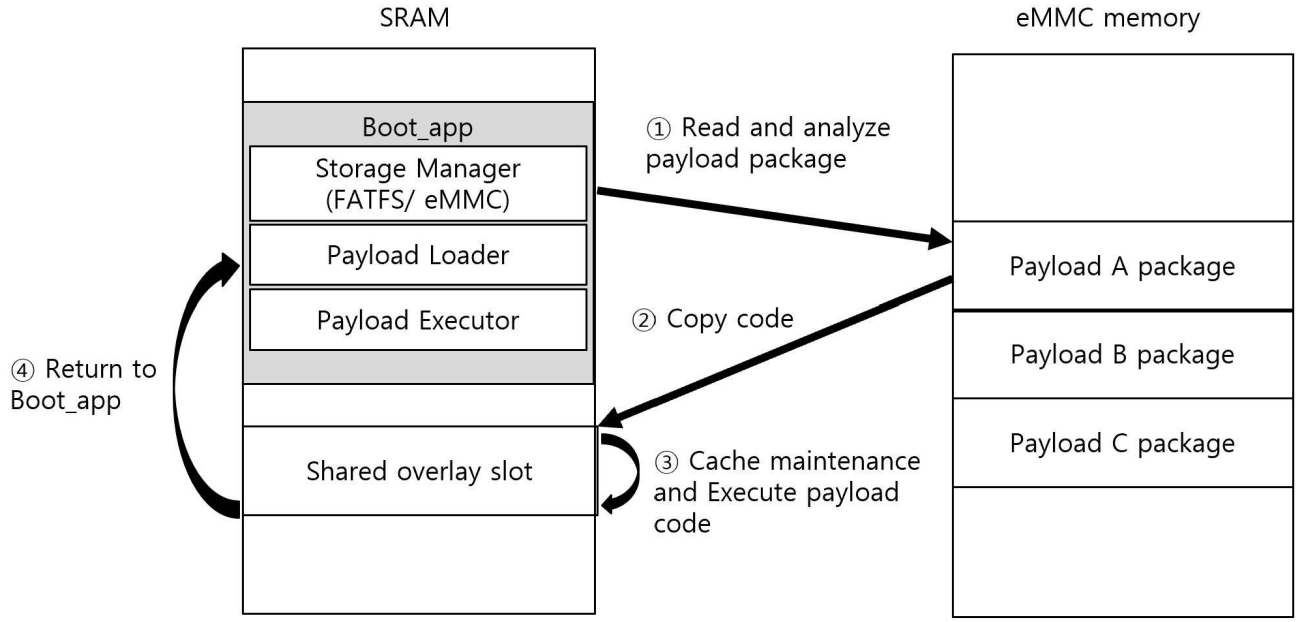

図 3-1. ランタイム コード オーバーレイ手法と制御フロー

3.2 常駐ランタイム

常駐ランタイムは、アプリケーションの有効期間中はアクティブなままです。この実装では、常駐ランタイムが MCU R5F コア上で動作する boot_app アプリケーションに統合されています。

常駐ランタイムは次の関数を実行します

- eMMC の初期化
- FATFS アクセス
- ペイロード ファイル管理
- オーバーレイ パッケージの解析
- ランタイム イメージのロード
- キャッシュ メンテナンス
- 関数ポインタの実行

常駐ランタイムには、個々のオーバーレイ ペイロードの実行可能機能は含まれていません。代わりに、ペイロードを動的にロードして実行するために必要なインフラストラクチャを提供します。

3.3 オーバーレイ ペイロード パッケージ

常駐する必要のないソフトウェア機能は、オーバーレイ ペイロードとしてパッケージされています。

このデモでは、3 つのペイロードを使用します

• payloadA.pkg

• payloadB.pkg

•payloadC.pkg

各ペイロード パッケージには、実行可能コードと、SRAM オーバーレイ領域にロードした後、常駐ランタイムによって呼び出せるエントリ ポイントが含まれています。

ペイロードは eMMC FATFS パーティション内に通常のファイルとして保存され、個別にロードおよび実行できます。

3.4 共有 SRAM オーバーレイ領域

専用の SRAM スクラッチ領域は、オーバーレイ実行用に予約されています。

構成例

• ベース アドレス:0x41C70000

• サイズ:16KB

一度に SRAM オーバーレイ領域を占有するペイロードは 1 つだけです。新しいペイロードが要求されると、以前のペイロードは新しくロードされたペイロードに置き換えられます。これにより、複数の実行可能モジュールが、各モジュールに個別のメモリを割り当てることなく、同じ実行メモリを共有できるようになります。

