APPLICATION NOTE

Apollo SoC Multi-Protocol Bootloader

Ultra-Low Power Apollo SoC Family

A-SOCAP3-ANGA02EN v1.1

Legal Information and Disclaimers

AMBIQ MICRO INTENDS FOR THE CONTENT CONTAINED IN THE DOCUMENT TO BE ACCURATE AND RELIABLE. THIS CONTENT MAY, HOWEVER, CONTAIN TECHNICAL INACCURACIES, TYPOGRAPHICAL ERRORS OR OTHER MISTAKES. AMBIQ MICRO MAY MAKE CORRECTIONS OR OTHER CHANGES TO THIS CONTENT AT ANY TIME. AMBIQ MICRO AND ITS SUPPLIERS RESERVE THE RIGHT TO MAKE CORRECTIONS, MODIFICATIONS, ENHANCEMENTS, IMPROVEMENTS AND OTHER CHANGES TO ITS PRODUCTS, PROGRAMS AND SERVICES AT ANY TIME OR TO DISCONTINUE ANY PRODUCTS, PROGRAMS, OR SERVICES WITHOUT NOTICE.

THE CONTENT IN THIS DOCUMENT IS PROVIDED "AS IS". AMBIQ MICRO AND ITS RESPECTIVE SUPPLIERS MAKE NO REPRESENTATIONS ABOUT THE SUITABILITY OF THIS CONTENT FOR ANY PURPOSE AND DISCLAIM ALL WARRANTIES AND CONDITIONS WITH REGARD TO THIS CONTENT, INCLUDING BUT NOT LIMITED TO, ALL IMPLIED WARRANTIES AND CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHT.

AMBIQ MICRO DOES NOT WARRANT OR REPRESENT THAT ANY LICENSE, EITHER EXPRESS OR IMPLIED, IS GRANTED UNDER ANY PATENT RIGHT, COPYRIGHT, MASK WORK RIGHT, OR OTHER INTELLECTUAL PROPERTY RIGHT OF AMBIQ MICRO COVERING OR RELATING TO THIS CONTENT OR ANY COMBINATION, MACHINE, OR PROCESS TO WHICH THIS CONTENT RELATE OR WITH WHICH THIS CONTENT MAY BE USED.

USE OF THE INFORMATION IN THIS DOCUMENT MAY REQUIRE A LICENSE FROM A THIRD PARTY UNDER THE PATENTS OR OTHER INTELLECTUAL PROPERTY OF THAT THIRD PARTY, OR A LICENSE FROM AMBIQ MICRO UNDER THE PATENTS OR OTHER INTELLECTUAL PROPERTY OF AMBIQ MICRO.

INFORMATION IN THIS DOCUMENT IS PROVIDED SOLELY TO ENABLE SYSTEM AND SOFTWARE IMPLEMENTERS TO USE AMBIQ MICRO PRODUCTS. THERE ARE NO EXPRESS OR IMPLIED COPYRIGHT LICENSES GRANTED HEREUNDER TO DESIGN OR FABRICATE ANY INTEGRATED CIRCUITS OR INTEGRATED CIRCUITS BASED ON THE INFORMATION IN THIS DOCUMENT. AMBIQ MICRO RESERVES THE RIGHT TO MAKE CHANGES WITHOUT FURTHER NOTICE TO ANY PRODUCTS HEREIN. AMBIQ MICRO MAKES NO WARRANTY, REPRESENTATION OR GUARANTEE REGARDING THE SUITABILITY OF ITS PRODUCTS FOR ANY PARTICULAR PURPOSE, NOR DOES AMBIQ MICRO ASSUME ANY LIABILITY ARISING OUT OF THE APPLICATION OR USE OF ANY PRODUCT OR CIRCUIT, AND SPECIFICALLY DISCLAIMS ANY AND ALL LIABILITY, INCLUDING WITHOUT LIMITATION CONSEQUENTIAL OR INCIDENTAL DAMAGES. “TYPICAL” PARAMETERS WHICH MAY BE PROVIDED IN AMBIQ MICRO DATA SHEETS AND/OR SPECIFICATIONS CAN AND DO VARY IN DIFFERENT APPLICATIONS AND ACTUAL PERFORMANCE MAY VARY OVER TIME. ALL OPERATING PARAMETERS, INCLUDING “TYPICALS” MUST BE VALIDATED FOR EACH CUSTOMER APPLICATION BY CUSTOMER’S TECHNICAL EXPERTS. AMBIQ MICRO DOES NOT CONVEY ANY LICENSE UNDER NEITHER ITS PATENT RIGHTS NOR THE RIGHTS OF OTHERS. AMBIQ MICRO PRODUCTS ARE NOT DESIGNED, INTENDED, OR AUTHORIZED FOR USE AS COMPONENTS IN SYSTEMS INTENDED FOR SURGICAL IMPLANT INTO THE BODY, OR OTHER APPLICATIONS INTENDED TO SUPPORT OR SUSTAIN LIFE, OR FOR ANY OTHER APPLICATION IN WHICH THE FAILURE OF THE AMBIQ MICRO PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR. SHOULD BUYER PURCHASE OR USE AMBIQ MICRO PRODUCTS FOR ANY SUCH UNINTENDED OR UNAUTHORIZED APPLICATION, BUYER SHALL INDEMNIFY AND HOLD AMBIQ MICRO AND ITS OFFICERS, EMPLOYEES, SUBSIDIARIES, AFFILIATES, AND DISTRIBUTORS HARMLESS AGAINST ALL CLAIMS, COSTS, DAMAGES, AND EXPENSES, AND REASONABLE ATTORNEY FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PERSONAL INJURY OR DEATH ASSOCIATED WITH SUCH UNINTENDED OR UNAUTHORIZED USE, EVEN IF SUCH CLAIM ALLEGES THAT AMBIQ MICRO WAS NEGLIGENT REGARDING THE DESIGN OR MANUFACTURE OF THE PART.

