Application Note

Runtime Code Overlay Using eMMC FATFS on TI TDA4x MCU R5F

Johnny Kim

ABSTRACT

TI TDA4x devices support a wide range of embedded applications that may require software functionality beyond the memory footprint of a resident runtime environment. Runtime code overlay provides a mechanism to dynamically load executable code only when required, allowing multiple software modules to share a common execution region.

This application note presents a runtime code overlay methodology using eMMC FATFS on a TI TDA4x MCU R5F core. Executable payloads are stored as files in the eMMC User Data Area (UDA) and dynamically loaded into a reusable SRAM execution region at runtime. Unlike memory-mapped storage devices, eMMC is accessed as a block storage device and does not support direct code execution by the MCU R5F core. Therefore, executable payloads must be read through the FATFS layer, loaded into a designated SRAM overlay region, and executed through a runtime overlay mechanism. A resident runtime is responsible for payload management, FATFS access, code relocation, cache maintenance, and function pointer execution.

The implementation is validated by loading and executing multiple payloads from eMMC FATFS using a single SRAM overlay slot. The same execution region is repeatedly reused by different payloads without requiring dedicated memory allocation for each software module. This methodology demonstrates a practical approach for storage-backed executable loading and runtime feature expansion on TI TDA4x MCU R5F.

Table of Contents

Trademarks

All trademarks are the property of their respective owners.

1 Introduction

TI TDA4x devices support a wide range of embedded applications running on the MCU R5F cores. As software complexity increases, additional functions such as diagnostics, maintenance utilities, recovery services, and feature-specific modules may be required beyond the functionality provided by the resident runtime environment.

A common approach is to statically link all executable code into a single application image. While straightforward to implement, this approach permanently allocates memory for every software component regardless of whether it is actively used during runtime. As the number of software features grows, the memory footprint of the application also increases.

Runtime code overlay provides an alternative approach by loading executable code only when required. Instead of keeping all software modules resident in memory, executable payloads can be stored in external storage and dynamically loaded into a shared execution region at runtime. This allows multiple software modules to reuse the same memory region while reducing the resident memory footprint.

This application note demonstrates a runtime code overlay methodology using eMMC FATFS on a TI TDA4x MCU R5F core. Executable payloads are stored as files in the eMMC User Data Area (UDA) and loaded into a reusable SRAM execution region when execution is requested. Unlike memory-mapped storage devices, eMMC is accessed as a block storage device and does not support direct code execution by the MCU R5F core. Therefore, executable payloads must be read through the FATFS layer, copied into executable memory, and executed through a runtime overlay mechanism.

The implementation presented in this application note validates runtime loading and execution of multiple payloads using a single SRAM overlay slot. The document describes the overlay architecture, payload format, software implementation, runtime execution flow, and verification procedure on TI TDA4x devices.

2 Runtime Code Overlay Background

2.1 Memory Architecture of TDA4x

TI TDA4x devices provide multiple memory resources that can be used by the MCU R5F cores. These include on-chip SRAM regions, tightly coupled memories (TCMs), OCMC memory, and external DDR memory.

For many software architectures, executable code is stored in external memory and loaded into DDR during system initialization. This approach provides sufficient memory capacity for complex software applications and multiple software modules.

In addition to memory resources, TDA4x devices support various storage devices such as eMMC, OSPI NOR flash, and SD cards. These storage devices can be used to store software images, configuration data, and application-specific content.

2.2 Challenges of Static Code Allocation

A common software deployment model statically links all executable modules into a single application image. During system initialization, the entire executable image is loaded into memory and remains resident throughout runtime.

While this approach simplifies software management, it permanently allocates memory for every software feature regardless of actual usage. As the number of software modules increases, the memory footprint of the resident runtime environment also increases.

Examples of software components that may not be required continuously include

- Diagnostic functions
- Recovery utilities
- Factory test functions
- Service tools
- Feature-specific application modules

Keeping all such components resident simultaneously may lead to inefficient memory utilization.

