Airoha IoT SDK for RTOS XBOOT Developer's Guide

Version: 1.0
Release date: 28 April 2017

Document revision history

Revision	Date	Description
1.0	28 April 2017	Initial release

Table of contents

Lists of tables and figures

1. Introduction

Airoha MT5932 is a highly integrated chipset containing a microcontroller unit (MCU), a low power 1x1 11n single-band Wi-Fi subsystem and a power management unit (PMU). The MCU is an ARM Cortex-M4 processor with floating point unit.

MT5932 itself doesn't have an internal flash memory and the binary image is loaded from host to the System Random Access Memory (SYSRAM) of MT5932 using an external boot (XBOOT) method. The implementation of XBOOT resides in the Boot ROM (read-only memory) of MT5932. After the binary image is loaded from the host, MT5932 can execute the image to provide Wi-Fi connectivity or other features.

This document guides the developers to implement and apply XBOOT to transfer binary images from host device to MT5932 acting as a slave device.

1.1. Host interface

On MT5932, XBOOT supports two host interfaces: Serial Peripheral Interface (SPI) and Secure Digital Input/Output (SDIO). MT5932 acts as slave in both SPI and SDIO interfaces with a source clock provided by the host.

1.2. Binary image transfer

The block diagram of the binary image transfer from host to MT5932 is shown in Figure 1. MT5932 plays the role of SPI/SDIO slave and the host plays the role of SPI/SDIO master. Detailed data flow is described below:

1) Host middleware prepares the binary image.
2) Host SPI/SDIO master driver sends the binary image to MT5932 according to the XBOOT sequence described in chapter 2, "XBOOT Sequence".
3) MT5932 SPI/SDIO slave driver receives the binary image from the host and puts the image into the SYSRAM.
4) MT5932 then executes the binary image from SYSRAM.

Note: The start address of SYSRAM is `0x04200000` and the maximum size of binary image is 378KB.

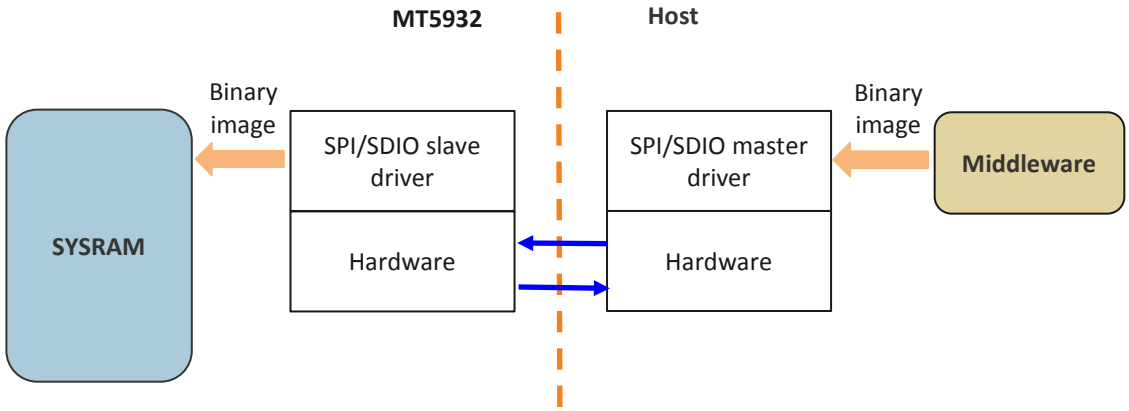

Figure 1. Binary image transfer from host to MT5932

2. XBOOT Sequence

The XBOOT mechanism is implemented on both MT5932 and the host. The implementation on MT5932 already resides in the Boot ROM (read-only memory). This section describes the XBOOT implementation sequence on the host device.

If your host device is based on Airoha MT2523, refer to the Airoha IoT SDK for RTOS get started guide to learn how to configure the hardware, program an application and create a binary image. For any other host, refer to its corresponding developer or get started guide.

An example project is also provided (in the SDK package) using MT7686 chipset as a host to demonstrate XBOOT flow.

2.1. Defining the XBOOT parameters

Start the implementation by defining types, enumerations and structures on the host side.