Revision History

Revision	Date	Description
1.0	April 12, 2022	Initial release
1.1	December 1, 2022	Processed branding updates

Reference Documents

Document ID	Description

Table of Contents

List of Tables

List of Figures

SECTION

1

Introduction

The Multi-Protocol Bootloader itself is a simple program that resides in the first few pages of flash memory. It runs on power-up or on any reset event, and checks to see if there is a valid application to run. If it finds one, it will run that application. If notor if a host processor requests a forced update—it will listen for a new binary image over SPI, I^2C, or UART, and proceed to update the internal flash with the new application.

The multi-protocol bootloader functionality can be primarily categorized into the following:

- Verification and Execution of the flash-resident image

  Verifying the integrity of the main program image in the flash. It protects against image corruption or tampering. Also, it could optionally be used to authenticate the image with a secure version of bootloader.

  At the end of the boot process, the multi-protocol bootloader configures the system to run the verified image.

- Image upgrade

  In many cases, users of Ambiq's microcontrollers will need to update a program in internal flash memory without using the standard SWD interface. Often, this is to solve the problem of field updates or programming assembled boards in a production environment. The Multi-Protocol Bootloader described in this document provides a way for a host processor to update the program in an Apollo or Apollo2 device.

  Multiboot can also be built to support Over the Air image upgrade. This is detailed in *Section 7 Over the Air (OTA) Upgrade on page 23*.

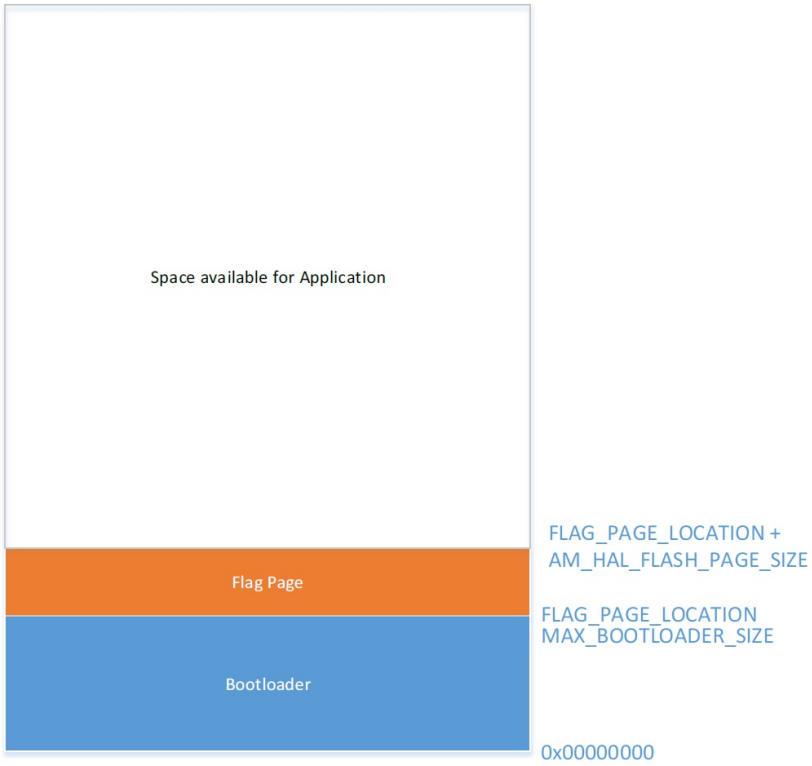
SECTION

2

Memory Map

Figure 2-1 is a description of the basic memory map used by the bootloader:

Figure 2-1: Flash Memory Map



As shown in the diagram above, the bootloader consumes the first few KB of flash memory. One flash page, called the Flag Page, is reserved for dedicated use by the bootloader to keep necessary information used to validate the flash resident main image. Rest of the flash is available for the main application image (and other third party images, if needed) to use.

Note that Flag page could be located anywhere in the flash, and need not be immediately after bootloader.

SECTION

3

Verification of Flash-Resident Image

Once the bootloader has been programmed into flash, it is ready to talk to a host processor and download a new application. When it does this, the bootloader will store some information about the application in the Flag Page, including the starting address, the length, and a CRC value. The default location for the Flag Page is 0x00004000, but the bootloader can be recompiled to place the flag page in any location.

The exact contents of the Flag Page are shown in Figure 3-1:

Figure 3-1: Flag Page Contents

```
//*****************************************************************************
//
// Structure to keep track of boot image information.
// In the flash, the structure contains a 4 byte CRC for integrity
// verification. It needs to be ensured that the size of this structure is
// not more than AM_HAL_FLASH_PAGE_SIZE bytes
//
//*****************************************************************************
typedef struct
{
    // Starting address where the image was linked to run.
    uint32_t *pui32LinkAddress;

    // Length of the executable image in bytes.
    uint32_t ui32NumBytes;

    // CRC-32 Value for the full image.
    uint32_t ui32CRC;

    // Override GPIO number. (Can be used to force a new image load)
    uint32_t ui32OverrideGPIO;

    // Polarity for the override pin.
    uint32_t ui32OverridePolarity;

    // Stack pointer location.
    uint32_t *pui32StackPointer;

    // Reset vector location.
    uint32_t *pui32ResetVector;

    // Protection status of image in flash
    uint32_t bEncrypted;

    // CRC-32 value of this structure
    uint32_t ui32Checksum;
}
am_bootloader_image_t;
```

As shown here, the Flag Page structure is only 9 words long. It contains just enough information to find the downloaded application, verify its integrity, and run it. Field **bEncrypted** is a place holder for secure bootloader, where it is used to tell the bootloader program about the flash protection features in effect for the image. This feature is not currently implemented.

The flag page is entirely operated by the bootloader and shouldn't require any interaction from the user application.

