MA35D1 TF-A FIP Image Backup

Example Code Introduction for 64-bit NuMicro® Family

Document Information

Application	This example code implements the FIP image backup function through BL2 in TF-A, ensuring that the system can still boot successfully even if the primary image becomes corrupted.
BSP Version	Linux-5.10.x
Hardware	NuMaker-HMI-MA35D1 V1.0

The information described in this document is the exclusive intellectual property of Nuvoton Technology Corporation and shall not be reproduced without permission from Nuvoton.

Nuvoton is providing this document only for reference purposes of NuMicro microcontroller and microprocessor based system design. Nuvoton assumes no responsibility for errors or omissions.

All data and specifications are subject to change without notice.

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

1. Overview

This example code implements the backup function of Trusted Firmware-A (TF-A). When BL2 in TF-A detects that a FIP image fails to load or that signature verification during secure boot fails, it will attempt to load the image from the backup region to complete the system boot process.

TF-A is an official boot framework provided by Arm®, designed to implement a secure boot flow for SoCs. In accordance with the Platform Security Architecture (PSA), TF-A consists of the following execution stages:

- BL1 (ROM code): The first bootloader executed after the chip powers on. In the MA35D1, it is implemented in the internal Mask ROM and cannot be modified.
- BL2 (Trusted Bootloader): Loaded and verified by BL1. It is responsible for loading and verifying subsequent images, such as the FIP and SCP.
- BL31 (EL3 Runtime Firmware): Provides secure monitor and context-switching functions.
- BL32: Runs in the Secure World as the trusted execution environment (TEE). The MA35D1 adopts the open-source OP-TEE implementation.
- BL33: The bootloader runs in the Non-secure World. The MA35D1 uses U-Boot for this stage.

The Firmware Image Package (FIP) is a container format used to bundle multiple TF-A images such as BL31, BL32, and BL33 into a single package for BL2 to verify and load at once. Its structure is defined within the fiptool utility.

BL2 plays a critical role in the Arm TF-A boot architecture. It is responsible for the following tasks:

- Load images from the FIP (Image List Search) and verify them using the ECDSA algorithm.
- Ensure the integrity of the Chain of Trust.
- Initialize essential hardware and memory controllers (such as DDR).
- Launch the next-stage images (BL31, BL32, and BL33).

1.1 Principle

The complete boot process of the MA35D1 is illustrated in Figure 1-1. Each time the system powers on, it starts from the Mask ROM. When the Secure Boot feature is enabled, the Mask ROM performs the first-stage verification on BL2.

This verification checks the integrity and authenticity of the BL2 image using the ECDSA P-256 signature attached to it. Once BL2 passes verification, it is loaded into the designated system memory address, after which BL2 takes control and begins the process of loading the subsequent FIP images.

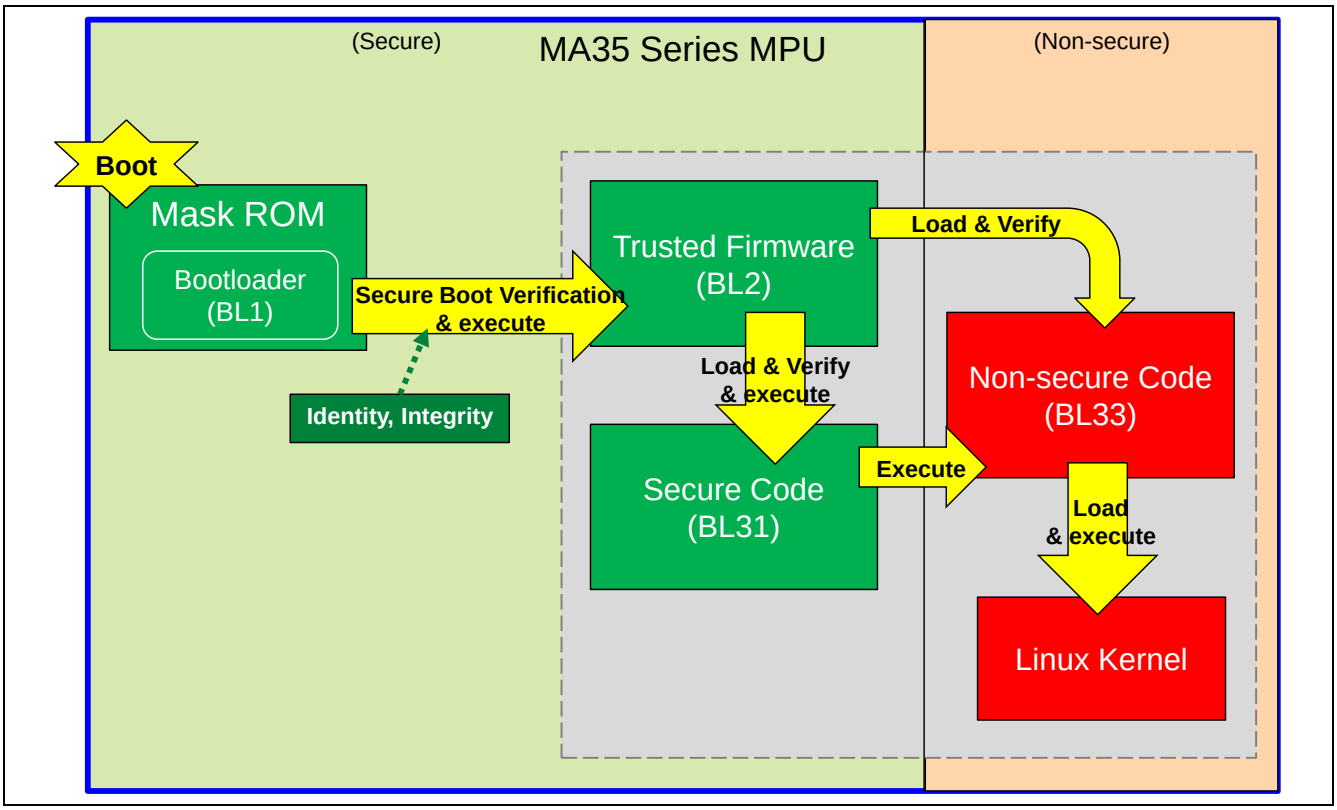


Figure 1-1 MA35D1 Boot Flow

The process of BL2 loading the FIP image can be divided into two stages. In the first stage, BL2 loads the images stored in the external storage device into their designated memory addresses. If the external storage device lacks the required image information, the system determines that the boot has failed. In that case, BL2 switches to a predefined backup address region and retrieves the backup image for loading instead.

Once the FIP images have been successfully loaded into DDR and SRAM, BL2 proceeds to the next stage — performing secure boot verification on the FIP contents. Based on each Image ID, BL2 determines which image (e.g., BL31, BL32, or BL33) needs to be verified and uses the ECDSA P-256 algorithm to confirm its authenticity and integrity.