3.5 ランタイム オーバーレイ シーケンス

ランタイム オーバーレイ プロセスは、次の手順で構成されます。

1) eMMC FATFS からペイロード ファイルを開きます。

2) オーバーレイ パッケージ ヘッダを読み取り、検証します。

3) 実行可能コード セクションを SRAM オーバーレイ領域にコピーします。

4) ロードしたコード イメージのキャッシュ メンテナンスを実行します。

5) ランタイム エントリ アドレスを計算します。

6) 関数ポインタを介してペイロードを実行します。

7) 常駐ランタイムに制御を戻します。

異なるペイロードが要求されるたびに、同じシーケンスが繰り返されます。

4 ランタイム コード オーバーレイ アーキテクチャ

4.1 ソフトウェア アーキテクチャ

ランタイム コード オーバーレイ フレームワークは、4 つの主要なソフトウェア コンポーネントで構成されています。

- 常駐ランタイム
- FATFS レイヤ
- オーバーレイ ローダ
- オーバーレイ ペイロード

常駐ランタイムはペイロード リクエストを管理し、オーバーレイ実行プロセスを調整します。FATFS レイヤは、eMMC UDA に保存されているペイロード イメージへのファイル アクセスを提供します。オーバーレイ ローダは、ペイロード パッケージの解析、実行可能コードの SRAM へのロード、実行環境の準備を行います。オーバーレイ ペイロードには、実行時に動的にロードされる実行可能機能が含まれます。これらのコンポーネントは連携して動作し、共有 SRAM オーバーレイ領域を使用して、ストレージに格納された実行可能ファイルをロードし、実行できます。

4.2 オーバーレイ パッケージ形式

各実行可能ペイロードは、パッケージ ヘッダとそれに続く実行可能コード セクションから構成される、オーバーレイ イメージとしてパッケージされています。パッケージ ヘッダは、ペイロードの検証、実行可能コードのサイズの決定、および実行時エントリ ポイントの検索に必要なランタイム ローダが必要とするメタデータを提供します。現在の実装では、次のパッケージ形式を使用しています。

```
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 メモリ レイアウト

実行可能ペイロードは eMMC FATFS に格納され、実行時に専用の SRAM 実行領域にコピーされます。

オーバーレイ実行スロットは以下にあります

- ベース アドレス:0x41C70000
- サイズ:16KB

オーバーレイ領域に常駐するのは、一度に 1 つのペイロードのみです。この領域は、別のペイロードがロードされるたびに再利用されます。

4.4 ランタイム イメージのロード

eMMC はブロック ストレージ デバイスとしてアクセスされるため、実行可能コードをストレージ メディアから直接実行することはできません。

ペイロードが要求されると、オーバーレイ ローダはペイロード パッケージから実行可能コード セクションを読み取り、それを SRAM オーバーレイ領域にコピーします。再配置の後、ランタイム エントリ アドレスは、パッケージ ヘッダに格納されている SRAM のベース アドレスとエントリ オフセットを使用して計算されます。再配置されたコードは、その後 SRAM から実行されます。

4.5 ランタイム実行

実行可能コードが SRAM に移動された後、命令の一貫性を確認するために、キャッシュ メンテナンスが実行されます。

ランタイム エントリ アドレスは、次のように計算されます

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

その後、ペイロードは関数ポインタを介して実行されます。完了すると、制御は常駐ランタイムに戻り、別のペイロードを同じ SRAM オーバーレイ領域にロードできます。このメカニズムにより、常駐ランタイム フットプリントを小さく抑えながら、複数の実行可能モジュールで 1 つの実行スロットを共有できます。

5 デモの実装

5.1 ソフトウェア構成

ランタイム コードのオーバーレイ デモは、TI プロセッサ SDK RTOS パッケージに含まれている「boot_app」サンプル内に実装されています。

実装は次の部分で構成されます

- 常駐ランタイム統合
- FATFS ベースのペイロード アクセス
- オーバーレイ パッケージ ローダ
- SRAM オーバーレイ マネージャ
- ペイロード実行フレームワーク

オーバーレイ ペイロード生成ユーティリティは個別に提供されており、ランタイム ロード用の実行可能ペイロード パッケージを作成するために使用されます。

5.2 オーバーレイ SRAM 構成

専用の SRAM スクラッチ領域は、ランタイム オーバーレイ実行用に予約されています。

現在のデモは次を使用しています

- オーバーレイ ベース アドレス: 0x41C70000
- オーバーレイ サイズ: 16KB

以下は、リンカ コマンド ファイル「emmc_overlay_payload.cmd」のサンプルです。