When programming the flag page in the flash, the bootloader also updates a 4 byte CRC value in the last 4 bytes. The same is verified by the bootloader to assure integrity of the flag page before using the contents therein.

SECTION

4

Image Update

The following sections explain how to use the bootloader to perform updates of the main application components. The main image can be updated two different ways:

- Image upgrade initiated by a host communicating directly with the bootloader using SPI/ I^2C/UART
- Image upgrade using the OTA (Over the air) functionality built in to the main application

This section covers the image upgrade directly using the bootloader. OTA upgrade is described in *Section 7 Over the Air (OTA) Upgrade on page 23*.

Multi-Protocol Bootloader implements a message based protocol to allow host to control the bootloader behavior and also allows upgrading the image on the flash. Each message is identified by a 4 byte command, followed by command dependent message contents.

Multi-Protocol Bootloader supports connectivity to an external host using SPI, I^2C or UART. The physical signaling used by the bootloader changes slightly depending on the interface used. The following sections show how commands can be sent to the bootloader over various serial protocols.

4.1 SPI Boot Protocol

The Multi-Protocol Bootloader is designed to use a simple SPI based protocol. The Apollo or Apollo2 device operates as a slave, and can send messages back to a host processor with the help of a dedicated interrupt line.

4.1.1 SPI Command Format

Bootloader commands are sent in two separate SPI frames. The first SPI frame starts with 0x84, and contains the command parameters. The second SPI frame starts with 0x80, and contains the command word. The boot slave will execute the

command immediately after CS returns to logic HIGH at the end of the second SPI frame.

> **NOTE:** If a command has no parameters, the host processor does not need to send a parameter frame.

The following diagrams show the command format:

Figure 4-1: Parameter Frame

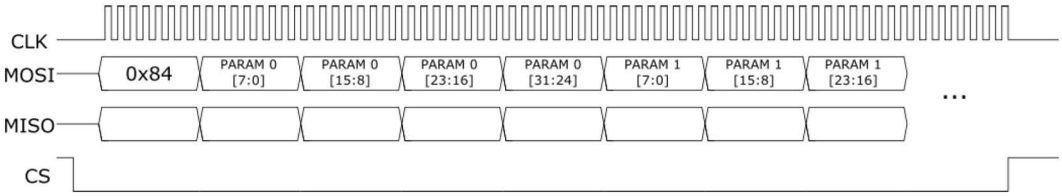
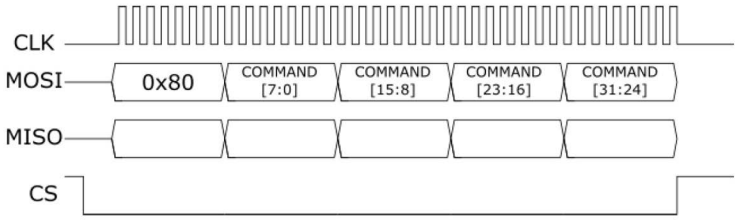


Figure 4-2: Command Frame



4.2 I²C Boot Protocol

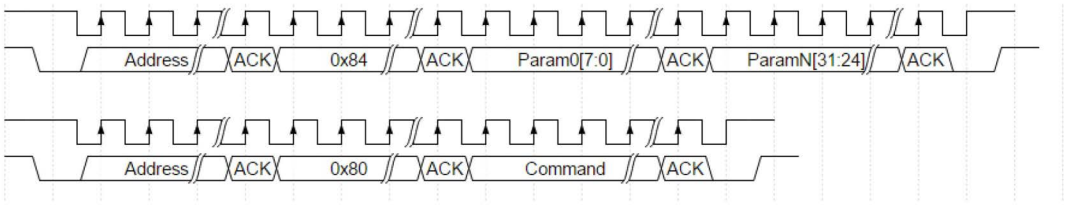
The Multi-Protocol Bootloader is designed to use a simple I²C-based protocol. The Apollo or Apollo2 device operates as a slave, and can send messages back to a host processor with the help of a dedicated interrupt line.

4.2.1 I²C Command Format

The I²C command format is similar to SPI. First, the host sends a parameter frame starting with the slave address and the byte 0x84. Second, the host must send the command frame, which starts with the slave address and the byte 0x80. Both commands and parameters are 32-bits long, and should be sent Least Significant Byte first.

> **NOTE:** If a command has no parameters, the host processor does not need to send a parameter frame..

Figure 4-3: I2C Command Format



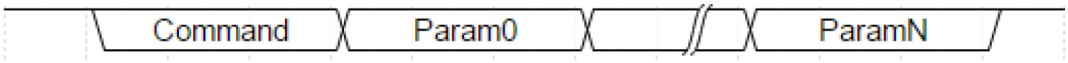
4.3 UART Boot Protocol

The Multi-Protocol Bootloader can also use a serial UART interface to connect to the host. The Rx and Tx serial Data is then used to transmit and received the bootloader messages.

4.3.1 UART Command Format

UART commands are very straightforward. They do not require multiple frames, additional overhead bytes, or INTERRUPT signaling because the slave is full-duplex. Each command or parameter is 32-bits long, sent LSB first.

Figure 4-4: UART Command Format



4.4 Supported Commands

This section describes commands currently supported by the Multi-Protocol Bootloader. Any command words not listed in this section are reserved for future commands.

4.4.1 Command 0x00000000: ACK

Acknowledge a message from the boot slave. This command has no response.

Parameters: None

4.4.2 Command 0x00000002: New Image

Send this command to start a new image. This command does not support any encryption. It is used to download firmware from a plain (non-secure) binary. The

host should wait for a response of READY or ERROR before sending additional packets.

Table 4-1: New Image Parameters

Parameter #	Description	Example Value	Notes
0	Link Address	0x00008000	
1	Image Size	0x000004C0	(1216 bytes)
2	Image CRC	0xE5D761C0	

4.4.3 Command 0x00000003: New Data Packet

This command is used to transfer binary data from the host to the slave. Send this command after a New Image packet to transfer the actual binary. The same command is used for both secure and non-secure downloads. The host should wait for a response of READY before sending additional packets.

