User's Guide

OTA Example Project

Date	Revision History	Reviser
2016-12-20	V0.1 draft created	Renton Ma
2016-12-30	V0.2 added ota profile and app descriptions.	Renton Ma

Table of Contents

1. Introduction

This document describes the firmware Over-The-Air (OTA) update example using BLE 4.x for APOLLO MCU as well as the firmware running inside of the HCI controller. The example project consists of the following parts to complete the function:

- Program running on the APOLLO MCU.
  - Bootloader (multi_boot)
  - OTA BLE Service (amota)
  - BLE stack (ARM Cordio BTLE Stack)
- Firmware running on BLE HCI controller device (DA14581)
- Smartphone APP on iOS system. (OTA Demo)
- Scripts to generate binary files for APP to load.

The example is built based on the existing exactle_fit example of the Ambiqsuite SDK for the APOLLO MCU. The purpose of the example is to provide a reference for firmware update of the MCU and the HCI controller over BLE communication while the application is still running. Data transfer is in background operation of the application and can be paused and resumed during the progress. Data being transferred is verified by each communication package as well as a whole image once the transfer is complete. Data received can be stored either inside the empty area of the internal flash (if there is enough space left in the internal flash of the MCU) or into the external serial flash. After the complete image is received and stored correctly, the system will keep operating from the existing firmware until a system reset is triggered. The new image will be loaded into the target internal flash by the bootloader after a system reset, and executed automatically if checked available.

2. System Description

This section of the document describes the system of the OTA example project in general.

The example is developed, compiled and tested with Keil ARM compiler V5.06 update 2 (build 183).

2.1 Features

- Functions during OTA
  - Robust to communication disturbances.
    Continue update progress after re-connection.
  - Side-by-side firmware update.
    The system update progress is executed while application is running, providing a “silent update” experience to the end user.
- Proprietary OTA service on top of BLE stack.
  - RF IC always works in HCI controller mode and does not require image switching.
- High speed communication
  - Firmware data transfer at a speed of > 5KB / sec (target) with iOS and Android.
    (First target: iOS 10 and Android 6.0)
- Error handling
  - CRC checking is applied to each data packet received as well as to the whole image after the completion of the data transfer.
  - Packet will be requested to be re-transmitted by the central device (smartphone) if there is error happened during the communication.
- Firmware version information is transmitted and stored.
- Image storage
  - Image can be selected to be stored either in internal flash or external storage, enabling larger image size to be updated.
- Update of the BLE stack itself
  - Since the image is transferred as one entity containing the BLE stack as well as the OTA service itself, this OTA update flow allows user to update the BLE stack as well as the OTA service to be updated.
- Encrypted image data (extended feature)
  - Image being transferred and stored can be encrypted and decrypted during the boot process.
- Data type can be specified (extended feature)
  - Type of the data being transferred and stored can be specified as application firmware or plain data providing the possibility to update only the data arrays inside the firmware without updating the rest of the code.

(Extended features are not included in version 1.0 of the example project.)

2.2 System Architecture and operation flow

A high-level system architecture and operation flow is illustrated in the diagram below:

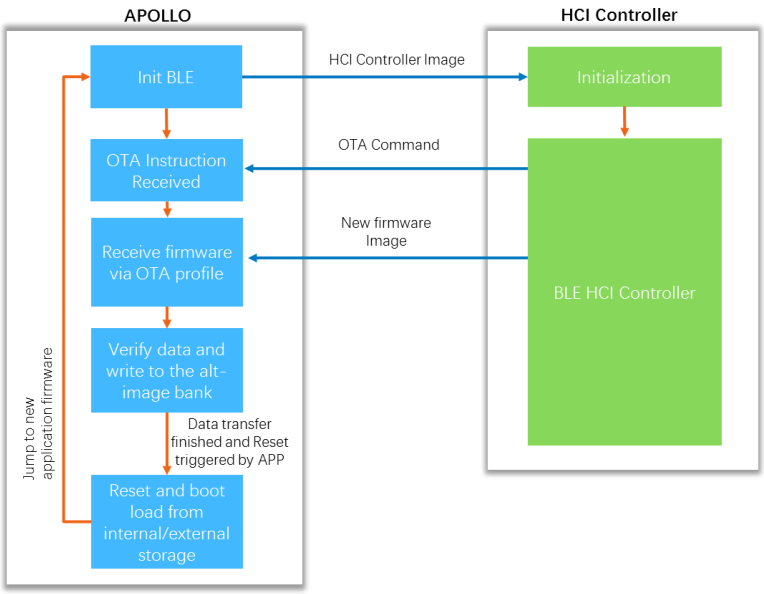


FIGURE 1 SYSTEM ARCHITECTURE AND OPERATION FLOW

The BLE stack used in this example is the ARM Cordio BTLE stack (a.k.a Wicentric BLE Stack, or exactLE Stack) software version 1.3. The upper layers (above and including HCI host layer) of the stack is running on the APOLLO MCU as a part of the application firmware leaving the RF IC working as a standard HCI controller. This architecture utilizes the ultra-low operating power feature of the APOLLO MCU for the BLE communication as much as possible to further reduce overall system power consumption, as well as providing a flexibility of choosing the external RF controller.

As the application firmware gets boot up, the lower level software driver will send the standard HCI controller image to the RF controller, and the BLE communication can be started once HCI controller and the stack are initialized.

Since the stack and the HCI controller static image are parts of the application firmware, they can be updated during the OTA progress.

For further information of the ARM Cordio BTLE, please refer to the documents located in the directory of ..\AmbiqMicro\AmbiqSuite\third_party\exactle\docs\pdf .

The example starts to broadcast with standard services (heart rate, device information and battery) once get loaded. When a central device connects to the example device, AMOTA service can be discovered, and data transfer can be triggered according to the pre-defined meta data description. For more information of the AMOTA service, please refer to 2.5 AMOTA Service of this document.

During data transfer, application code can keep running. This example uses a binary counter that turns on and off the LED arrays on the APOLLO MCU EVB to indicate the operating status of the application code. Data received is stored into internal flash or external serial flash according to the user specification.

Once the data transfer is completed, the AMOTA service will update the flash flag page located at a fixed memory address (0x3C00) to mark a valid new image is available for the bootloader to load. User can make the choice of whether to trigger a POI reset to the MCU or keep running the old application code.

After a system reset, bootloader checks the flash flag page information and loads the available new image into the target memory address. Once verification passes, the new image gets executed from the bootloader. For details about bootloader, please check section 2.4 Bootloader.

2.3 MCU Memory Map

The MCU memory map is shown in the figure below:

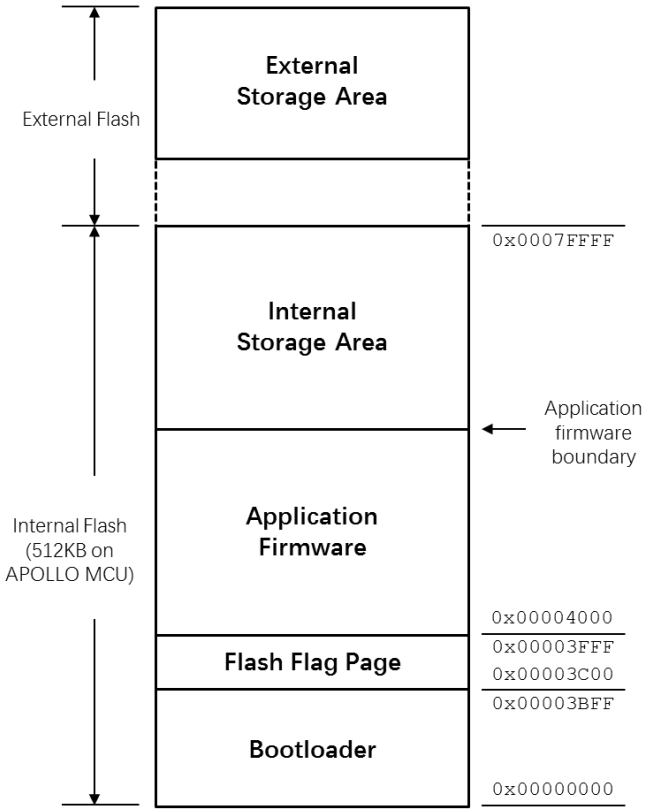


FIGURE 2 MCU MEMORY MAP

The first 16K bytes of the internal flash is mapped to the bootloader (15K bytes) and flash flag page (1K bytes). By default, flash flag page is fixed starting from 0x3C00. Application firmware can be mapped right after the flash flag page starting from 0x4000 (default), or any other address above. Application firmware boundary is the end of the application code aligned with the internal flash page size (which is 2048 bytes for APOLLO MCU). If the data received is to be stored inside the internal flash, user has to ensure that the space left inside the internal flash starting from the application firmware boundary is sufficient to hold the data. If not, it is recommended to store the data received inside the external storage device.

Checking the space left inside the internal flash is done by API provided along with the example project.

2.4 Bootloader and Flash Flag Page

2.4.1 Bootloader

Bootloader of this example is built based on multi_boot example of the SDK. For more information of multi_boot, please check MultiProtocolBootloader.pdf in the folder of ..\AmbiqMicro\AmbiqSuite\docs\app_notes\bootloader.

This section only describes the modification made on multi_boot as well as the extended flash flag page settings.

The modified boot sequence is shown in the figure below:

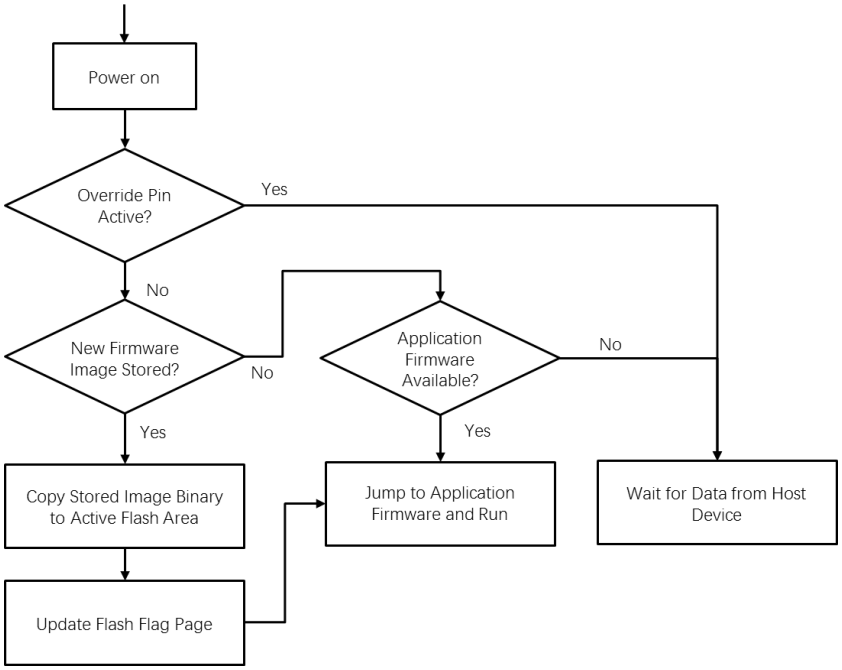


FIGURE 3 BOOT SEQUENCE

After checking the override pin, the bootloader checks the ui32Options flag stored in flash flag page (address offset 0x1C) to determine whether there is a new image stored in internal or external flash. If there is, the bootloader will first check the availability of the stored image by calculate and compare the CRC of the stored data with the CRC inside the flash flag page (ui32CRCNewImage, address offset 0x34), and load the image into the target link address (pui32LinkAddress, address offset 0x00) if the image is valid.

Bootloader is a standalone project besides the application.

It is located at ..\Ambiqsuite\boards\apollo_evk_base\examples\multi_boot.