If the verification fails, an error flag is returned. BL2 will then repeat the first-stage process — loading the backup FIP image from the predefined backup region into DDR and SRAM — and perform secure boot verification again. A schematic representation of this process is shown in Figure 1-2:

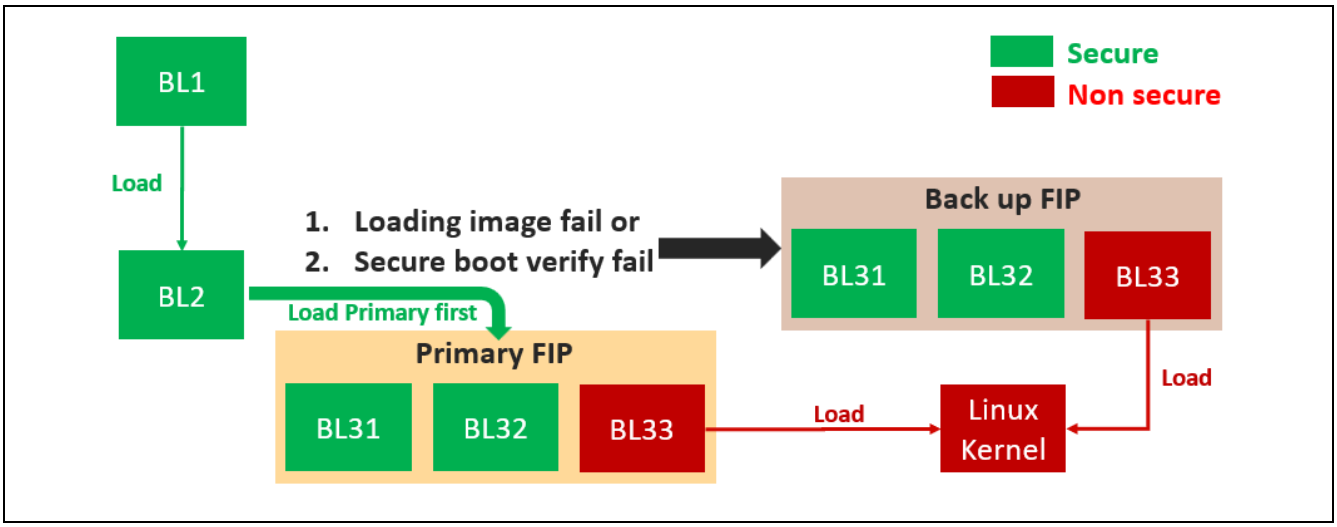


Figure 1-2 FIP Loading Flow

During the build process of the MA35D1, each individual image is packaged into a single combined image by NuWriter, based on the configuration defined in the *pack.json* file corresponding to the selected boot device. This combined image includes the programming information for each individual component. When flashed to external storage via NuWriter, it uses this metadata to write BL2, FIP, Kernel, and the file system to their respective locations. In this example project, the *pack.json* file has been modified so that the backup FIP image is also included within the *pack.image*. When the primary FIP image fails to load, the backup FIP will be loaded into DDR and re-verified again.

Using the SD card as an example of external storage, the pack-sdcard image structure is shown in Figure 1-3:

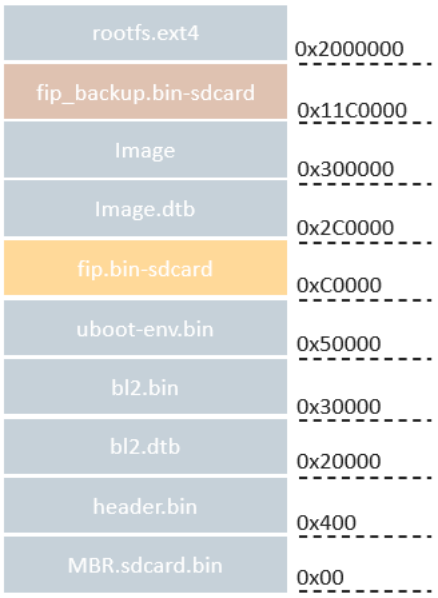


Figure 1-3 Pack-sdcard Structure

1.2 Demo Result

The execution result of this example code is shown in Figure 1-4. Before the first run, all images except the primary FIP are programmed into the external storage device. When the MA35D1 power on and BL2 is successfully loaded, it searches for the FIP image locations based on the FIP information. Since the primary FIP has not been programmed into the storage device, BL2 displays a “load failed” message and then attempts to read the image from the backup FIP location.