Table 4-2: New Data Packet Parameters

Parameter #	Description	Example Value	Notes
0	Data Length (bytes)	0x00000020	Length in bytes of binary data that follows
1 - N	Image Size	0x000004C0	(1216 bytes)

The maximum length of the binary data that can be carried in one packet is restricted based on the underlying physical medium. When using I2C/SPI, it is restricted to 112 Bytes.

4.4.4 Command 0x00000004: Reset and Run

Commands the slave to execute the newly downloaded image. Send this after receiving an **IMAGE_COMPLETE** message from the slave if the slave should start running the new image.

This command also causes the slave device to save the image information to its internal “flag page”. This means that it will run the downloaded image on every power-up instead of starting the bootloader.

This command will have no response.

Parameters: None

4.4.5 Command 0x00000005: Set Override Pin

This optional command may be used at any time between a New Image command and a Reset and Run command to set the BOOT-Override pin behavior. On later reset events, the host can use the BOOT-Override pin to force an image upgrade instead of running the image in the flash.

The host should wait for a response of READY before sending additional packets.

Table 4-3: Set Override Pin Parameters

Parameter #	Description	Example Value	Notes
0	Override GPIO	12	GPIO number for the new Boot pin
1	Override Polarity	0x0	0x0: Active LOW, 0x1: Active HIGH

4.4.6 Command 0x00000006: Bootloader Version

The host may use this command to check the bootloader version number. Ambiq will update this number for future bootloader releases. The host should wait for a response before sending additional commands.

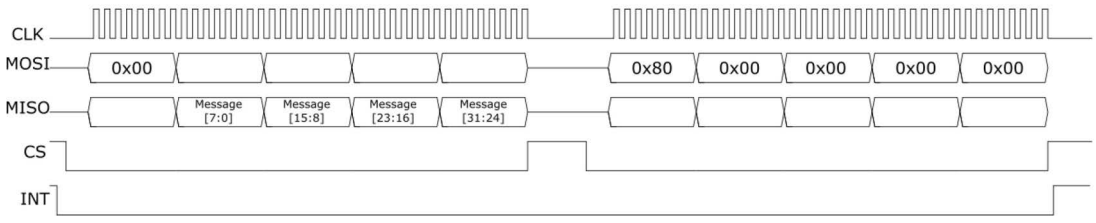
Parameters: None

4.5 Response Formats

During the boot process, the Bootloader slave will sometimes send messages back to the host. Each message will start with the slave asserting the INT pin. When this happens, the host should start a new SPI or I^2C frame in the following format. The slave message will always be 4 bytes long. After receiving a message from the slave, the boot host should send an ACK packet. This ensures that the INT line returns to the inactive state before the next command starts.

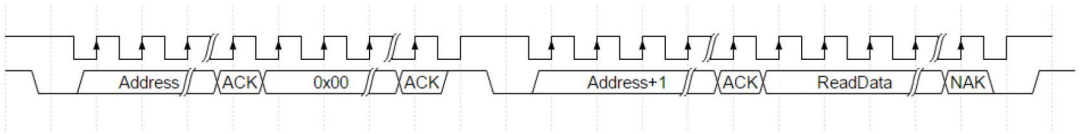
Figure 4-5 shows a SPI slave message followed by a host ACK frame:

Figure 4-5: SPI Slave Message Followed by a Host ACK Frame



This image shows an I^{2}C slave message. Please note the repeated-start condition. For the I^{2}C case, the "Host ACK" packet (not shown) is also a command packet where the command word is 0x00000000.

Figure 4-6: I^{2}C Slave Message



4.5.1 Supported Slave Responses

There are the currently supported slave messages. Ambiq may add additional messages in future versions of the bootloader.

Table 4-4: Supported Slave Responses

Message	Encoding	Description
READY	0x00000000	Ready for additional packets.
ERROR	0x00000005	Error. Start over with New Image packet.
IMAGE_COMPLETE	0x00000003	Received all data bytes, CRC is correct.
BAD_CRC	0x00000004	Received all data bytes, but CRC is incorrect. Start over.
BL_VERSION	(hex version number)	Corresponds to bootloader version number.

4.6 Pin Configuration for Connectivity with Host

Figure 4-5 lists the Pins used for the I^{2}C/SPI Slave operation. INT pin is configurable, and the value shown below is just a sample value.

Table 4-5: Pins Used for the I^{2}C/SPI Slave Operation

GPIO Number	SPI Function	I^{2}C Function
0	SCK	SCL
1	MISO	SDA
2	MOSI	-
3	nCE	-
4	INT	INT

For UART, multiple options are possible by configuring the Pads accordingly. Following is a non-comprehensive list of possible options.

Table 4-6: Options for Configuring the Pads

GPIO Number	UART Function
35	TX (Apollo only)
36	RX (Apollo only)
22	TX (Apollo2 using UART0)
23	RX (Apollo2 using UART0)
39	TX (Apollo2 using UART1)
40	RX (Apollo2 using UART1)

4.7 Override Pin and the Protocol Selection

The first action the bootloader should take immediately on startup is override pin detection. In normal operation, if the bootloader detects a valid application in flash memory that matches the CRC located in the Flag Page, it will automatically run that image. The Override or BOOT pin is a way to tell the bootloader that it should prepare to receive a new application instead of running the old application.

The bootloader will look for the override pin very early in the startup procedure. If the override pin is in the “active” state (configured in the last “Set Override Pin” command), the bootloader will configure its serial interfaces and wait for a new application image from the host.

When boot override is in effect, or if there is no image in flash, bootloader puts itself in a forced upgrade mode. In this mode, a host connection is expected to download a new image.