1) Define the type. All data types in XBOOT are 32-bit unsigned integer (U32) based.

```
typedef unsigned int    U32;
```

2) `XBOOT_CMD_ID`. Enumeration of command ID (32 bits).

```
typedef enum {
    BIN_LOAD_START          = 0,
    ACK_BIN_LOAD_START      = 1,
    GET_BIN                 = 2,
    ACK_GET_BIN             = 3,
    BIN_LOAD_END            = 4,
    ACK_BIN_LOAD_END        = 5,
    XBOOT_CMD_ID_END        = 0x0FFFFFFF
} XBOOT_CMD_ID;
```

3) `XBOOT_STATUS`. Enumeration of XBOOT status (32 bits). If `XBOOT_STATUS` is not `XBOOT_OK`, it means an error occurred.

```
typedef enum {
    XBOOT_OK            = 0,
    XBOOT_ERROR         = 0x1000,
    XBOOT_STATUS_END    = 0x0FFFFFFF
} XBOOT_STATUS;
```

4) `Xboot_cmd_status`. A structure to communicate the command status (32 bytes).

```
typedef struct Xboot_cmd_status {
    U32             m_magic;
    XBOOT_CMD_ID    m_msg_id;
    XBOOT_STATUS    m_status;
    U32             m_reserve[5];
} Xboot_cmd_status;
```

5) `Xboot_cmd_get_bin`. A structure to communicate the image information (32 bytes), see section 2.2, “XBOOT sequence flow”.

```
typedef struct Xboot_cmd_get_bin {
    U32               m_magic;
    XBOOT_CMD_ID      m_msg_id;
    U32               m_offset;
    U32               m_len;
    U32               m_reserve[4];
} Xboot_cmd_get_bin;
```

2.2. XBOOT sequence flow

XBOOT implementation has four stages, as described below:

1) Handshake: MT5932 sends a start command and the host replies with an acknowledgement.
2) Image information exchange: MT5932 and the host exchange the details of the binary image, such as the size of binary image.
3) Image transfer: The host transfers the binary image to MT5932.
4) Checksum and execution: After the binary image transfer is complete, MT5932 reports the checksum of the image back to the host and then executes the image.

2.2.1. Handshake stage

During handshake, XBOOT tries to establish connection between MT5932 and the host, as shown in Figure 2.

- MT5932 sends a start command and expects to receive an acknowledgement from the host within a certain time. If MT5932 fails to receive any acknowledgement from the host before the timeout, the process of XBOOT halts and MT5932 enters dead state where the Boot ROM stays in an infinite loop unable to recover from an error. MT5932 has to be reset to retry the handshake.
- The timeout for SPI (see section 2.3, “XBOOT through SPI”) and SDIO (see section 2.4, “XBOOT through SDIO”) are 150ms and 2s, respectively.

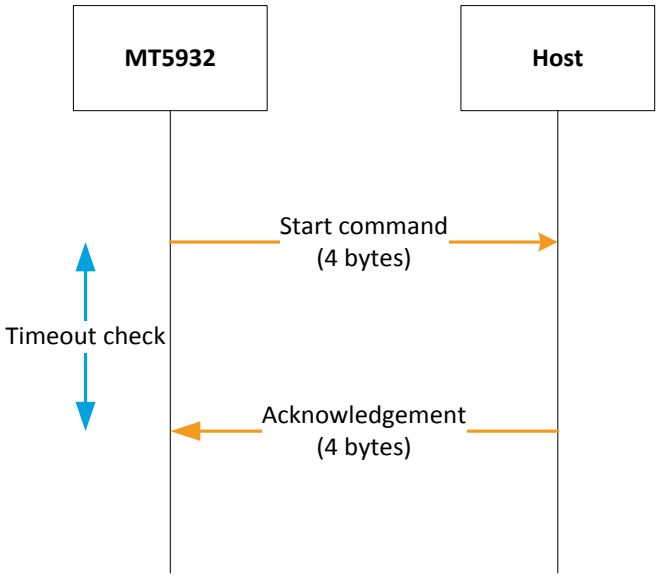

Figure 2. Handshake flow

2.2.2. Image information exchange stage

In this stage, MT5932 provides the maximum size of the binary image that MT5932 can accommodate which is 378KB. Then the host replies with an actual size, as shown in Figure 3.