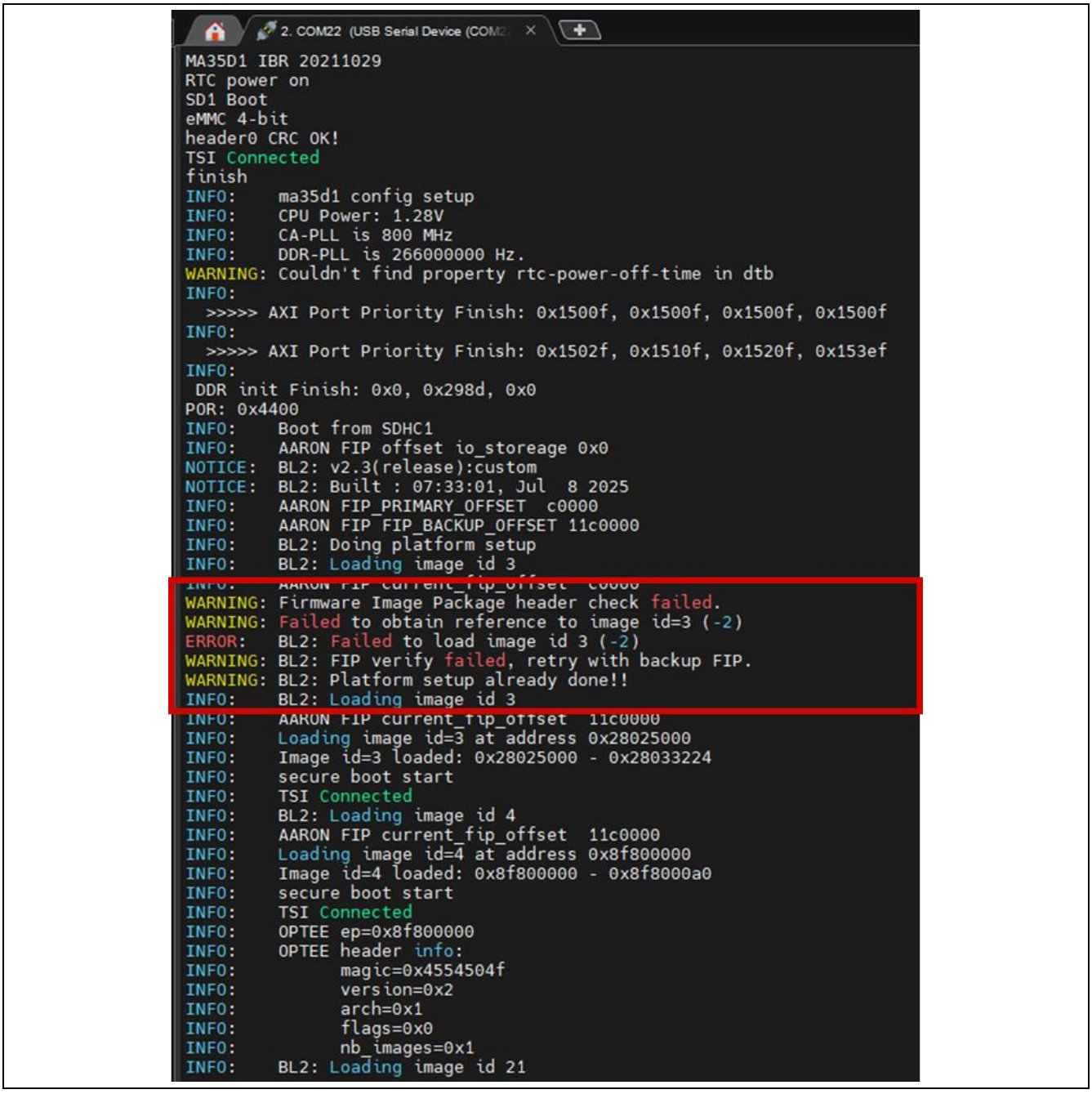


Figure 1-4 Executed Results – Primary Image Loading Failed

The second test execution demonstrates the behavior when the signature verification fails.

Before running the test, a FIP image with an invalid signature is programmed into the primary FIP location. When BL2 successfully loads the FIP image from the primary region, it begins the signature verification. Since the primary FIP contains an incorrect signature, the verification process fails, prompting BL2 to read from the backup FIP location and perform the verification process again.

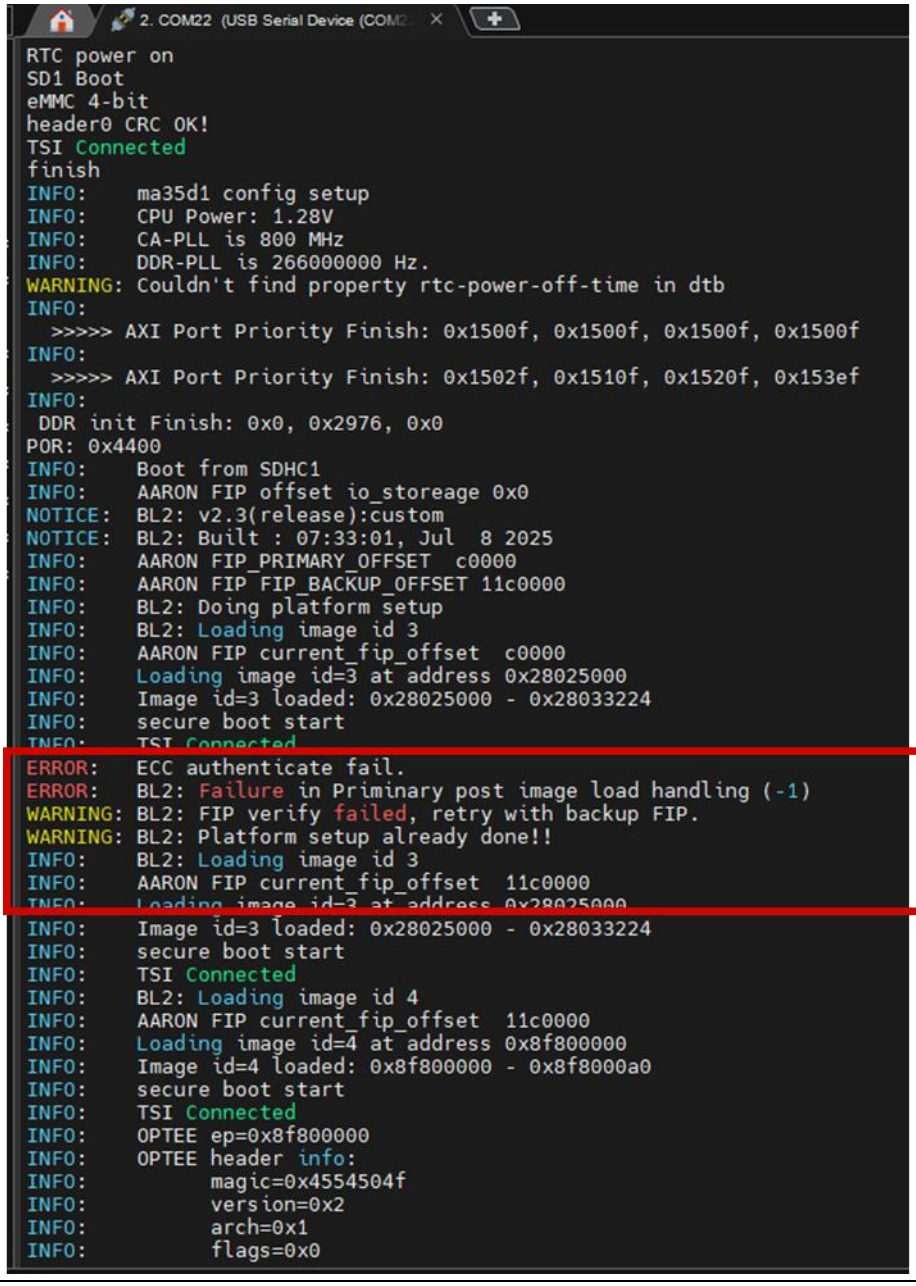


Figure 1-5 Executed Results – Primary Image Verification Failed

2. Code Description

This example primarily implements the backup mechanism by modifying BL2 in TF-A. During execution, BL2 uses bl2_load_images() to load and verify the FIP images. Before using this example code, please first confirm that the system can successfully run the MA35D1 secure boot function.

For clarity, the example is divided into two parts. Part 1:

```
int plat_get_image_source(unsigned int image_id, uintptr_t *dev_handle, uintptr_t
*image_spec)
```

By calling this function, you can obtain the following information:

- **dev_handle**: It represents the access interface of the storage device, such as SD card, NAND flash, or SPI NAND Flash.
- **image_spec**: It specifies the location information of the image on the storage device, typically including the offset, size and format.