2.3 Why Runtime Code Overlay?

Runtime code overlay is a technique that allows executable code to be loaded into memory only when execution is required.

Instead of keeping all software modules resident in memory, executable payloads are stored in external storage and dynamically loaded into a shared execution region at runtime. Once execution completes, the same memory region can be reused by another payload. This approach is particularly useful when software functionality continues to expand while the resident runtime memory footprint must remain controlled. Unlike OSPI NOR flash operating in memory-mapped mode, eMMC is accessed as a block storage device and does not support direct code execution by the MCU R5F core. Therefore, executable code stored in eMMC must first be read through the FATFS layer and copied into executable memory before execution. The runtime code overlay methodology presented in this application note uses eMMC FATFS as the payload storage mechanism and a reusable SRAM region as the execution target. This enables multiple executable modules to share a common execution region while maintaining a small resident runtime footprint.

3 Runtime Code Overlay Methodology

3.1 Overview

The runtime code overlay methodology presented in this application note separates the software into two categories: a resident runtime and transient overlay payloads.

The resident runtime remains loaded throughout system execution and is responsible for payload management, FATFS access, code relocation, cache maintenance, and execution control. Overlay payloads are stored as executable package files in eMMC FATFS and are loaded only when execution is requested.

Instead of reserving dedicated memory for each software module, all payloads share a common SRAM execution region. The same SRAM region can be reused repeatedly by different payloads during runtime.

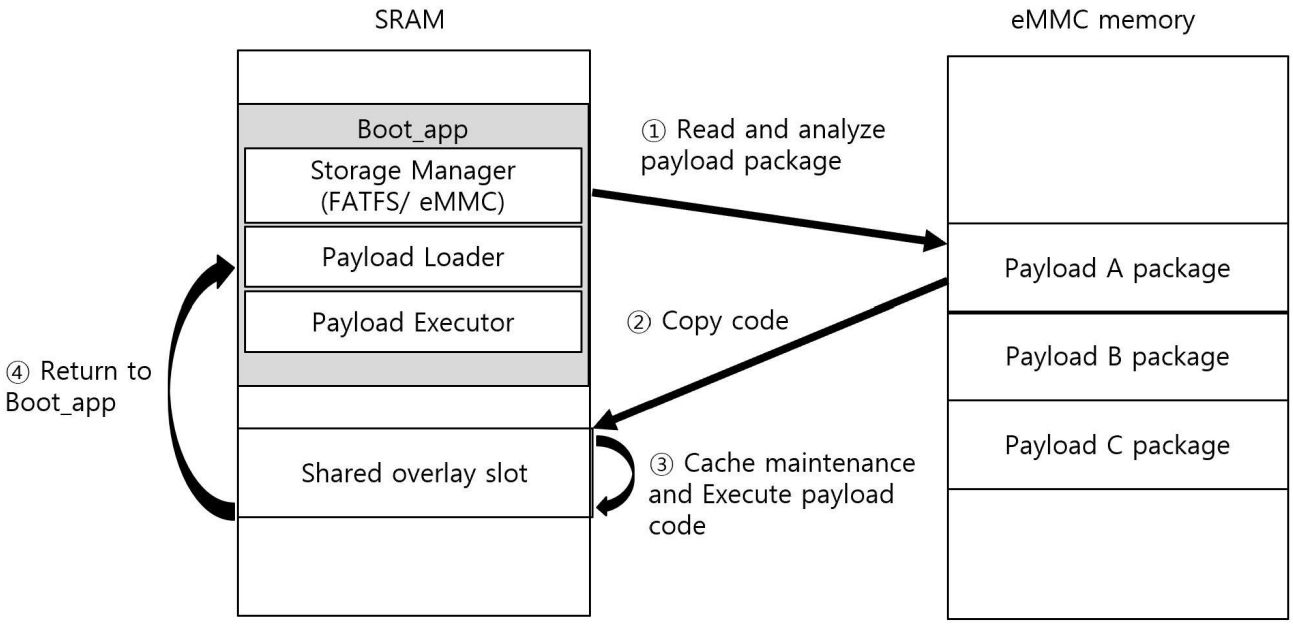

Figure 3-1. Runtime Code Overlay Methodology and Control Flow

3.2 Resident Runtime

The resident runtime remains active throughout the lifetime of the application. In this implementation, the resident runtime is integrated into the boot_app application running on the MCU R5F core.