When the slave processor first boots up, it will configure pin 0 as an input to determine whether it should use SPI or I^2C mode. If pin 0 is LOW at boot, the bootloader will use SPI (Mode 0). If pin 0 is high, it will use I^2C.

A bootloader could be implemented such that if no activity is detected from host on the SPI/I^2C for a short time, bootloader assumes UART as the connection mode.

If enabled for baudrate auto-detect, when host first connects to the slave, it is expected to send one byte 0x55 to the slave. Slave uses this known pattern to auto-detect the UART baudrate.

SECTION

5

Boot Procedure

Following describes the boot procedure when using the **multi_boot** example application supplied with the SDK. In order to download an image to an Apollo device, the host processor will need to perform the following steps in order:

1. Reset the Apollo or Apollo2 device holding the **BOOT** pin in the active state if applicable. The first boot-up will not require a **BOOT** pin to start the download process.
2. Wait for the Apollo or Apollo2 boot slave to send a **READY** signal.
3. If a **BOOT** pin is needed for a future update (e.g., after this new application is programmed into flash), the host must send a **Set Override Pin** command at this time. (Most applications will require this)
4. Send a **NEW_IMAGE** packet containing the appropriate image parameters, and wait for a READY packet in response.
5. Send **NEW_DATA_PACKET** commands containing the actual binary to be programmed. Wait for a READY packet in response to each **NEW_DATA_PACKET**.
6. After the last data packet, the slave should send an **IMAGE_COMPLETE** packet
7. Send a **RESET_AND_RUN** packet to execute the new firmware image.

5.1 Multi-Segment Images

If the main application image is divided into separate individual segments, each segment is programmed by sending a **NEW_IMAGE** command, followed by a series of **NEW_DATA_PACKET** commands.

The last segment to be programmed should correspond to the main image entry point to which the bootloader passes execution control to. **RESET_AND_RUN** command should follow the last segment programming.

SECTION

6

Secure Image Upgrade

For deployments requiring enhanced security, a secure version of the Multi-Protocol Bootloader can be used. This is provided as a build option in AmbiqSuite SDK.

For this purpose, the **New Image** command is extended as follows.

Table 6-1: Secure Image Upgrade

Parameter #	Description	Example Value	Notes
0	Link Address	0x00008000	
1	Image Size	0x000004C0 (1216 bytes)	The image can be encrypted if image confidentiality is desired
2	Image CRC	0xE5D761C0	(Calculated over the encrypted bytes)
3	Security Trailer Size	Variable	Size of the Security trailer that follows this field
Rest of the Payload	Security Trailer	Variable	The contents of this trailer can be customized to suit individual security needs like Authentication, Encryption etc.

The contents of the **Security Trailer** and the **Secure Image** depends on the implementation choice and security requirements of individual vendors.

For example, if Image Confidentiality and Authentication is a requirement, this protocol can be used to send an encrypted image to the bootloader. The Security trailer can be customized to include necessary information for successful decryption and authentication of the image (e.g., key-index, initialization vector, signature etc.).

The Ambiq Multi-Protocol Bootloader provides a flexible framework, which can be customized to suit individual security needs like Authentication, Encryption etc. Customization is implemented by means of security function hooks, which can be implemented as a plugin component to create a secure bootloader.

AmbiqSuite SDK includes a sample (though cryptographically insecure) implementation of this plugin to demonstrate the concept. This example is designed to demonstrate the framework which can be extended by the customer to integrate more secure algorithms.

6.1 Customizable Security Functions

Secure build of Multiboot provides following function hooks that can be implemented to suit specific deployment needs.

6.1.1 Initialization

Multiboot calls this function as part of processing the **NEW_IMAGE** message.

```
int
init_multiboot_secure(uint32_t length, uint32_t *pData,
                      bool bStoreInSram,
                      am_bootloader_image_t *psImage,
                      uint32_t *pProtect);
```

Table 6-2: Initialization Parameter Description

Parameter	Description
Length	Length of the security trailer
pData	Pointer to the security trailer
bStoreInSram	Indicates if the image can be accumulated in SRAM as a whole. If not set, multi boot would need to keep flashing the image segments as they arrive overwriting the existing content, even before the image could be verified.
psImage	Pointer to the image properties as operated upon by multiboot
pProtect	Used to pass information back to multiboot, if the flashed image needs any protection features. This is a place holder for future.

This function could be implemented to verify the security trailer itself, and initialize the security engine. It could also be used to verify the validity of the key used for encryption. It returns 0 if the verification succeeds.

6.1.2 Decryption

Multiboot calls this function as part of processing the **NEW_DATA_PACKET** message after accumulating enough data to correspond to one flash page size (or less if this is the last segment of the image).

```
void
multiboot_secure_decrypt(void *pData, uint32_t ui32NumBytes);
```

Table 6-3: Decryption Parameter Description

Parameter	Description
pData	Pointer to encrypted image data in SRAM
ui32NumBytes	Length of data

This function should implement an in-place decryption of the data using the selected security algorithm. The decryption engine should have been initialized as part of **init_multiboot_secure** implementation. This function should also compute the running CRC for the clear image.

6.1.3 Verification

Multiboot calls this function after processing the last image segment and making sure the CRC verification across the encrypted image was successful.

```
int
multiboot_secure_verify(uint32_t *pui32ClearCRC);
```

Table 6-4: Verification Parameter Description

Parameter	Description
pui32ClearCRC	Pointer used to return the CRC of the clear image (unless it is marked Copy-Protected)

This function could implement additional verification or authentication of the downloaded (and decrypted) image. It returns 0 if the verification succeeds. On successful verification, and if the image did not need copy-protection in flash, it also returns the CRC of the clear image in flash, so that on subsequent boots the bootloader can check the image for integrity.

SECTION

7

Over the Air (OTA) Upgrade

For production devices, it is likely that there would be need to support image upgrades without requiring direct physical connection to host. Multi-Protocol Bootloader can be built to support OTA upgrades.