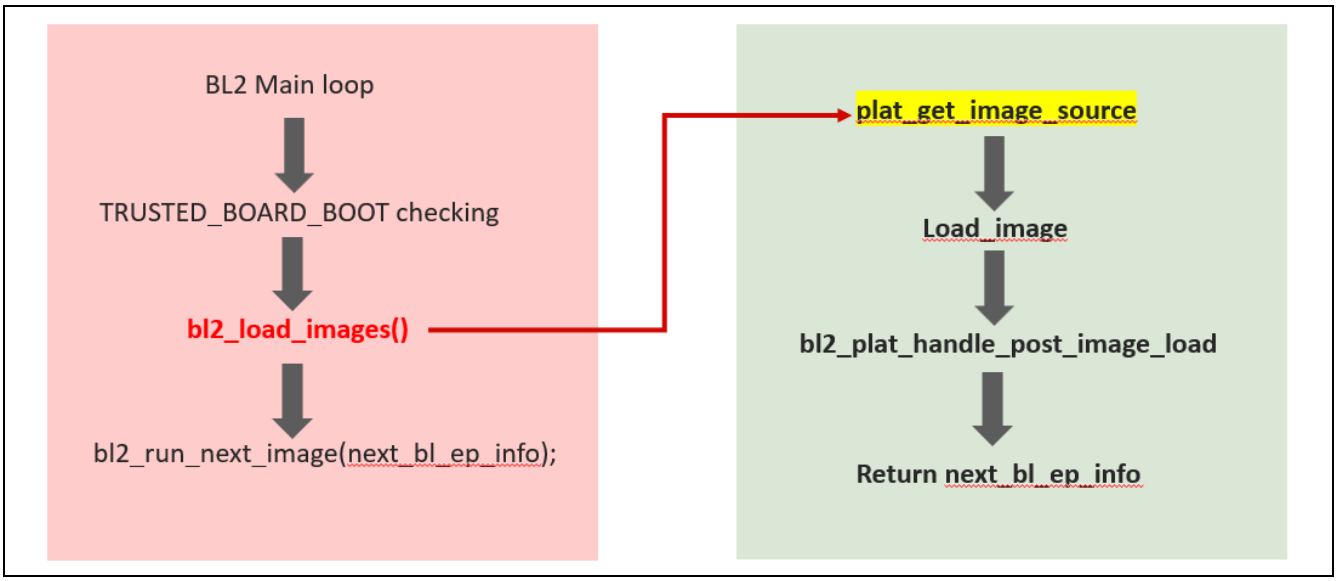


Figure 2-1 BL2 Operation Flow – Loading Process

In the second stage, after obtaining the dev_handle and image_spec, the Load_image function will call:

- io_open(dev_handle, image_spec, &image_handle): A pointer to a function to establish a stream.
- io_read(image_handle, image_base, image_size, &bytes_read): Read the contents from the corresponding offset into DDR.

Next, the process enters **bl2_plat_handle_post_image_load**, where the MA35D1 implements the second stage of secure boot.

In this stage, the system uses the ECDSA P-256 algorithm to verify the authenticity and integrity of the BL31, BL32, and BL33 images.