The resident runtime performs the following functions

- eMMC initialization
- FATFS access
- Payload file management
- Overlay package parsing
- Runtime image loading
- Cache maintenance
- Function pointer execution

The resident runtime does not contain the executable functionality of individual overlay payloads. Instead, it provides the infrastructure required to dynamically load and execute payloads.

3.3 Overlay Payload package

Software functionality that is not required to remain resident is packaged as overlay payloads.

In this demonstration, three payloads are used

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

Each payload package contains executable code and an entry point that can be invoked by the resident runtime after being loaded into the SRAM overlay region.

Payloads are stored as regular files within the eMMC FATFS partition and can be independently loaded and executed.

3.4 Shared SRAM Overlay Region

A dedicated SRAM scratch region is reserved for overlay execution.

Example configuration

- Base Address : 0x41C70000
- Size : 16 KB

Only one payload occupies the SRAM overlay region at any given time. When a new payload is requested, the previous payload is replaced by the newly loaded payload. This allows multiple executable modules to share the same execution memory without requiring separate memory allocation for each module.

3.5 Runtime Overlay Sequence

The runtime overlay process consists of the following steps.

1) Open the payload file from eMMC FATFS.

2) Read and validate the overlay package header.

3) Copy the executable code section into the SRAM overlay region.

4) Perform cache maintenance for the loaded code image.

5) Calculate the runtime entry address.

6) Execute the payload through a function pointer.

7) Return control to the resident runtime.

The same sequence is repeated whenever a different payload is requested.

4 Runtime Code Overlay Architecture

4.1 Software Architecture

The runtime code overlay framework consists of four primary software components.

- Resident Runtime
- FATFS Layer
- Overlay Loader
- Overlay Payload

The resident runtime manages payload requests and coordinates the overlay execution process. The FATFS layer provides file access to payload images stored in eMMC UDA. The overlay loader is responsible for parsing payload packages, loading executable code into SRAM, and preparing the execution environment. Overlay payloads contain the executable functionality that is dynamically loaded at runtime. These components work together to enable storage-backed executable loading and execution using a shared SRAM overlay region.

4.2 Overlay Package Format

Each executable payload is packaged as an overlay image consisting of a package header followed by the executable code section. The package header provides the metadata required by the runtime loader to validate the payload, determine the executable code size, and locate the runtime entry point. The current implementation uses the following package format.

```
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 Memory Layout

Executable payloads are stored in eMMC FATFS and copied into a dedicated SRAM execution region during runtime.

The overlay execution slot is located at

- Base Address : 0x41C70000
- Size : 16 KB

Only one payload is resident in the overlay region at any given time. The region is reused whenever a different payload is loaded.

4.4 Runtime Image Loading

Because eMMC is accessed as a block storage device, executable code cannot be directly executed from the storage medium.

When a payload is requested, the overlay loader reads the executable code section from the payload package and copies it into the SRAM overlay region. After relocation, the runtime entry address is calculated using the SRAM base address and the entry offset stored in the package header. The relocated code is then executed from SRAM.

4.5 Runtime Execution

After the executable code has been relocated into SRAM, cache maintenance is performed to verify instruction coherency.

The runtime entry address is calculated as

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

The payload is then executed through a function pointer. Upon completion, control returns to the resident runtime, allowing another payload to be loaded into the same SRAM overlay region. This mechanism enables multiple executable modules to share a single execution slot while maintaining a small resident runtime footprint.

5 Demo Implementation

5.1 Software Organization

The runtime code overlay demonstration is implemented within the 'boot_app' example included in the TI Processor SDK RTOS package.

The implementation consists of

- Resident runtime integration
- FATFS-based payload access
- Overlay package loader
- SRAM overlay manager
- Payload execution framework

Overlay payload generation utilities are provided separately and are used to create executable payload packages for runtime loading.

5.2 Overlay SRAM Configuration

A dedicated SRAM scratch region is reserved for runtime overlay execution.

The current demonstration uses

- Overlay Base Address : 0x41C70000
- Overlay Size : 16 KB

Below is a sample linker command file '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
}
```

The overlay region is used as a temporary execution slot for all payloads. Only one payload occupies the overlay region at a time. When a new payload is loaded, the previous payload is overwritten.