```
--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
}
```

オーバーレイ領域は、すべてのペイロードの一時実行スロットとして使用されます。一度にオーバーレイ領域を占有するペイロードは 1 つだけです。新しいペイロードがロードされると、以前のペイロードが上書きされます。

5.3 ペイロードの生成

実行可能ペイロードは、スタンドアロン オーバーレイ パッケージとして生成されます。

以下は 'bin' ファイルからオーバーレイ パッケージを生成するためのサンプル 'package_overlay.py' スクリプトです。

```
#!/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())
```

以下は、payloadA.pkg、payloadB.pkg、payloadC.pkg ペイロードを生成するシェル スクリプト 'build_payloads.sh' のサンプルです。

```
#!/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
```

各パッケージには以下が含まれます

- オーバーレイ パッケージ ヘッダ
- 実行可能コード セクション
- ランタイム エントリ ポイント情報

ペイロード パッケージを生成するには、次のコマンドを使用します

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

生成されたパッケージ ファイルは、その後 eMMC FATFS パーティションにコピーされます。

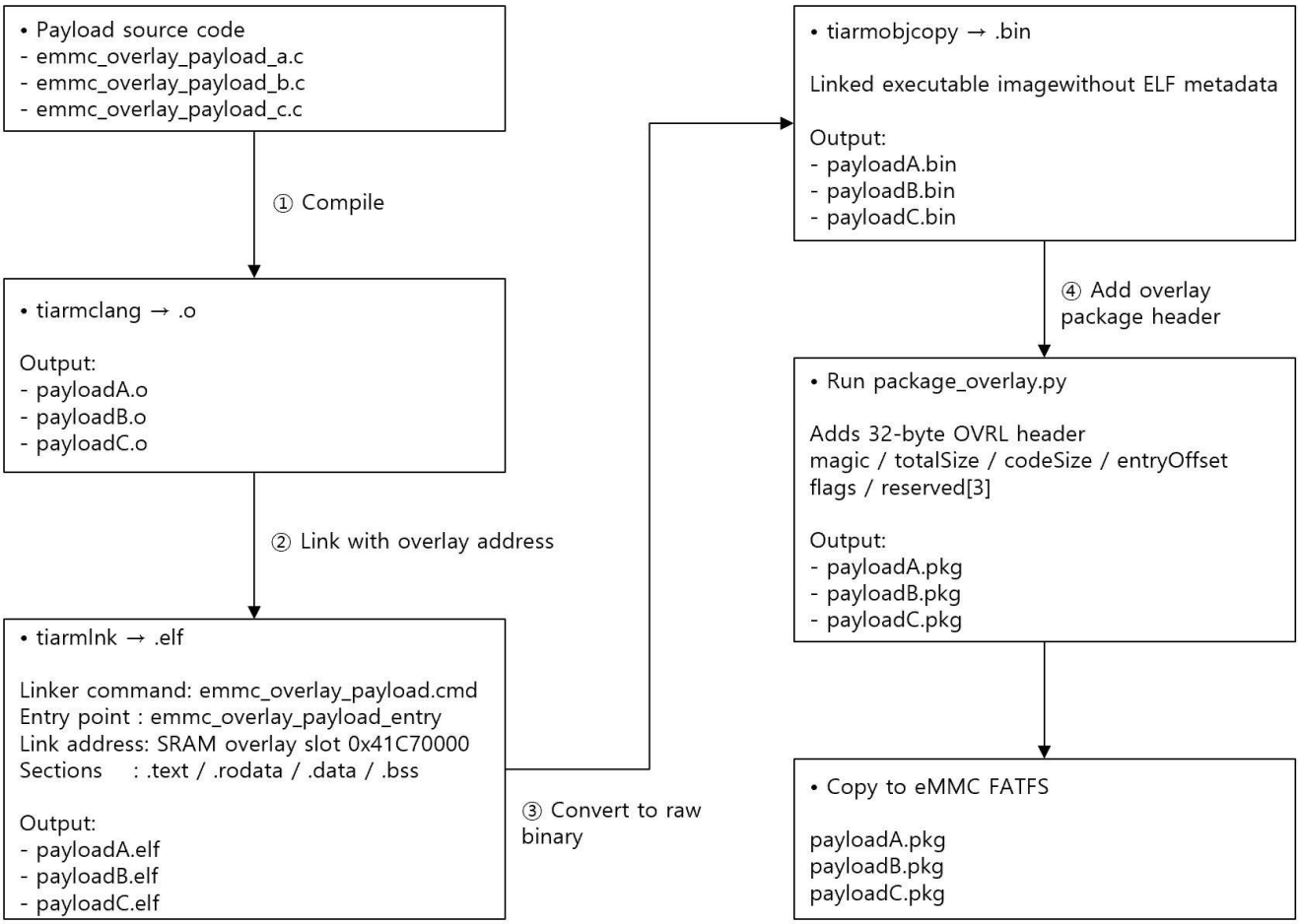

図 5-1. オーバーレイ ペイロード パッケージの生成フロー

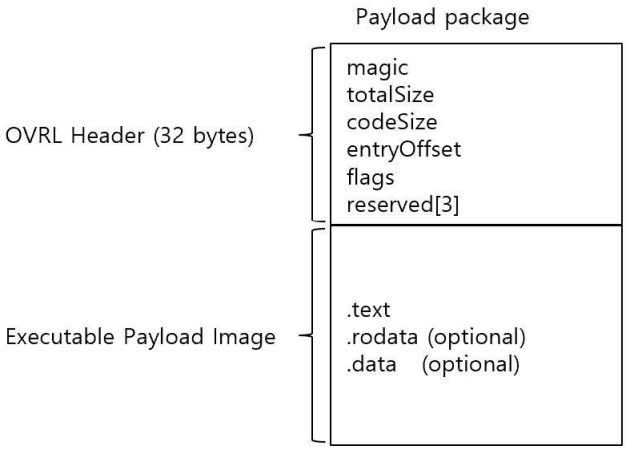