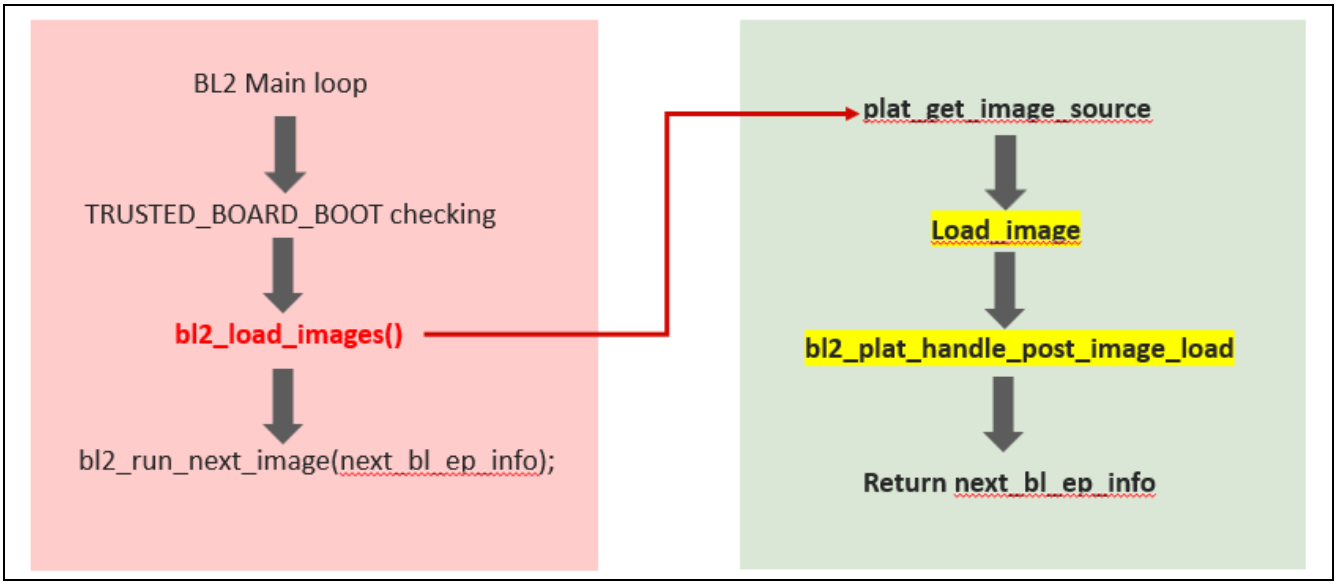


Figure 2-2 BL2 Operation Flow – Verification Process

In the main structure of BL2, within the function bl2_load_images, if either the image loading process (load_auth_image) or the image authentication process (bl2_plat_handle_post_image_load) fails, the FIP image load address will be redirected to the backup region, and the loading operation will be retried. If the backup image also fails to boot, the MA35D1 will print corresponding error messages to indicate the failure.

```
/*
 * Copyright (c) 2016-2020, ARM Limited and Contributors. All rights reserved.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 */

#include <assert.h>
#include <stdint.h>

#include <platform_def.h>

#include <arch.h>
#include <arch_helpers.h>
#include <common/bl_common.h>
#include <common/debug.h>
#include <common/desc_image_load.h>
#include <drivers/auth/auth_mod.h>
#include <plat/common/platform.h>

#include "bl2_private.h"

/*******************************************************************************
 * This function loads SCP_BL2/BL3x images and returns the ep_info for
 * the next executable image.
 *******************************************************************************/
```

```
 extern uint32_t FIP_PRIMARY_OFFSET;
 extern uint32_t FIP_BACKUP_OFFSET;
 extern uint32_t FIP_PRIMARY_OFFSET_SPI;
 extern uint32_t FIP_BACKUP_OFFSET_SPI;

 uint32_t current_fip_offset = 0x00;


struct entry_point_info *bl2_load_images(void)
{
    bl_params_t *bl2_to_next_bl_params;
    bl_load_info_t *bl2_load_info;
    const bl_load_info_node_t *bl2_node_info;
    int plat_setup_done = 0;
    int err;
    bool retried = false;

    INFO(" FIP_PRIMARY_OFFSET  %x\n", FIP_PRIMARY_OFFSET);
    INFO(" FIP FIP_BACKUP_OFFSET %x\n", FIP_BACKUP_OFFSET);
    /*
     * Current FIP offset setting.
     */
    current_fip_offset = FIP_PRIMARY_OFFSET;
    /*
     * Get information about the images to load.
     */
    bl2_load_info = plat_get_bl_image_load_info();
    assert(bl2_load_info != NULL);
    assert(bl2_load_info->head != NULL);
    assert(bl2_load_info->h.type == PARAM_BL_LOAD_INFO);
    assert(bl2_load_info->h.version >= VERSION_2);

    retry_fip:
    bl2_node_info = bl2_load_info->head;

    while (bl2_node_info != NULL) {
        /*
         * Perform platform setup before loading the image,
         * if indicated in the image attributes AND if NOT
         * already done before.
         */
        if ((bl2_node_info->image_info->h.attr &
            IMAGE_ATTRIB_PLAT_SETUP) != 0U) {
            if (plat_setup_done != 0) {
                WARN("BL2: Platform setup already done!!\n");
            } else {
                INFO("BL2: Doing platform setup\n");
                bl2_platform_setup();
                plat_setup_done = 1;
            }
        }

        err = bl2_plat_handle_pre_image_load(bl2_node_info->image_id);
        if (err != 0) {
            ERROR("BL2: Failure in pre image load handling (%i)\n", err);
```

```
            plat_error_handler(err);
        }

        if ((bl2_node_info->image_info->h.attr &
            IMAGE_ATTRIB_SKIP_LOADING) == 0U) {
            INFO("BL2: Loading image id %d\n", bl2_node_info->image_id);
            err = load_auth_image(bl2_node_info->image_id,
                bl2_node_info->image_info);
            if (err != 0) {
                ERROR("BL2: Failed to load image id %d (%i)\n",
                      bl2_node_info->image_id, err);
            goto try_backup;
            }
        } else {
            INFO("BL2: Skip loading image id %d\n", bl2_node_info->image_id);
        }

        /* Allow platform to handle image information.
         * Perform ECDSA verification and AES Image Decryption
         */
        err = bl2_plat_handle_post_image_load(bl2_node_info->image_id);
        if (err != 0) {
            ERROR("BL2: Failure in Primary post image load handling (%i)\n", err);
            goto try_backup;
        }

        /* Go to next image */
        bl2_node_info = bl2_node_info->next_load_info;
    }

    goto success;

    /*
     * Backup Image offset configuration.
     */
    try_backup:
    if (!retried){
        WARN ("BL2: Loading Primary failed, retry with backup FIP.\n");
        current_fip_offset = FIP_BACKUP_OFFSET ;
        retried = true;
        goto retry_fip;

    }else{
        ERROR("BL2: Primary and backup FIP both failed\n");
        plat_error_handler(err);
    }



    success:
    /*
     * Get information to pass to the next image.
     */
    bl2_to_next_bl_params = plat_get_next_bl_params();
    assert(bl2_to_next_bl_params != NULL);
    assert(bl2_to_next_bl_params->head != NULL);
```

```
    assert(bl2_to_next_bl_params->h.type == PARAM_BL_PARAMS);
    assert(bl2_to_next_bl_params->h.version >= VERSION_2);
    assert(bl2_to_next_bl_params->head->ep_info != NULL);

    /* Populate arg0 for the next BL image if not already provided */
    if (bl2_to_next_bl_params->head->ep_info->args.arg0 == (u_register_t)0)
        bl2_to_next_bl_params->head->ep_info->args.arg0 =
                    (u_register_t)bl2_to_next_bl_params;

    /* Flush the parameters to be passed to next image */
    plat_flush_next_bl_params();

    return bl2_to_next_bl_params->head->ep_info;
}
```

In the first stage, within the function **plat_get_image_source**, the system checks whether the image being processed is a FIP image. If it is, the ** current_fip_offset ** determines the read position. By default, it points to the primary FIP storage location. If either the image loading or verification fails, the pointer is redirected to the backup FIP address, and the process is retried.

```
/*
 * Return an IO device handle and specification which can be used to access
 * an image. Use this to enforce platform load policy.
 */
extern  uint32_t current_fip_offset;
int plat_get_image_source(unsigned int image_id, uintptr_t *dev_handle,
              uintptr_t *image_spec)
{
    uintptr_t init_params;

    assert(image_id < ARRAY_SIZE(policies));

    *dev_handle = *policies[image_id].dev_handle;

    /*
     * If image id is FIP, using “current_fip_offset”.
     * Others, using platform load policy.
     */
    if (image_id == FIP_IMAGE_ID){
    INFO(" FIP current_fip_offset  %x\n", current_fip_offset);
        static io_block_spec_t local_fip_spec;
        local_fip_spec.offset = current_fip_offset;
        local_fip_spec.length = MA35D1_FIP_SIZE;
        *image_spec = (uintptr_t)&local_fip_spec;
        init_params = policies[image_id].init_params;


    }else{
        *image_spec = policies[image_id].image_spec;
        init_params = policies[image_id].init_params;
    }
    return io_dev_init(*dev_handle, init_params);
}
```

Based on the configured external storage interface, the system reads the storage location specified in the device tree file (*ma35d1.dtsi*) during the initialization phase.

```
static void ma35d1_sdhc_setup(struct mmc *mmc)
{
    int node;

    /* get device tree information */
    if (fdt_check_header(fdt) < 0) {
        WARN("device tree header check error.\n");
    }

    if (mmc->base == SDH0_BASE) {
        node = fdt_node_offset_by_compatible(fdt, -1,
                "snps,dwcmshc-sdhci0");
        if (node < 0) {
            WARN("The compatible property `snps,dwcmshc-sdhci0` not found\n");
        }
    } else {
        node = fdt_node_offset_by_compatible(fdt, -1,
                "snps,dwcmshc-sdhci1");
        if (node < 0) {
            WARN("The compatible property `snps,dwcmshc-sdhci1` not found\n");
        }
    }

    FIP_PRIMARY_OFFSET= fdt_read_uint32_default(fdt, node, "mmc-image-offset", 0);
    FIP_BACKUP_OFFSET= fdt_read_uint32_default(fdt, node, "mmc-image-backup-offset", 0);

    mmc->bus_width = fdt_read_uint32_default(fdt, node, "bus-width", 1);

    ma35d1_sdhc_hw_init(mmc);
}
```

ma35d1.dtsi:

```
    sdhci0: sdhci@40180000 {
     compatible = "snps,dwcmshc-sdhci0";
     bus-width = <4>;
     /* sdhc information */
     mmc-image-offset = <0xc0000>;
     mmc-image-backup-offset = <0x11c0000>;
    };

    sdhci1: sdhci@40190000 {
     compatible = "snps,dwcmshc-sdhci1";
     bus-width = <4>;
     /* sdhc information */
     mmc-image-offset = <0xc0000>;
     mmc-image-backup-offset = <0x11c0000>;
    };
```