5.3 Payload Generation

Executable payloads are generated as standalone overlay packages.

Below is a sample 'package_overlay.py' script for generating a overlay package from a 'bin' file.

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

Below is a sample shell script 'build_payloads.sh' to generate payload payloadA.pkg, payloadB.pkg and 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
```

Each package contains

- Overlay package header
- Executable code section
- Runtime entry point information

The following command is used to generate payload packages

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

The generated package files are subsequently copied to the eMMC FATFS partition.

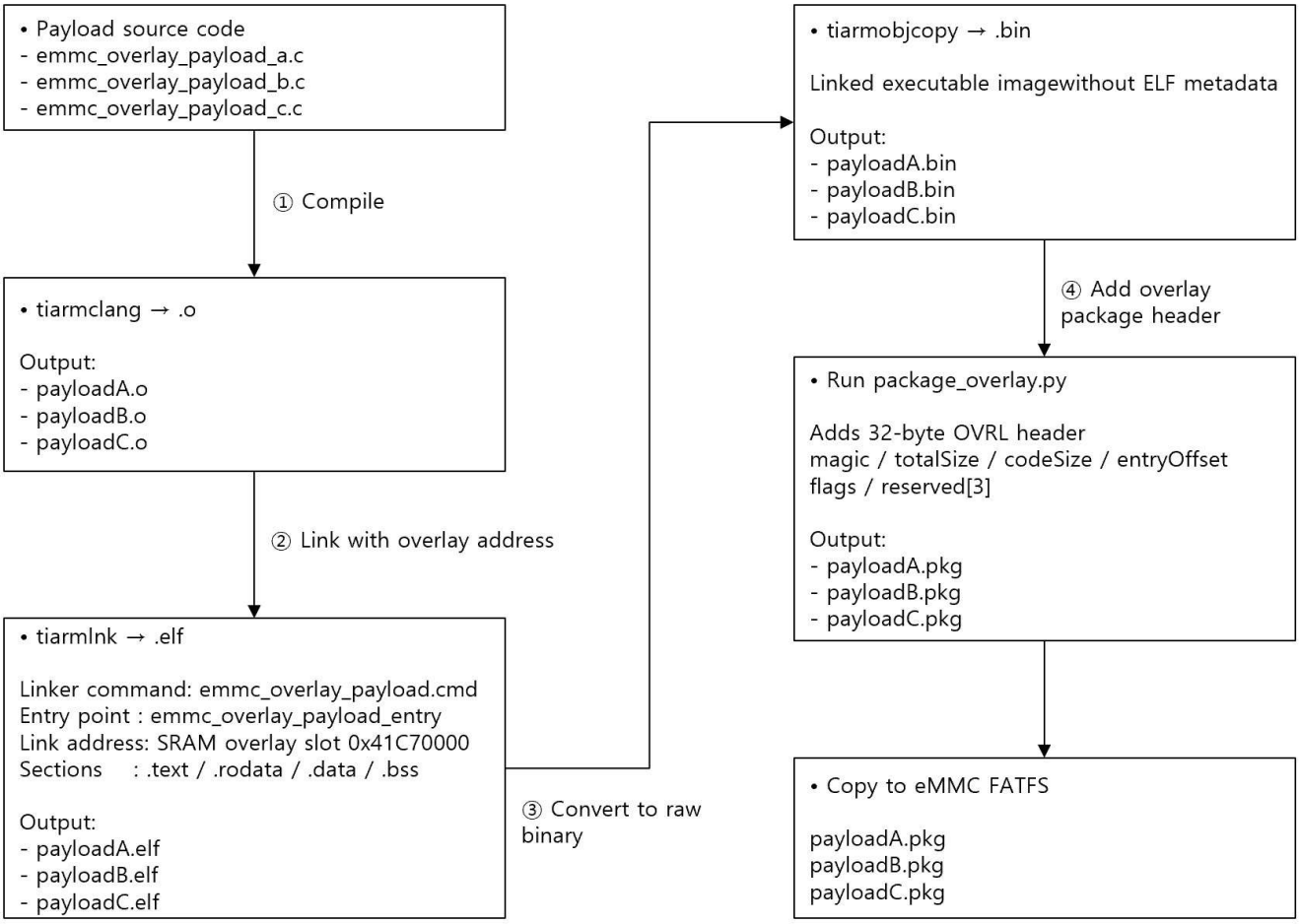

Figure 5-1. Overlay Payload Package Generation Flow

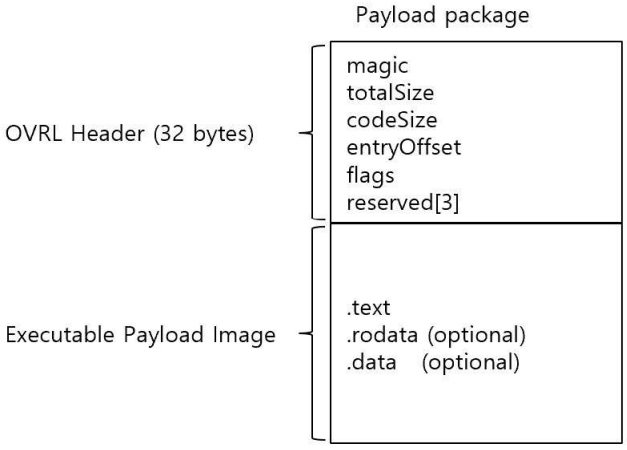

Figure 5-2. Overlay Package Format

5.4 Payload Loading and Execution

The runtime overlay framework loads executable payloads from eMMC FATFS into a shared SRAM overlay slot. The loading process is implemented by the BootApp_emmcOverlayLoadAndRun() function.

The loader first reads and validates the overlay package header. The shared SRAM overlay slot is then cleared and the executable payload image is copied from the package file into the overlay execution region.

After the payload image has been loaded into SRAM, execution is performed by the BootApp_emmcOverlayExecute() function. Once execution is complete, the payload returns control to the resident boot_app runtime. The same SRAM overlay slot can then be reused for another payload package.

Below is a code snippet related to payload loading and execution.