図 5-2. オーバーレイ パッケージ形式

5.4 ペイロードのロードと実行

ランタイム オーバーレイ フレームワークでは、eMMC FATFS から実行可能ペイロードを共有 SRAM オーバーレイ スロットにロードします。ロードのプロセスは BootApp_emmcOverlayLoadAndRun() 関数によって実装されます。

ローダは最初にオーバーレイ パッケージ ヘッダを読み取り、検証します。次に、共有 SRAM オーバーレイ スロットがクリアされ、実行可能ペイロード イメージがパッケージ ファイルからオーバーレイ実行領域にコピーされます。

ペイロード イメージが SRAM にロードされた後、BootApp_emmcOverlayExecute() 関数によって実行されます。実行が完了すると、ペイロードは常駐 boot_app ランタイムに制御を返します。その後、同じ SRAM オーバーレイ スロットを別のペイロード パッケージに再利用できます。

ペイロードのロードと実行に関連するコード スニペットを以下に示します。

```
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 ビルド構成

ランタイム オーバーレイのデモは、専用のビルド オプションを通じてイネーブルになります。

現在の実装では、次のビルド コマンドが使用されています

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

これらのオプションを使用して、eMMC FATFS ベースのランタイム オーバーレイ フレームワークと、関連するデモ コードを有効にします。

6 ランタイム コード オーバーレイ検証

このセクションでは、単一の SRAM オーバーレイ領域を使用して eMMC FATFS から複数のペイロードをロードして実行することにより、ランタイム コード オーバーレイ フレームワークを検証します。