The second part, also known as the second-stage secure boot verification, involves calling **ma35d1_fip_verify**, which uses the TSI to perform ECDSA verification. Once the verification is successful, **ma35d1_fip_deaes** is invoked to decrypt and restore the encrypted image using AES decryption.

```
/*******************************************************************************
 * This function can be used by the platforms to update/use image
 * information for given `image_id`.
 ******************************************************************************/
int bl2_plat_handle_post_image_load(unsigned int image_id)
{
    int err = 0;
    bl_mem_params_node_t *bl_mem_params = get_bl_mem_params_node(image_id);

    bl_mem_params_node_t *pager_mem_params;
    bl_mem_params_node_t *paged_mem_params = NULL;

    assert(bl_mem_params != NULL);
#if FIP_DE_AES
    INFO("secure boot start\n");
    if(ma35d1_fip_verify(bl_mem_params->image_info.image_base,bl_mem_params-
>image_info.image_size)!=0) {
        ERROR("ECC authenticate fail.\n");
        return -1;
        //while(1);
    }
    ma35d1_fip_deaes(bl_mem_params->image_info.image_base,bl_mem_params-
>image_info.image_size);
#endif
    switch (image_id) {
    case BL32_IMAGE_ID:
        pager_mem_params = get_bl_mem_params_node(BL32_EXTRA1_IMAGE_ID);
        assert(pager_mem_params);

        paged_mem_params = get_bl_mem_params_node(BL32_EXTRA2_IMAGE_ID);
        assert(paged_mem_params);

        err = parse_optee_header(&bl_mem_params->ep_info,
                &pager_mem_params->image_info,
                &paged_mem_params->image_info);
        if (err != 0) {
            WARN("OPTEE header parse error.\n");
        }

        /*
         * When ATF loads the DTB the address of the DTB is passed in
         * arg2, if an hw config image is present use the base address
         * as DTB address an pass it as arg2
         */
        bl_mem_params->ep_info.args.arg0 = bl_mem_params->ep_info.args.arg1;
        bl_mem_params->ep_info.args.arg1 = 0;
        bl_mem_params->ep_info.args.arg2 = 0;
        bl_mem_params->ep_info.args.arg3 = 0;

        bl_mem_params->ep_info.spsr = ma35d1_get_spsr_for_bl32_entry();
```

```
        break;

    case BL33_IMAGE_ID:
        /* BL33 expects to receive the primary CPU MPID (through r0) */
        bl_mem_params->ep_info.args.arg0 = 0xffff & read_mpidr();
        bl_mem_params->ep_info.spsr = ma35d1_get_spsr_for_bl33_entry();
        break;

    case SCP_BL2_IMAGE_ID:
        /* The subsequent handling of SCP_BL2 is platform specific */
        err = plat_ma35d1_bl2_handle_scp_bl2(&bl_mem_params->image_info);
        if (err) {
            WARN("Failure in platform-specific handling of SCP_BL2 image.\n");
        }
        break;

    default:
        /* Do nothing in default case */
        break;
    }

    return err;
}
```

3. Software and Hardware Requirements

3.1 Software Requirements

- BSP version
  - Linux-5.10.x

3.2 Hardware Software

- Circuit components
  - NuMaker-HMI-MA35D1 V1.0

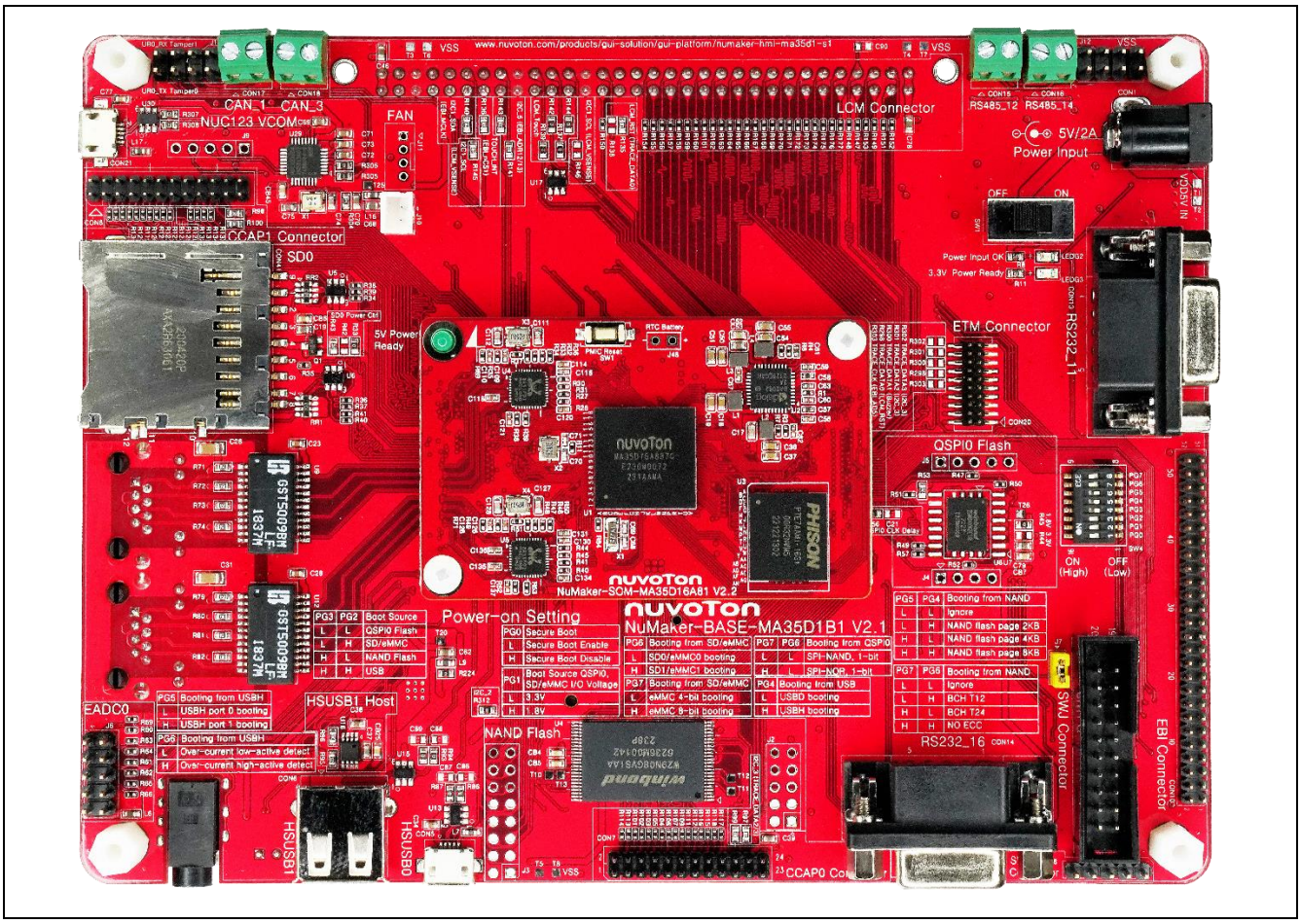


Figure 3-1 NuMaker-HMI-MA35D1 V1.0

4. Directory Information

The directory structure is shown below.