It is up to the user application to implement a mechanism to remotely connect to the host and download image assets to the flash storage. This section only details the interface of the OTA application to the multiboot & the OTA procedure itself.

When enabled, Multiboot checks for availability of a new image via OTA application (e.g. by checking the value at a well-defined location in the flash. Exact mechanism for this notification is customizable). The data structure used to communicate information between the OTA application and the multiboot resembles closely with the content of the **NEW_IMAGE** command described previously.

```
#define OTA_INFO_OPTIONS_EXT_FLASH  0x1
#define OTA_INFO_MAGIC_NUM          0xDEADCAFE
typedef struct
{
    // Should be set to OTA_INFO_MAGIC_NUM
    uint32_t    magicNum;
    // Address in flash where the new image should be programmed
    uint32_t    *pui32LinkAddress;
    // Length of image blob
    uint32_t    ui32NumBytes;
    // CRC of the image blob
    uint32_t    ui32ImageCrc;
    // (Optional) Security Info length
    uint32_t    secInfoLen;
    // Options - e.g. Read from external flash device
    uint32_t    ui32Options;
    // (optional) Security Information location
    uint32_t    *pui32SecInfoPtr;
    // Location of image blob - Address needs to be aligned to 4 Byte address
    uint32_t    *pui32ImageAddr;
    // CRC to confirm integrity of the OTA Descriptor structure
    uint32_t    ui32Crc;
} am_multiboot_ota_t;
```

In order to support devices with limited free flash space, the OTA upgrade mechanism also supports reading the image from external flash by providing customization hooks to enable accessing a third party flash device during the upgrade process.

Multiboot validates the OTA image and installs the same similar to the wired updates through a host.

If enabled for secure boot, the OTA upgrade process calls appropriate security function hooks through the upgrade process to enable implementation of security features.

SECTION

8

Multiboot Customization

Multi-Protocol Bootloader supplied as part of AmbiqSuite is designed for flexibility. The core protocol functionality is implemented as a set of helper functions part of core AmbiqSuite, and it is expected that individual deployments would implement their own bootloader application using these core functions.

Standard AmbiqSuite SDK contains sample example projects for both the secure and non-secure version of bootloader.

> **NOTE:** Current implementation of multiboot helper functions assume that all the SRAM and Flash in the chip has been powered up, even if the program needs to use only a part of it. For flash, this is needed to allow a new image to be flashed in at any location in the flash. The SRAM area beyond the region used for the program (as indicated by **MAX_SRAM_USED** below), is used as a scratch space by multiboot. The user application needs to make sure that unused flash and SRAM regions are not powered off.

8.1 Compile Time Constants

These constants can be overridden by a particular bootloader implementation for multiboot customization.

```
//
//*****************************************************************************
//
// Secure Boot.
//
//
//*****************************************************************************
#define MULTIBOOT_SECURE
//
//*****************************************************************************
//
// Run with flag page.
//
//
//*****************************************************************************
#define USE_FLAG_PAGE 1
//
//*****************************************************************************
//
// Location of the flag page.
//
//
//*****************************************************************************
#define FLAG_PAGE_LOCATION              0x00006000
//
//*****************************************************************************
//
// Max Size of Bootloader.
//
//
//*****************************************************************************
// The value here must match (at least) with the ROLength restriction imposed
at
// bootloader linker configuration
#define MAX_BOOTLOADER_SIZE             0x00006000
// The value here must match (at least) with the RWLength restriction imposed
at
// bootloader linker configuration
#define MAX_SRAM_USED                   0x00008000
//
//*****************************************************************************
//
// I2C Address to use
//
//
//*****************************************************************************
#define I2C_SLAVE_ADDR                  0x10
```

8.2 Helper Functions

8.2.1 Check for Override Status and Flash Image Validity

Bootloader implementation should call this function at the beginning.

```
bool
am_multiboot_check_boot_from_flash(bool *pbOverride,
am_bootloader_image_t **ppsImage)
```

This function checks the flag page (if enabled) and verifies the flash image for integrity. It also checks for the override pin status in case forced host boot is requested. Returns **true**, if it is okay to boot from the image in flash, and passes the Image structure back to the caller.

Table 8-1: Check for Override Status and Flash Image Parameter Description

Parameter	Description
`pbOverride`	return parameter, used to pass back the override status
`ppsImage`	return parameter, used to pass back the image structure

8.2.2 Run Image

Bootloader implementation should call this function to run the image as per the image information provided.

```
void
am_bootloader_image_run(am_bootloader_image_t *psImage)
```

Table 8-2: Run Image Parameter Description

Parameter	Description
`psImage`	Pointer to the image structure

This function can be used to **run** a program that has already been downloaded and written to flash. It does this by reading the initial stack-pointer and reset vector information from the image written in flash, writing that information to the relevant registers, and immediately branching to the new reset vector location.

Note that this method does not include any type of reset. It is the caller's responsibility to ensure that the MCU is in a valid state for the subsequent program to run. One way to guarantee this is to run this function very early after a **RESET** event, before clocks or peripherals are configured.

8.3 Image Upgrade Over IOS/UART

8.3.1 Initialize Multiboot

This function should be called to provide multiboot with the scratch memory in SRAM.

```
bool
am_multiboot_init(uint32_t *pBuf, uint32_t bufSize)
```

Table 8-3: Initialize Multiboot Parameter Description

Parameter	Description
pBuf	This is the temporary buffer for multiboot to operate on.
bufSize	Temporary buffer size. This should be at least equal to the AM_HAL_-FLASH_PAGE_SIZE

This function should be called for both IOS or UART based image upgrade. The function returns failure if the buffer provided is not sufficient.