2.4.2 Flash Flag Page

This example project utilizes a modified flash flag page data structure which is shown in the table below:

Symbol	Length (bytes)	Address Offset	Description
pui32LinkAddress	4	0x00	Starting address where the image was linked to run. This value shall not be small than 0x4000.
ui32NumBytes	4	0x04	Length of the executable image in bytes.
ui32CRC	4	0x08	CRC-32 Value for the full image.
ui32OverrideGPIO	4	0x0C	Override GPIO number. Can be used to force a new image load.
ui32OverridePolarity	4	0x10	Active polarity for the override pin. 0: Logic low; 1: Logic high If the selected GPIO input value matches active polarity, bootloader is forced to load new image from external host via serial communication.
pui32StackPointer	4	0x14	Stack pointer location. This value shall not be small than starting address of the internal SRAM.
pui32ResetVector	4	0x18	Reset vector location. This value shall not be smaller than pui32LinkAddress.
ui32Options	4	0x1C	Boot Options. 0x01: New image available in internal flash. 0x02: New image available in external flash. Other: No new image available.
ui32Version	4	0x20	Version Informatin of the Current Image
ui32VersionNewImage	4	0x24	Version Informatin of the New Image
pui32StorageAddressNewImage	4	0x28	Starting address where the new image was stored.
ui32TotalNumBytesNewImage	4	0x2C	Length of the new image being received in bytes.
ui32StoredNumBytesNewImage	4	0x30	Bytes already received and stored. (Not used in this example.)
ui32CRCNewImage	4	0x34	CRC-32 Value for the new image being received.
bEncrypted	4	0x38	Use to determine whether the image is encrypted. 0: image is not encrypted. 1: image is encrypted.

TABLE 1 FLASH FLAG PAGE

2.5 AMOTA Service

The following section describes the AMOTA service that is implemented in this example to perform the key function of the OTA process.

2.5.1 Service Declaration

The service UUID of Ambiq Micro OTA (AMOTA) service is defined as below:

00002760-08C2-11E1-9073-0E8AC72E1001.

Note:

Base UUID of Bluetooth SIG is 00000000-0000-1000-8000-00805F9B34FB.

All customized 128-bit UUIS should be less than base UUID.

2.5.2 Service Characteristics Definitions

Rx : 00002760-08C2-11E1-9073-0E8AC72E0001

Tx : 00002760-08C2-11E1-9073-0E8AC72E0002

Characteristic	Requirements	Mandatory Properties	Security Permissions	Description
Characteristic Rx	M	Write	None	Data from client
Characteristic Rx User Description	N	Read	None	Value read by client
Characteristic Tx	M	Notify	None	Value notification to client
Characteristic Tx Client Characteristic Configuration descriptor	M	Read	None	Value notification configuration

TABLE 2 CHARACTERISTICS

2.5.3 Characteristics

The following characteristics are defined in the AM OTA Service. Only one instance of each characteristic is permitted within this service.

Characteristic Name	Mandatory Properties	Security Permission
Received data Characteristic	Write Command	None
Send data Characteristic	Notify	None

Characteristic Descriptors:

Characteristic User Description

This characteristic descriptor defines the AM OTA version with read permission property.

Client Characteristic Configuration Descriptor:

The notification characteristic will start to notify if the value of the CCCD (Client Characteristic Configuration Descriptor) is set to 0x0001 by client. The send data characteristic will stop notifying if the value of the CCCD is set to 0x0000 by client.

2.5.4 Service Behaviors

1. OTA client sends firmware header/meta info to server on amota packet format
2. Server replies with received byte counters
3. OTA client starts to send firmware data by amota packet format
4. Server replies with received byte counters
5. OTA client sends verify command to ask server to calculate the whole firmware checksum
6. Server replies checksum result to client
7. Client sends reset command to server (APP behavior)
8. Server sends reset command response to server before reset (APP behavior)

AMOTA packet format

Length : two bytes (data + checksum)

Cmd : 1 byte

Data : 0 ~ 512 bytes

Checksum : 4 bytes

Length	Command	Data	Checksum
Two bytes	1 byte	0 ~ 512 bytes	4 bytes

Commands:

```
/* amota commands */
typedef enum
{
    AMOTA_CMD_UNKNOWN,
    AMOTA_CMD_FW_HEADER,
    AMOTA_CMD_FW_DATA,
    AMOTA_CMD_FW_VERIFY,
    AMOTA_CMD_FW_RESET,
    AMOTA_CMD_MAX
}eAmotaCommand;
```