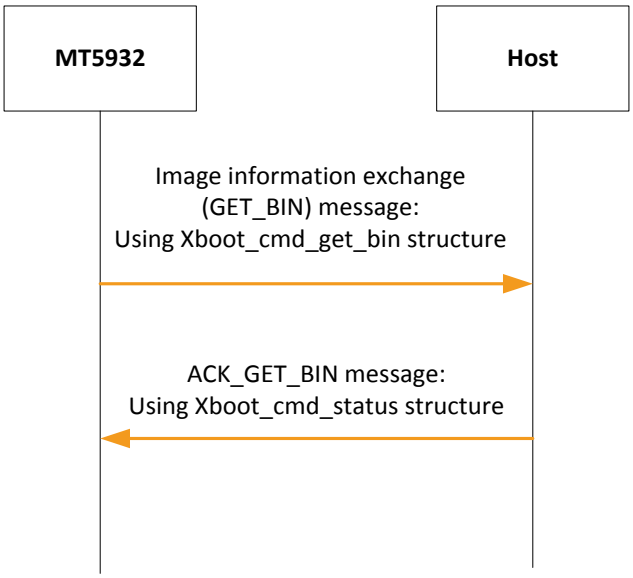

Figure 3. Image information exchange flow

The message content sent to the host with fixed values is shown below. The parameter `m_magic` indicates an ASCII value for “MCMD”.

```
Xboot_cmd_get_bin GetBinFromMT5932 = {
    .m_magic   = 0x444D434D,
    .m_msg_id  = GET_BIN,
    .m_offset  = 0,
    .m_len     = 0x5E800,
    .m_reserve = {0, 0, 0, 0}
};
```

The message content received from the host is shown below. The values of `m_magic` and `m_msg_id` are fixed. The host uses `m_status` to report the XBOOT status, and uses `m_reserve[0]` to report the actual size of binary image in bytes. The parameter `m_magic` indicates an ASCII value for “ACKM”.

```
Xboot_cmd_status AckGetBinFromHost = {
    .m_magic   = 0x4D4B4341,
    .m_msg_id  = ACK_GET_BIN,
    .m_status  = XBOOT_OK,
    .m_reserve = {0x5E800, 0, 0, 0, 0}
};
```

2.2.3. Image transfer stage

The host sends the binary image to MT5932, as shown in Figure 4. The size is described in `AckGetBinFromHost.m_reserve[0]`.

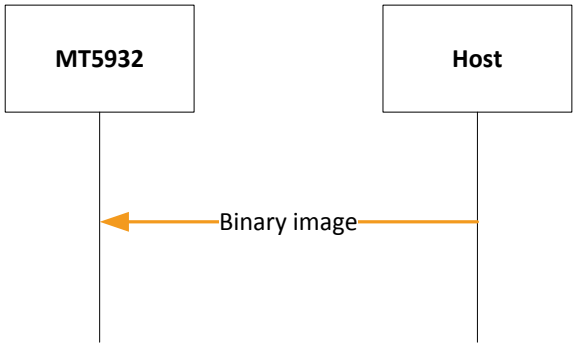

Figure 4. Transferring the binary image

2.2.4. Checksum and execution stage

Once the binary image transfer is complete, MT5932 reports the checksum back to the host and the host replies with an acknowledgement, as shown in Figure 5.

Then, Boot ROM jumps to SYSRAM to execute the binary image.