8.3.2 Upgrade Over IOS

8.3.2.1 Setup IOS Interface

This function should be called to set up the IOS interface for connecting with host.

```
void
am_multiboot_setup_ios_interface(uint32_t interruptPin);
```

Table 8-4: Setup IOS Interface Parameter Description

Parameter	Description
interruptPin	Handshake pin to be used for interrupting host

This function auto-detects for SPI or I^2C based on the clock pin signal level.

8.3.2.2 Cleanup IOS Interface

In case of bootloader enabling multiple interfaces to boot from (e.g., UART), this function is used to clean up the IOS configuration before switching to alternate interfaces.

```
void
am_multiboot_cleanup_ios_interface(void)
```

This function un-configures the IOS interface.

8.3.2.3 IOS Interrupt Handler

This function implements the core multiboot data transfer & control protocol over the IOS interface. Bootloader implementation should call this function from the IOS ACC interrupt handler.

```
void
am_multiboot_ios_acc_isr_handler(void);
```

8.3.3 Image Upgrade Over UART

8.3.3.1 AutoDetect Baudrate

If desired, this helper function can be used to autodetect the baud rate host is using. When host first connects to the slave, it is expected to send one byte 0x55 to the slave. Slave uses this known pattern to auto-detect the UART baudrate.

```
uint32_t
am_multiboot_uart_detect_baudrate(uint32_t ui32RxPin);
```

This function returns the detected baudrate.

8.3.3.2 Setup Serial Interface

This function should be called to set up the UART interface for connecting with host.

```
void
am_multiboot_setup_serial(int32_t i32Module, uint32_t ui32BaudRate);
```

Table 8-5: Setup Serial Interface Parameter Description

Parameter	Description
`i32Module`	UART module to use
`ui32BaudRate`	Desired baudrate

8.3.3.3 UART Interrupt Handler

This function implements the core multiboot data transfer & control protocol over the UART interface. Bootloader implementation should call this function from the respective UART interrupt handler.

```
void
am_multiboot_uart_isr_handler(uint32_t ui32Module);
```

Table 8-6: UART Interrupt Handler Parameter Description

Parameter	Description
i32Module	UART module to use

8.4 OTA Image Upgrade

8.4.1 External Flash Access Hooks

This structure is used by the caller to provide core multiboot functions means to access external flash, if desired for OTA upgrade.

```
// All functions return 0 on success
typedef int (*flash_read_func_t)(uint32_t ui32DestAddr, uint32_t *pSrc, uint32_t
ui32Length);
typedef int (*flash_write_func_t)(uint32_t ui32DestAddr, uint32_t *pSrc, uint32_t
ui32Length);
typedef int (*flash_erase_func_t)(uint32_t ui32Addr);
typedef int (*flash_init_func_t)(void);
typedef int (*flash_deinit_func_t)(void);
typedef int (*flash_enable_func_t)(void);
typedef int (*flash_disable_func_t)(void);

typedef struct
{
     // Minimum Granularity for Write
     // Should be power of 2
     uint32_t               flashPageSize;
     // Minimum Granularity for Erase
     // Should be power of 2
     uint32_t               flashSectorSize;
     // Initialize the flash device
     flash_init_func_t      flash_init;
     // De-Initialize the flash device
     flash_deinit_func_t    flash_deinit;
     // Enable (Power up) the flash device
     flash_enable_func_t    flash_enable;
     // Disable (Put in Low power mode) the flash device
     flash_disable_func_t   flash_disable;
     // Read a block of data from within a flash page
     flash_read_func_t      flash_read_page;
     // Read a block of data within a flash page
     flash_write_func_t     flash_write_page;
     // Erase the flash sector corresponding to address specified
     flash_erase_func_t     flash_erase_sector;
} am_multiboot_flash_info_t;

typedef void (*invalidate_ota_func_t)(am_multiboot_ota_t *pOtaInfo);
```

8.4.2 OTA Upgrade Helper Function

This function validates the OTA blob, and installs the image if verified. It updates the flag page with the new image information and issues a POI.

Bootloader should call this function on detecting a valid OTA upgrade image.

```
bool
am_multiboot_ota_handler(am_multiboot_ota_t *pOtaInfo,
       uint32_t *pTempBuf, uint32_t tmpBufSize,
       invalidate_ota_func_t invalidateOtaFunc,
       am_multiboot_flash_info_t *pExtFlash);
```

The function return false if OTA upgrade fails. Otherwise this function does not return.

Table 8-7: OTA Upgrade Helper Function Parameter Description

Parameter	Description
pOtaInfo	Pointer to OTA descriptor with image information
pTempBuf	Pointer to a temporary buffer. This buffer should be sized to minimum one flash sector (bigger of internal and external flash page, if ext flash is being used)
tmpBufSize	Size of the temporary buffer
invalidateOtaFunc	Pointer to function called to invalidate the OTA for subsequent boots
pExtFlash	Pointer external flash access info, if needed

x

x

x

For production devices, it is likely that there would be need to support image upgrades without requiring direct physical connection to host. Multi-Protocol Bootloader can be built to support OTA upgrades.

It is up to the user application to implement a mechanism to remotely connect to the host and download image assets to the flash storage. This section only details the interface of the OTA application to the multiboot & the OTA procedure itself.

When enabled, Multiboot checks for availability of a new image via OTA application (e.g. by checking the value at a well-defined location in the flash. Exact mechanism for this notification is customizable). The data structure used to communicate information between the OTA application and the multiboot resembles closely with the content of the **NEW_IMAGE** command described previously.

In order to support devices with limited free flash space, the OTA upgrade mechanism also supports reading the image from external flash by providing customization hooks to enable accessing a third party flash device during the upgrade process.

Multiboot validates the OTA image and installs the same similar to the wired updates through a host.

If enabled for secure boot, the OTA upgrade process calls appropriate security function hooks through the upgrade process to enable implementation of security features.

SECTION

9

Appendix

9.1 Downloading a New Image Using Multi-Protocol Bootloader Supplied with AmbiqSuite

AmbiqSuite includes a Multi-Protocol bootloader example, which can be customized to suit specific needs to implement a bootloader for final project.