FW Header Info

Amota packet header (two bytes length + 1 byte cmd)
Version 4 bytes;
Firmware/Data size 4 bytes;
CRC32 4 bytes;
Start address in flash 4 bytes;
Byte counter for retransmitting 4 bytes;
Data Type 4 bytes;
Amota packet checksum (4 bytes)

FW Data Packet

Amota packet header (two bytes length + 1 byte cmd)
Data : 0 ~ 512 bytes
Amota packet checksum (4 bytes)

FW Verify Command

Amota packet header (two bytes length + 1 byte cmd)
Amota packet checksum (4 bytes)

Target Reset Command

Amota packet header (two bytes length + 1 byte cmd)
Amota packet checksum (4 bytes)

Command Response Format

Length : 2 bytes (data + status)

Cmd : 1 byte

Status : 1 byte

Data : 0 ~ 16 bytes

```
/* amota status */
typedef enum
{
    AMOTA_STATUS_SUCCESS,
    AMOTA_STATUS_CRC_ERROR,
    AMOTA_STATUS_INVALID_HEADER_INFO,
    AMOTA_STATUS_INVALID_PKT_LENGTH,
    AMOTA_STATUS_INSUFFICIENT_BUFFER,
    AMOTA_STATUS_UNKNOWN_ERROR,
    AMOTA_STATUS_MAX
}eAmotaStatus;
```

FW Header Info Response & FW Data Response

Amota packet header (two bytes length + 1 byte cmd)

Status : 1 byte

Received packet counter : 4 bytes

FW Verify & Target Reset Response

Amota packet header (two bytes length + 1 byte cmd)

Status : 1 byte

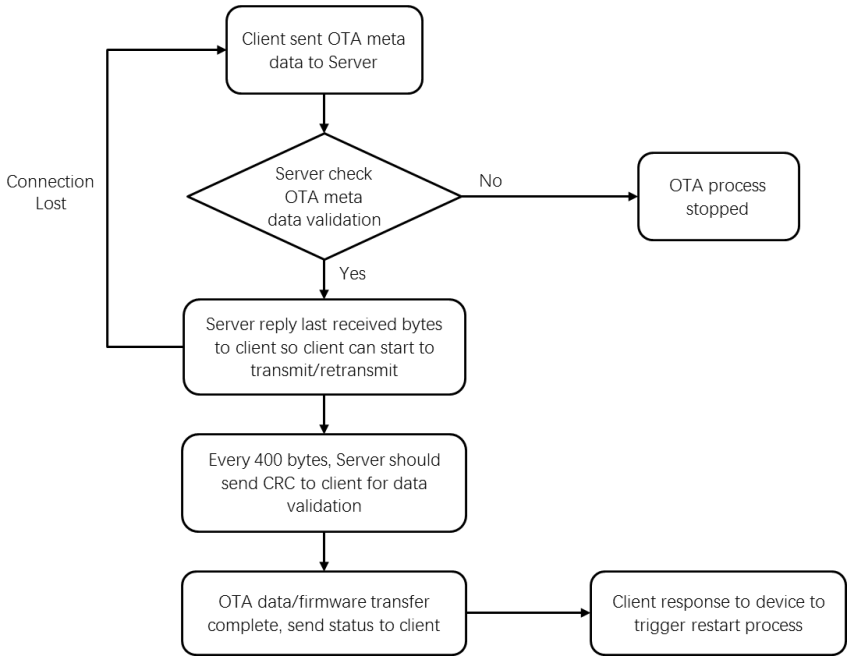


FIGURE 4 OTA SERVICE FLOW

- Create services
  svc_amotas.c
  svc_amotas.h
- Profile and OTA logic implementation
  amotas_main.c
  amotas_api.h
- Initialize in application
  Set AmotasCfg_t
  Add AMOTAS_TX_CH_CCC_HDL in attsCccSet_t
  Register callback in FitStart()
    SvcAmotasCbackRegister(NULL, amotas_write_cback);
  Add service in FitStart()
    SvcAmotasAddGroup();
  Call amotas_proc_msg() in fitProcMsg() for event DM_CONN_OPEN_IND
  Add amotas_start() and amotas_stop() in function fitProcCccState()