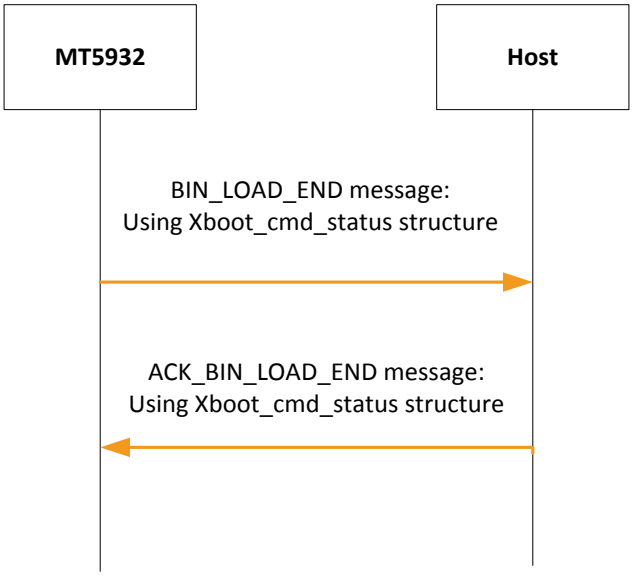

Figure 5. Checksum exchange flow

The content of `BIN_LOAD_END` with fixed values of `m_magic`, `m_msg_id` and `m_status` is shown below. MT5932 uses `m_reserve[0]` to report the checksum.

```
Xboot_cmd_status BinLoadEndFromMT5932 = {
    .m_magic   = 0x444D434D,
    .m_msg_id  = BIN_LOAD_END,
    .m_status  = XBOOT_OK,
    .m_reserve = {0, 0, 0, 0, 0}
};
```

The content of `ACK_BIN_LOAD_END` message with values of `m_magic` and `m_msg_id` is shown below. `m_status` is used to report the XBOOT status.

```
Xboot_cmd_status AckBinLoadEndFromHost = {
    .m_magic   = 0x4D4B4341,
```

```
    .m_msg_id  = ACK_BIN_LOAD_END,
    .m_status  = XBOOT_OK,
    .m_reserve = {0, 0, 0, 0, 0}
};
```

If the checksum received by the host does not match to the original checksum, the host would assume that an error occurred during the binary image transfer. In this case, the host should reset MT5932 to restart the XBOOT process.

2.2.5. Checksum

32 bits exclusive OR (XOR) is used to calculate the checksum.

2.3. XBOOT through SPI

MT5932 acts as SPI slave and cannot send data to the host directly. The host has to stay at read-waiting state by using “Configure Read” and “Read” commands to receive data from the slave.

In the section, XBOOT flow is implemented through SPI interface.

2.3.1. Handshake stage

Apply “Configure Read” and “Read” commands before the start command, then “Configure Write” and “Write” commands before receiving acknowledgement from the host. The start command has a magic number of 0x54535053 and 0x4B415053 for acknowledgement. The flow is shown in Figure 6.

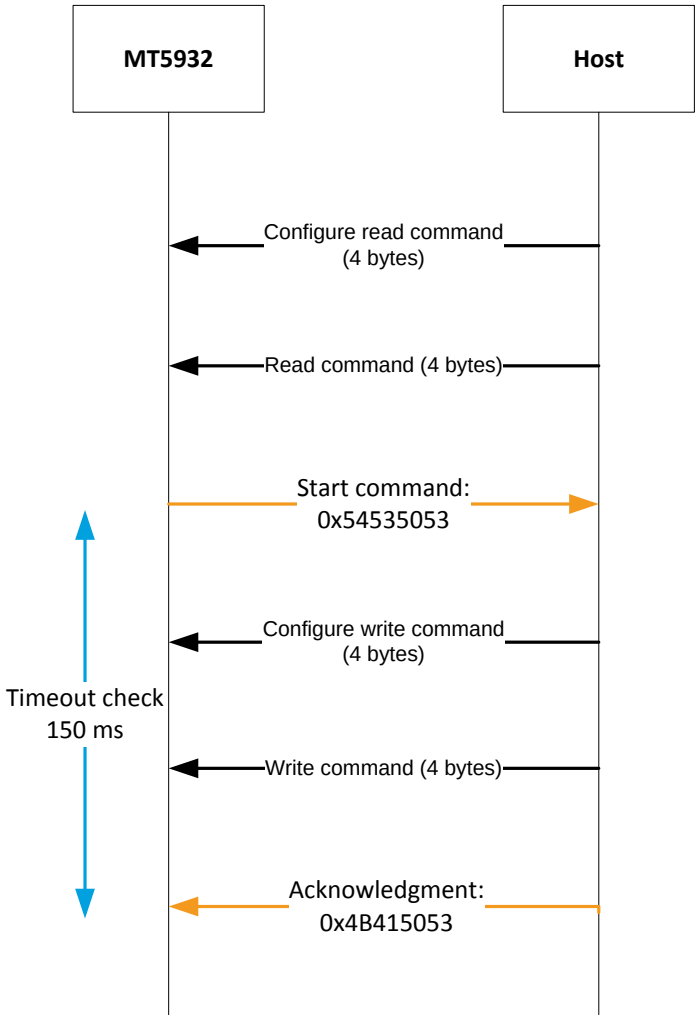

Figure 6. Handshake flow through SPI

Note: The magic number 0x54535053 is the ASCII representation of “SPSD”, indicating “SPI Start”. The magic number 0x4B415053 is the ASCII representation of “SPAK”, indicating “SPI Acknowledgement”.