```
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 Build Configuration

The runtime overlay demonstration is enabled through dedicated build options.

The following build command is used in the current implementation

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

These options enable the eMMC FATFS-based runtime overlay framework and associated demonstration code.

6 Runtime Code Overlay Verification

This section verifies the runtime code overlay framework by loading and executing multiple payloads from eMMC FATFS using a single SRAM overlay region.

The verification focuses on the following objectives

- FATFS-based payload loading
- Runtime image loading
- SRAM-based payload execution
- Function pointer invocation
- Reuse of a shared SRAM overlay slot

All tests are executed using the same overlay region located at address 0x41C70000.

6.1 PayloadA Execution

The first verification step loads payloadA.pkg from eMMC FATFS and copies the executable code into the SRAM overlay slot.

Example log

```
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
```

The result confirms that the payload was successfully loaded, relocated, and executed from SRAM.

6.2 PayloadB Execution

The second verification step loads payloadB.pkg using the same SRAM overlay slot.

Example log

```
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
```

The result confirms that a different payload can be loaded and executed using the same overlay region.

6.3 PayloadC Execution

The third verification step repeats the process using payloadC.pkg.

Example log

```
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
```

The result confirms successful runtime execution of a third independent payload.

6.4 Shared SRAM Overlay Slot Reuse

A key objective of the runtime code overlay framework is the reuse of a single SRAM execution region by multiple payloads and below is a code snippet.

```
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;
}
```

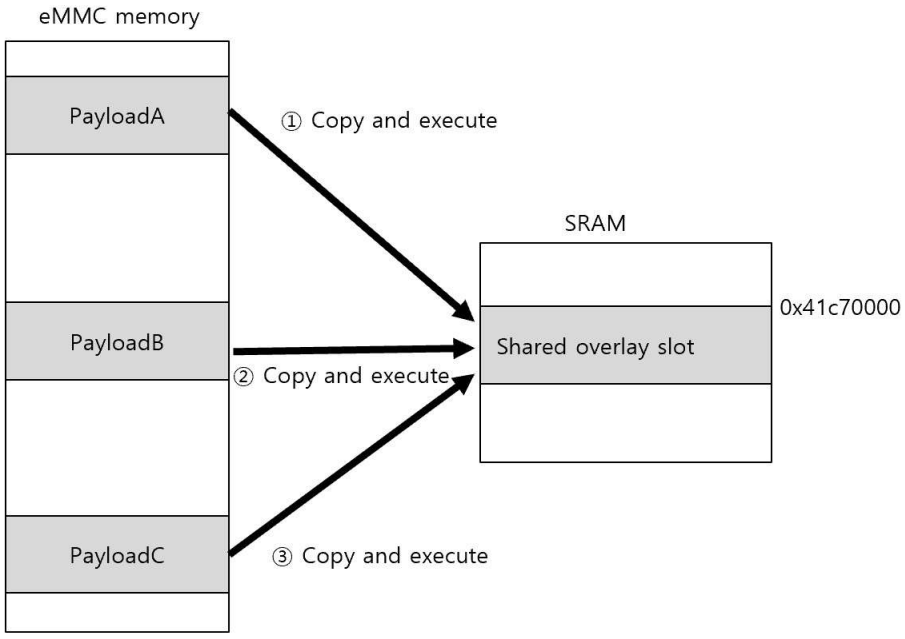

Figure 6-1. Overlay slot reuse

The same SRAM overlay slot is repeatedly reused by PayloadA, PayloadB, and PayloadC. No dedicated execution memory is allocated per payload.

This demonstrates that multiple software modules can share a common execution region through runtime loading and relocation.

6.5 Complete Runtime Verification

The final log confirms successful execution of the complete runtime overlay sequence.

```
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
```

The result verifies that executable payloads stored in eMMC FATFS can be dynamically loaded into SRAM and executed using a shared overlay slot.

7 Summary

This application note presented a runtime code overlay methodology using eMMC FATFS on a TI TDA4x MCU R5F core.

The implementation stores executable payloads as files in the eMMC User Data Area (UDA) and dynamically loads them into a reusable SRAM execution region at runtime. Because eMMC is accessed as a block storage device and does not support direct code execution, payloads are read through the FATFS layer, loaded into the SRAM overlay region, and executed through a runtime overlay mechanism.

The demonstrated framework consists of a resident runtime, an overlay loader, and executable payload packages. Using this framework, multiple payloads were successfully loaded and executed from eMMC FATFS while sharing a single SRAM overlay slot.

Verification results confirmed FATFS-based payload access, runtime image loading, SRAM-based execution, function pointer invocation, and repeated reuse of the same execution region. The methodology provides a practical approach for storage-backed executable loading and runtime feature expansion on TI TDA4x MCU R5F platforms.

8 References

- Texas Instruments, TDA4VE, product page.
- Texas Instruments, *Texas Instruments, J721S2, TDA4AL, TDA4VL, TDA4VE, AM68A Technical Reference Manual*, technical reference manual.
- Texas Instruments, *Texas Instruments, TDA4VE TDA4AL TDA4VL Jacinto™ Processors, Silicon Revision 1.0 datasheet*, datasheet.
- Texas Instruments, PROCESSOR-SDK-J721S2, product page.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you fully indemnify TI and its representatives against any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale, TI’s General Quality Guidelines, or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

TI objects to and rejects any additional or different terms you may propose.

Last updated 10/2025