検証の主な目的は以下のとおりです

- FATFS ベース ペイロードのロード
- ランタイム イメージのロード
- SRAM ベースのペイロード実行
- 関数ポインタの呼び出し
- 共有 SRAM オーバーレイ スロットの再利用

すべてのテストは、アドレス 0x41C70000 にある同じオーバーレイ領域を使用して実行されます。

6.1 PayloadA の実行

最初の検証ステップでは、eMMC FATFS から payloadA.pkg をロードし、実行可能コードを SRAM オーバーレイ スロットにコピーします。

ログのサンプル

```
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
```

検証結果により、ペイロードが SRAM から正常にロード、再配置、実行されたことが確認できました。

6.2 PayloadB の実行

2 番目の検証ステップでは、同じ SRAM オーバーレイ スロットを使用して payloadB.pkg をロードします。

ログのサンプル

```
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
```

この結果、同じオーバーレイ領域を使用して異なるペイロードをロードして実行できることが確認されました。

6.3 PayloadC の実行

3 番目の検証ステップでは、payloadC.pkg を使用してプロセスを繰り返します。

ログのサンプル

```
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
```

この結果、3 番目の独立したペイロードの実行が正常に完了したことが確認されます。

6.4 共有 SRAM オーバーレイ スロットの再利用

ランタイム コード オーバーレイ フレームワークの主な目的は、単一の SRAM 実行領域を複数のペイロードで再利用することです。コード スニペットを以下に示します。

```
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;
}
```

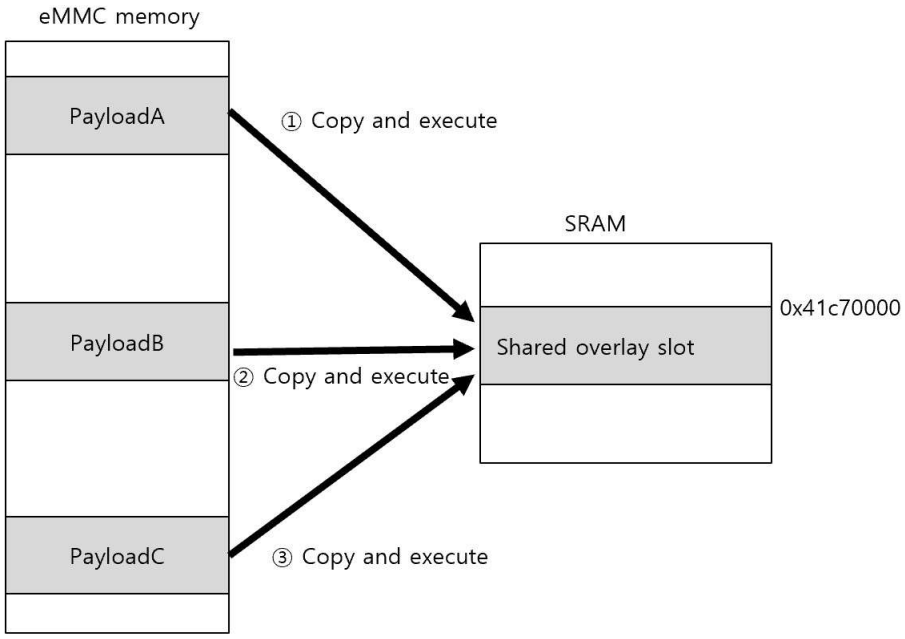

図 6-1. オーバーレイ スロットの再利用

同じ SRAM オーバーレイ スロットが、PayloadA、PayloadB、および PayloadC で繰り返し再利用されます。ペイロードごとに専用の実行メモリが割り当てられることはありません。

これは、複数のソフトウェア モジュールが、ランタイムのロードと再配置を通じて共通の実行領域を共有できることを示しています。

6.5 ランタイム検証を完了する

最終ログで、ランタイム オーバーレイ シーケンス全体の実行が正常に完了したことを確認できます。

```
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
```