2.3.2. Image information exchange stage

In the stage, apply “Configure Read” and “Read” commands before “GET_BIN”, then add “Configure Write” and “Write” commands before “ACK_GET_BIN”, as shown in Figure 7.

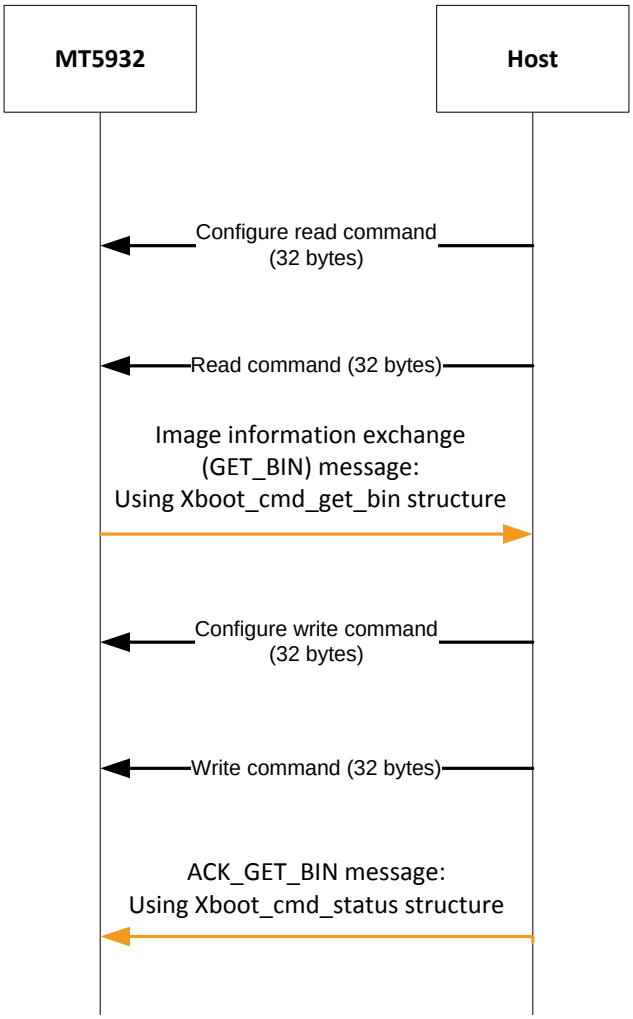

Figure 7. Image information exchange flow through SPI

2.3.3. Image transfer stage

In load binary stage, the host transfers the binary image, as shown in Figure 8.

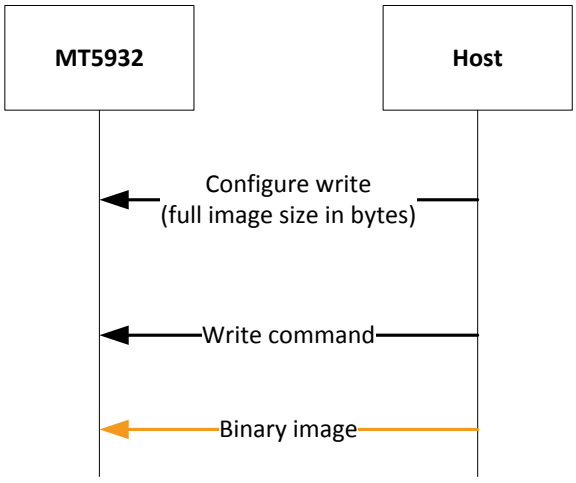

Figure 8. Transferring image through SPI

2.3.4. Checksum and execution stage

Once the binary image transfer is complete, XBOOT enters `BIN_LOAD_END` stage. In this stage, MT5932 reports the checksum back to the host and the host replies with an acknowledgement, as shown in Figure 9. Then, Boot ROM jumps to SYSRAM to execute the binary image.