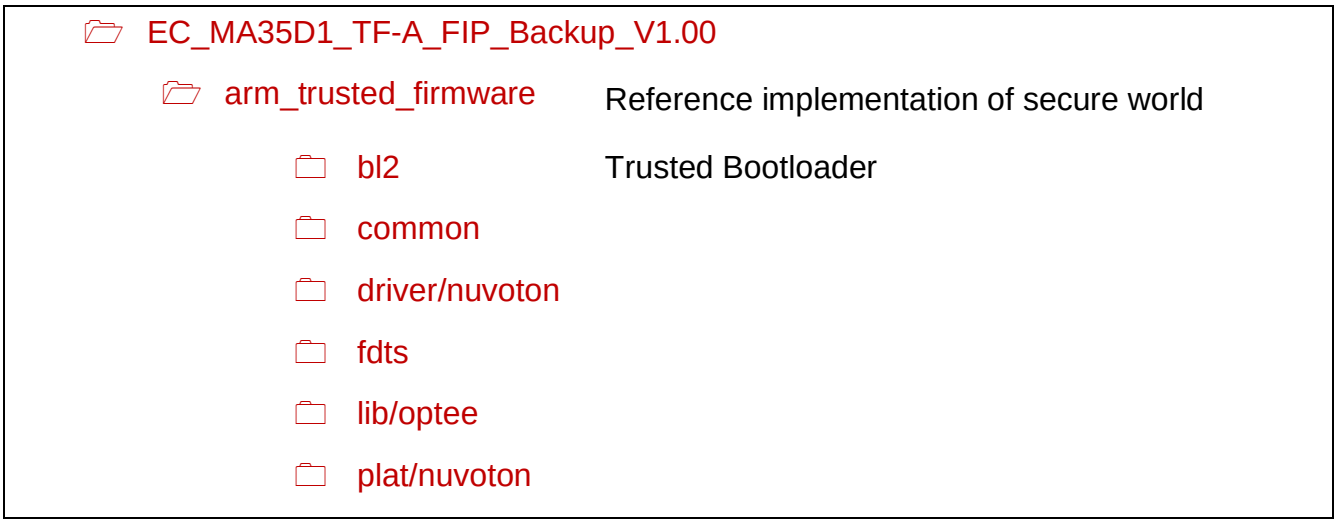


Figure 4-1 Directory Structure

5. Example Code Execution

By default, the MA35D1 builds the TF-A images. To run this example successfully, you must modify the relevant source code under the **arm-trusted-firmware** directory.

Below are the steps for Buildroot, using an SD card as the external storage device:

1. Enter the MA35D1 Buildroot directory (the exact path may vary depending on your environment configuration):

```
user@e9e6e7eb0c21:~/shared/buildroot_2024/output/build/arm-trusted-firmware-custom$ ◦
```

2. Copy the '.c', '.h', and '.dtsi' files from each folder in the example project and replace the corresponding files under '*arm-trusted-firmware-custom'*. Note that this operation is a **replacement**, not a direct copy of the entire example folder into the '*arm-trusted-firmware-custom*' directory.

3. In menuconfig, enable Secure Boot and configure the corresponding keys.

```
Bootloaders --- >
  [*] MA35D1 Secure Boot
  (0A0BC81E5AFBF836C5E0CFBEA12C1E269A2EBC3B0B6EC39EEE1C7142F760EBC4) AES Key
  (8F1571E9975006A545DF854264F7B18934B5CB2EE619E66DBC0FBEA52C71C919) ECDSA Key
```

4. Remove the *build* folder under *arm-trusted-firmware-custom*.

5. Rebuild arm-trusted-firmware-custom.

```
make arm-trusted-firmware-rebuild
```

6. Enter the following path:

```
user@e9e6e7eb0c21:~/shared/buildroot_2024/board/nuvoton/ma35d1/nuwriter$
```

7. Modify the offset address of "uboot-env.bin-sdcard" and add the backup FIP image configuration to the *pack-sdcard.json* file.

```
{
  "image": [
    {
      "offset": "0x0",
      "file": "MBR.sdcard.bin",
      "type": 0
    },
    {
      "offset": "0x400",
      "file": "header.bin",
      "type": 0
    },
    {
      "offset": "0x20000",
      "file": "nuwriter/bl2.dtb",
      "type": 0
    },
```

```
    {
      "offset": "0x30000",
      "file": "nuwriter/bl2.bin",
      "type": 0
    },
    {
      "offset": "0x50000",
      "file": "uboot-env.bin-sdcard",
      "type": 0
    },
    {
      "offset": "0xC0000",
      "file": "fip.bin-sdcard",
      "type": 0
    },
    {
      "offset": "0x2c0000",
      "file": "Image.dtb",
      "type": 0
    },
    {
      "offset": "0x300000",
      "file": "Image",
      "type": 0
    },
    {
      "offset": "37748736",
      "file": "rootfs.ext4",
      "type": 0
    },
    {
      "offset": "0x11C0000",
      "file": "fip.bin-sdcard",
      "type": 0
    }
  ]
}
```

8. Rebuild the MA35D1 image.

```
make
```

9. Use NuWriter to program the *pack* image.
10. Use NuWriter to erase the region where the primary *fip.bin-sdcard* has been written (0xc0000-0x1c0000, referring to the *pack_sdcard.json*).
11. After a successful build, the NuWriter will generate an "*otp_key.json*" file under *output/images/nuwriters*, containing the ECDSA public key and AES key. These keys should be programmed into the MA35D1 IBR Key region via NuWriter.
12. Set the Power-on configuration DIP switch on the MA35D1 evaluation board to boot from SD0, and power on the MA35D1 board.
13. Once entering BL2, the system begins loading the BL3x images. Since the primary FIP image was erased in step 10, BL2 fails to retrieve the image and reports a boot failure. It

then switches to the backup FIP location to load the image and completes the boot process successfully.