その結果、eMMC FATFS に格納されている実行可能ペイロードが SRAM に動的にロードされ、共有オーバーレイ スロットを使用して実行されることが検証されます。

7 まとめ

このアプリケーション ノートは、TI TDA4x マイコン R5F コア上で eMMC FATFS を使用しているランタイム コード オーバーレイ手法について説明します。

この実装では、実行可能なペイロードを eMMC User Data Area (UDA) にファイルとして保存し、実行時に動的に再利用可能な SRAM 実行領域にロードします。eMMC はブロック ストレージ デバイスとしてアクセスされ、直接コード実行をサポートしていないため、ペイロードは FATFS レイヤを介して読み取られ、SRAM オーバーレイ領域にロードされて、ランタイム オーバーレイ メカニズムを使用して実行されます。

提示されたフレームワークは、常駐ランタイム、オーバーレイ ローダ、および実行可能ペイロード パッケージで構成されています。このフレームワークを使用して、1 つの SRAM オーバーレイ スロットを共有しながら、複数のペイロードが eMMC FATFS から正常にロードされ、実行されました。

検証の結果、FATFS ベースのペイロード アクセス、ランタイム イメージのロード、SRAM ベースの実行、関数ポインタの起動、同じ実行領域の繰り返し再利用が確認されました。この手法は、TI TDA4x MCU R5F プラットフォームでストレージに格納された、実行可能ロードとランタイム機能を拡張するための、実用的なアプローチを提供します。

8 参考資料

- テキサス インスツルメンツ、TDA4VE、製品ページ。
- テキサス インスツルメンツ、『J721S2、TDA4AL、TDA4VL、TDA4VE、AM68A テクニカル リファレンス マニュアル』、テクニカル リファレンス マニュアル。
- テキサス インスツルメンツ、『TDA4VE TDA4AL TDA4VL Jacinto™ プロセッサ、シリコン リビジョン 1.0 データシート』、データシート。
- テキサス インスツルメンツ、PROCESSOR-SDK-J721S2、製品ページ。

重要なお知らせと免責事項

TI は、技術データと信頼性データ (データシートを含みます)、設計リソース (リファレンス デザインを含みます)、アプリケーションや設計に関する各種アドバイス、Web ツール、安全性情報、その他のリソースを、欠陥が存在する可能性のある「現状のまま」提供しており、商品性および特定目的に対する適合性の黙示保証、第三者の知的財産権の非侵害保証を含むいかなる保証も、明示的または黙示的にかかわらず拒否します。

これらのリソースは、TI 製品を使用する設計の経験を積んだ開発者への提供を意図したものです。(1) お客様のアプリケーションに適した TI 製品の選定、(2) お客様のアプリケーションの設計、検証、試験、(3) お客様のアプリケーションに該当する各種規格や、その他のあらゆる安全性、セキュリティ、規制、または他の要件への確実な適合に関する責任を、お客様のみが単独で負うものとします。

上記の各種リソースは、予告なく変更される可能性があります。これらのリソースは、リソースで説明されている TI 製品を使用するアプリケーションの開発の目的でのみ、TI はその使用をお客様に許諾します。これらのリソースに関して、他の目的で複製することや掲載することは禁止されています。TI や第三者の知的財産権のライセンスが付与されている訳ではありません。お客様は、これらのリソースを自身で使用した結果発生するあらゆる申し立て、損害、費用、損失、責任について、TI およびその代理人を完全に補償するものとし、TI は一切の責任を拒否します。

TI の製品は、 TI の販売条件 、 TI の総合的な品質ガイドライン 、 ti.com または TI 製品などに関連して提供される他の適用条件に従い提供されます。TI がこれらのリソースを提供することは、適用される TI の保証または他の保証の放棄の拡大や変更を意味するものではありません。 TI がカスタム、またはカスタマー仕様として明示的に指定していない限り、TI の製品は標準的なカタログに掲載される汎用機器です。

お客様がいかなる追加条項または代替条項を提案する場合も、TI はそれらに異議を唱え、拒否します。

最終更新日：2025 年 10 月