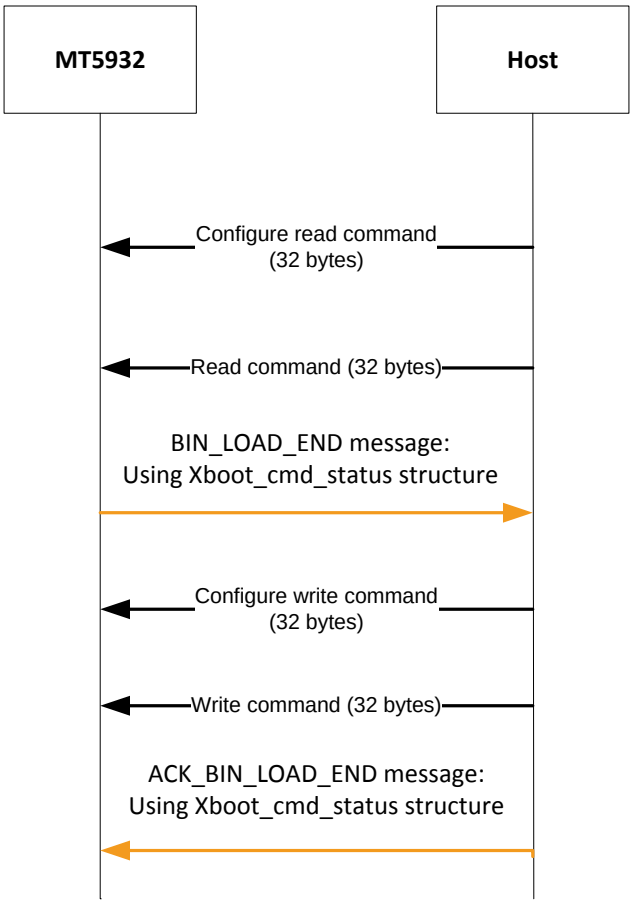

Figure 9. Checksum exchange flow through SPI

2.4. XBOOT through SDIO

In the section, XBOOT flow is implemented through SDIO interface.

2.4.1. Handshake stage

The data communication is performed using mailbox, given the small amount of data. The flow is shown in Figure 10 with a start command (0x53444254) and acknowledgement (0x534254FF).

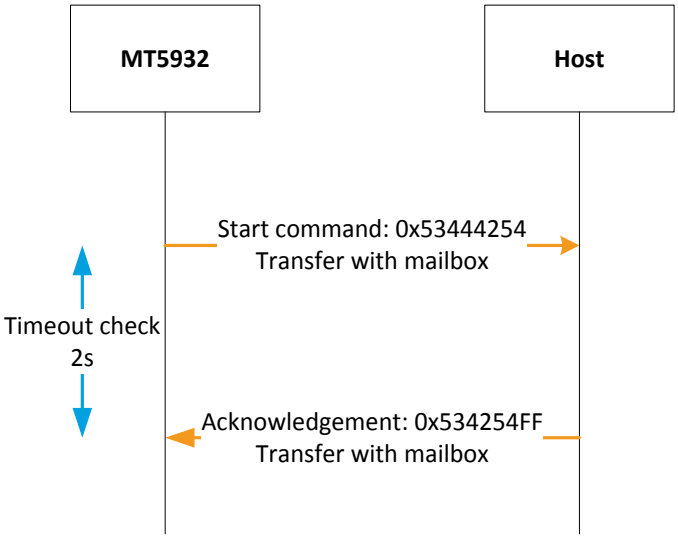

Figure 10. Handshake flow through SDIO

Note: The magic number 0x53444254 is the ASCII representation of “SDBT”, indicating “SDIO boot”. The magic number 0x534254FF is the ASCII representation of “SBT”, indicating “SDIO boot acknowledgement”.

2.4.2. Image information exchange stage

The transfer is implemented with a buffer instead of mailbox to transfer 32-byte structure. The flow is shown in Figure 11.