This section details how to use the supplied Multi-Protocol Bootloader.

9.1.1 Multi-Boot

This is the bootloader to be flashed on to the target board at address 0x0

- It checks for a valid image in the flash, and if not found (or when using optional flag page, if the checksum validation fails) waits for host to download a new image
  - It can be forced to upgrade an image even if a valid image is present by using BOOT-OVERRIDE pin
- When doing the image download, it first checks for image download through I^2C/SPI (it auto-detects the mode based on level setting of the clock pin)
- If host does not send image on I^2C/SPI, it then switches to UART after a brief wait

Once the device has multi-boot bootloader flashed in, the main firmware image can be upgraded using an external host, connected to the device using I^2C/SPI or UART.

9.1.2 Image Upgrade Over I²C/SPI Using Another Ambiq Evaluation Board

- AmbiqSuite supplies example programs - **i2c_boot_host** and **spi_boot_host** respectively, which can be run on a separate board acting as a host to download a new image to the target using I²C/SPI.
  - Both these programs have been compiled with pre-built binary image (converted to a header file **apollo*_boot_demo.h**) representing the example **binary_counter**
  - When using optional pins to drive the RESET and OVERRIDE pins, the host can force a new image download to the target
- Pin Fly lead assumptions for default programs

```
//! PIN fly lead connections assumed by multi_boot:
//! HOST                                    SLAVE (multi_boot target)
//! --------                                --------
//! GPIO[2]  GPIO Interrupt (slave to host) GPIO[4] GPIO interrupt
//! GPIO[4]  OVERRIDE pin (host to slave)   GPIO[18] Override pin or n/c
//! GPIO[5]  IOM0 SPI CLK/I2C SCL           GPIO[0] IOS SPI SCK/I2C SCL
//! GPIO[6]  IOM0 SPI MISO/I2C SDA          GPIO[1] IOS SPI MISO/I2C SDA
//! GPIO[7]  IOM0 SPI MOSI                  GPIO[2] IOS SPI MOSI
//! GPIO[11] IOM0 SPI nCE                   GPIO[3] IOS SPI nCE
//! GPIO[17] Slave reset (host to slave)    Reset Pin or n/c
//! Reset and Override pin connections from Host are optional
//! Keeping Button1 pressed on multi-boot target has same effect as host
driving
//! override
```

- Creating a header file from a binary image to be flashed
  - The bin file suitable for download like this needs to be created with start address 0x8000 or higher (Set ROBase to 0x8000 or higher in linker file).
  - Use the AmbiqSuite script **pack_for_boot.py** to generate a header file corresponding to the image
    - **pack_for_boot.py -o <header file name> -l <link address> <bin file>**
    - Here bin file is the desired binary firmware image file (needs to be built with link address 0x8000 or higher)
    - Link Address is a hex number indicating the link address used for the image file
  - Once generated, it can be edited to match the requirements of **apollo*_boot_demo.h**
- The host programs **i2c_boot_host** and **spi_boot_host** can be rebuilt with the new header files as replacements.

9.1.3 Image Upgrade Over UART

When using UART boot, a command line script included in AmbiqSuite can be used to accomplish the host side functionality.

Note that this script relies on add-on packages for python, namely **serial** and **pycryptodome** that needs to be installed using **pip install <packagename>**

```
tools\bootloader_scripts\uart_boot_host.py <bin file> <link address> COM??
```

- Here **bin file** is the desired binary firmware image file (needs to be built with link address 0x8000 or higher)
- **link Address** is a hex number indicating the link address used for the image file
- Replace "COM??" with the COM port on the host machine connected to the UART on the device.

9.1.4 Secure Boot

AmbiqSuite also provides a sample implementation of the secure version of Multi-Boot with provisions for Image confidentiality and Authentication and Integrity Verification along with other optional security features.

Currently AmbiqSuite provides mechanisms to exercise the secure boot over UART only.

When using the secure boot, a set of command line scripts included in AmbiqSuite can be used to accomplish the host side functionality.

Secure implementation of multi-boot is designed to be customized to suit individual deployment requirements. AmbiqSuite SDK contains a sample secure boot implementation to demonstrate the concept, and does not implement cryptographically secure methods. Even though the overall image upgrade procedure would follow the outline described below, the generation of secure boot assets need to be customized in accordance with the specific secure boot implementation.

- As with non-secure boot case, the desired image file needs to be built with link address 0x8000 or higher
- There is one extra step to generate encrypted image and the security trailer. **generate_secureboot_assets.py** can be found in the **multi_boot_secure** project.

```
.\ generate_secureboot_assets.py <bin file> <keyidx> <protection>
<encimagefilename> <sectrailerfilename>
```

  - **keyidx** identifies the key to use. The script should be pre-populated with the keys matching the bootloader implementation.
    - The sample secure implementation uses 8 4-byte keys, and hence this can be any value 0,1,2,3,4,5,6,7
    - The sample secure bootloader implementation marks the key 6 as "revoked" as illustration
  - **protection** – designed to provide image protection requirement (Write-Protect, Copy-Protect). Currently this is not implemented, and hence only 0 is a valid value
  - This generates two binary files – "encrypted image file" and "security trailer" as needed for the script **uart_boot_host.py** for the second step.

- The same **uart_boot_host.py** script can be used to interface with the secure multi-boot using additional parameters

```
tools\bootloader_scripts\uart_boot_host.py <encimagefilename>
<link address> COM?? –s <sectrailerfilename>
```

  - **encimagefilename** and **sectrailerfilename** represent the secureboot assets generated in the previous step

6500 River Place Boulevard, Building 7, Suite 200, Austin, TX 78730
www.ambiq.com
sales@ambiq.com
+1 (512) 879-2850

A-SOCAP3-ANGA02EN v1.1
December 2022