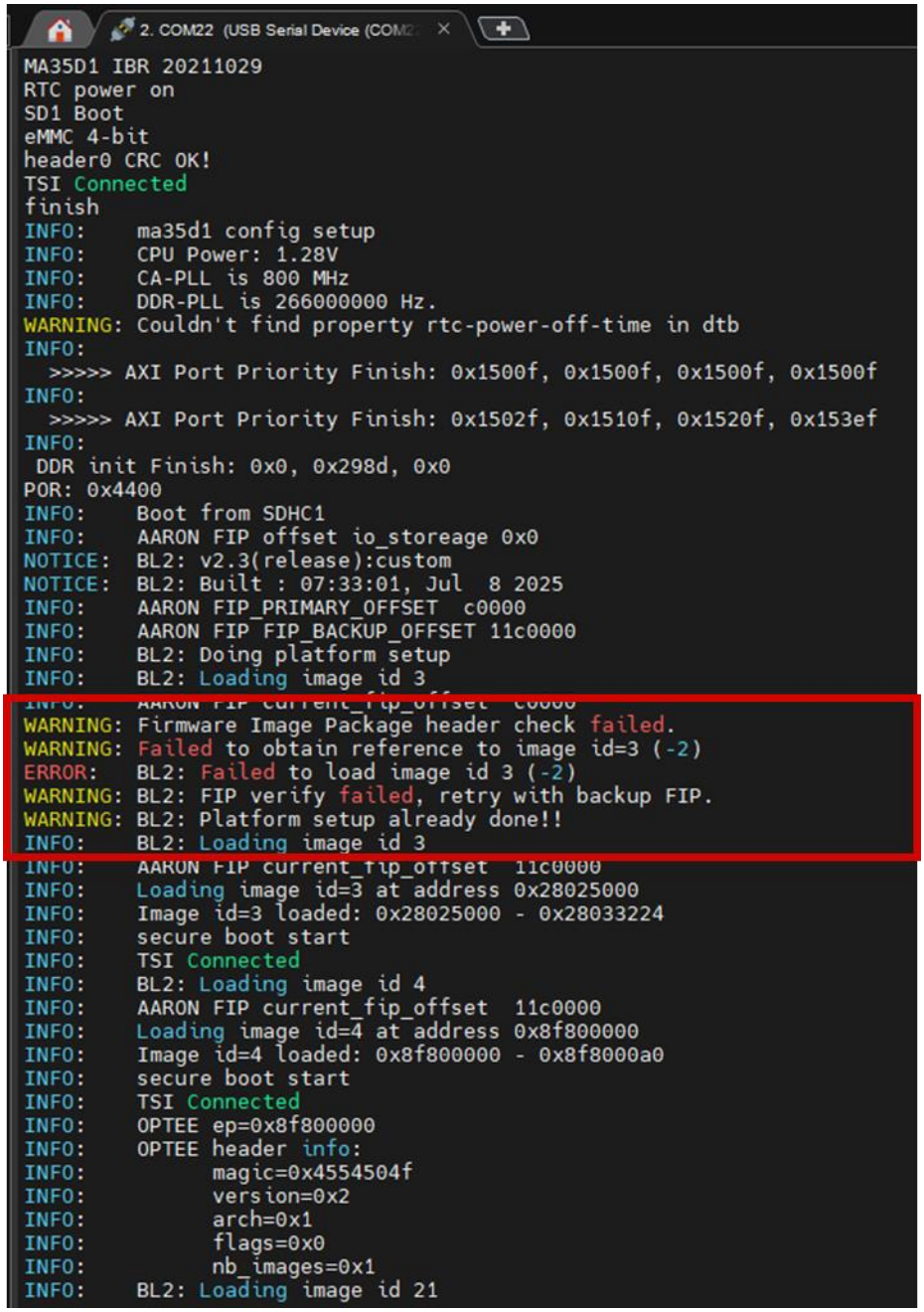


Figure 5-1 Executed Results – Primary Image Loading Failed

14. Second test: Verify the FIP image signed with an incorrect private key. First, clean the *arm-trusted-firmware folder*.

```
make arm-trusted-firmwared-dirclean
```

15. Reconfigure the ECDSA private key.

```
Bootloaders --- >
```

```
[*] MA35D1 Secure Boot
(0A0BC81E5AFBF836C5E0CFBEA12C1E269A2EBC3B0B6EC39EEE1C7142F760EBC4) AES Key
(8F1571E9975006A545DF854264F7B18934B5CB2EE619E66DBC0FBEA52C71C918) ECDSA Key
```

16. Rebuild the MA35D1 image.

17. Use NuWriter again to program the *fip.bin-sdcard* file signed with the invalid private key into the primary FIP storage location (0xc0000, referring to the *pack_sdcard.json*).

18. Reboot the MA35D1 evaluation board. After BL2 successfully loads the FIP image from the primary region, it performs the second-stage signature verification. Since the primary FIP is signed with an incorrect key, the verification fails, prompting BL2 to read from the backup FIP location and perform verification again. Once the verification succeeds, BL2 continues execution and completes the system boot process.

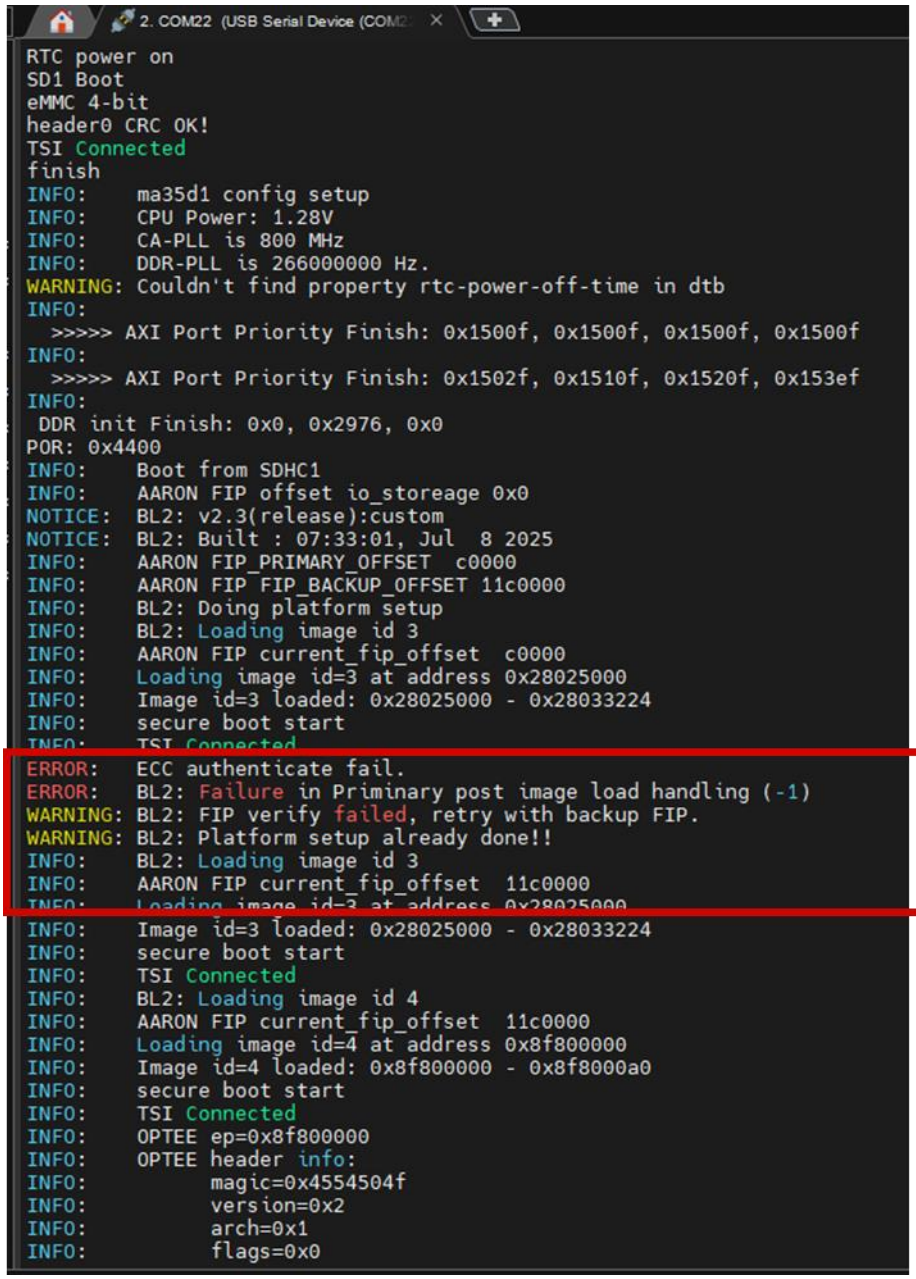


Figure 5-2 Executed Results – Primary Image Verification Failed

6. Revision History

Date	Revision	Description
2025.10.07	1.00	Initial version.

Important Notice

Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.

Please note that all data and specifications are subject to change without notice.
All the trademarks of products and companies mentioned in this datasheet belong to their respective owners.