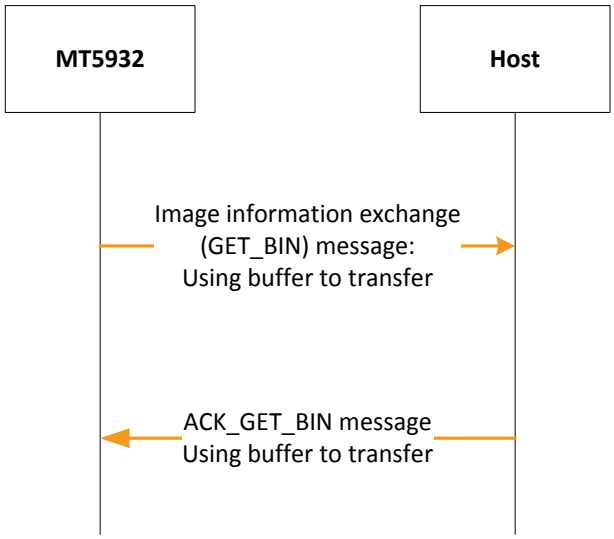

Figure 11. Image information exchange flow through SDIO

SDIO interface cannot support sending the full binary image (378KB) in a single transmission, thus MT5932 and host have to negotiate the block size of transmission.

The content of GET_BIN message is shown below. MT5932 uses `m_reserve[0]` to describe the maximum block size defined as 2048 bytes.

```
Xboot_cmd_get_bin GetBinFromMT5932_SDIO = {
    .m_magic    = 0x444D434D,
```

```
    .m_msg_id  = GET_BIN,
    .m_offset  = 0,
    .m_len     = 0x5E800,
    .m_reserve = {2048, 0, 0, 0}
};
```

The content of ACK_GET_BIN message is shown below. The host uses `m_reserve[1]` to report the actual block size in bytes, which must less than or equal to 2048.

```
Xboot_cmd_status AckGetBinFromHost_SDIO = {
    .m_magic   = 0x4D4B4341,
    .m_msg_id  = ACK_GET_BIN,
    .m_status  = XBOOT_OK,
    .m_reserve = {0x5E800, 2048, 0, 0, 0}
};
```

2.4.3. Image transfer stage

In load binary stage, the host partitions the entire binary image into several blocks. Then transfers each block to MT5932 sequentially, as shown in Figure 12.

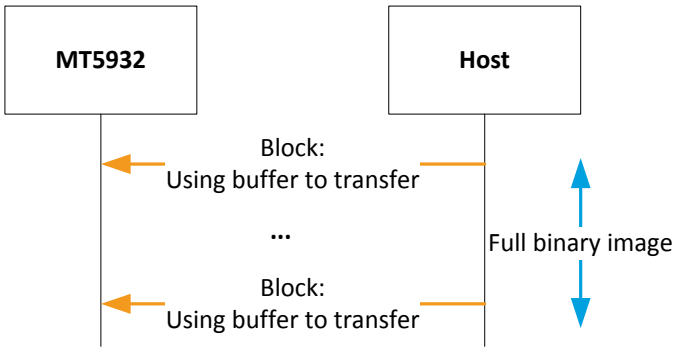

Figure 12. Transferring image through SDIO

2.4.4. Checksum and execution stage

Once the binary image transfer is complete, XBOOT enters `BIN_LOAD_END` stage. In this stage, MT5932 reports the checksum back to the host and the host replies with an acknowledgement, as shown in Figure 13. Then, Boot ROM jumps to SYSRAM to execute the binary image.

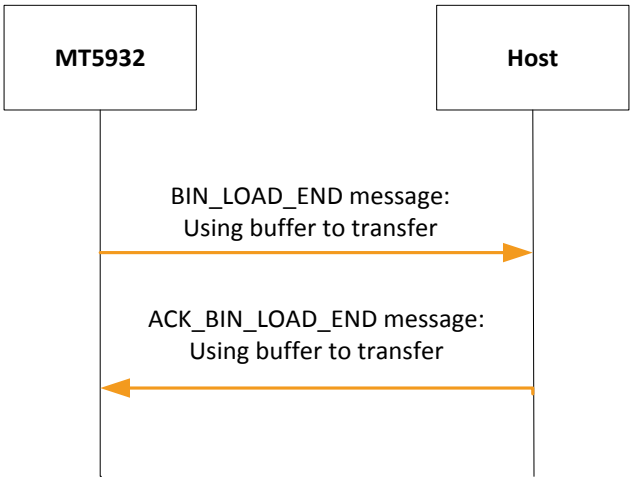

Figure 13. Checksum exchange flow through SDIO