3. Getting Started

3.1 Folder directory

The example project comes with the following folder structure:

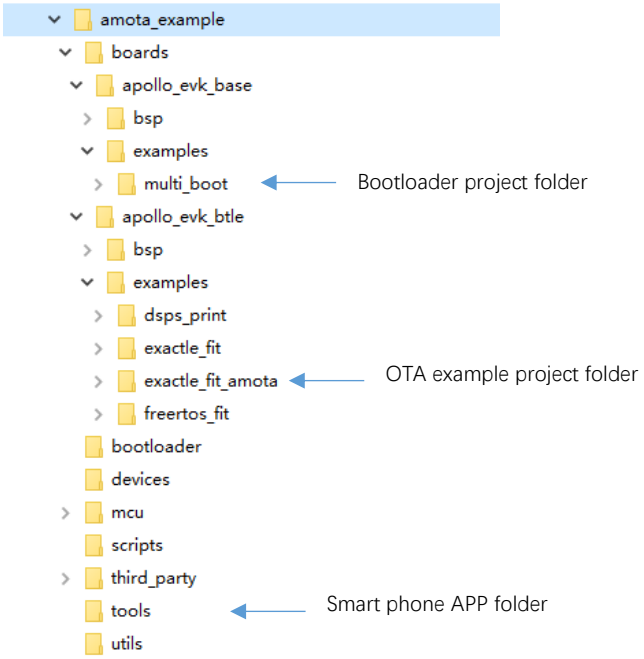


FIGURE 5 FOLDER STRUCTURE

3.2 Development Environment

- Hardware:

  This example runs on APOLLO MCU EVK (base board + BTLE board), make sure you have one available to run the example. Otherwise, modifications need to be done according to the hardware setup in the BSP of both multi_boot and exactle_fit_amota folders.

- Software:

  Install the latest AmbiqControlCenter from the following link and make sure the OpenOCD driver is installed properly following the guide in AmbiqControlCenter.

  Link: http://ambiqmicro.com/support

  Install Python 3.x to run the helper scripts for OTA binary file generation and combination.

  Install Keil MDK-ARM Plus Version 5.20 or later for code generation and debug.

  Install iTools PC tool for iOS device APP(ipa) installation.

  Link: http://www.itools.cn/

3.3 Run the example

To run the example, follow the steps listed below:

1. Run the bootloader_binary_combiner.py from a command line window in the folder of ..\boards\apollo_evk_btle\examples\exactle_fit_amota\keil.
   Python bootloader_binary_combiner.py --appbin bin/exactle_fit_amota.bin -o single_binary

```
$ python bootloader_binary_combiner.py --appbin bin/exactle_fit_amota.bin -o single_binary
boot size  9952
pad_length  6432
load_address  0x4000 ( 16384 )
app_size  0x11f44 ( 73540 )
crc =   0x25e33b2c
```

   This will generate a binary file called single_binary.bin, this binary is the combination of the bootloader binary (multi_boot.bin), the flash flag page options (default values) and exactle_fit_amota.bin (as the application firmware).

2. Modify the application firmware to make a new version.
   Open exactle_fit_amota project with KEIL, comment out line 457 and 458, and un-comment line 461 and 462.
   Build the project.

3. Generate the OTA binary file using the new binary just built.
   The OTA binary file is a file with the pre-defined OTA header on top of the binary data. Run the ota_binary_converter.py script to generate this file (ota_test_binary.bin):

```
$ python ota_binary_converter.py -o ota_test_binary
pad_length  48
load_address  0x4000 ( 0x4000 )
app_size  0x11f44 ( 73540 )
crc =   0x25e33b2c
app_ver 0 ( 0x0 )
bin_type 0 ( 0x0 )
```

4. Load the single_binary.bin into the target MCU.
   This can be done with AM FLASH as shown in the screenshot below.

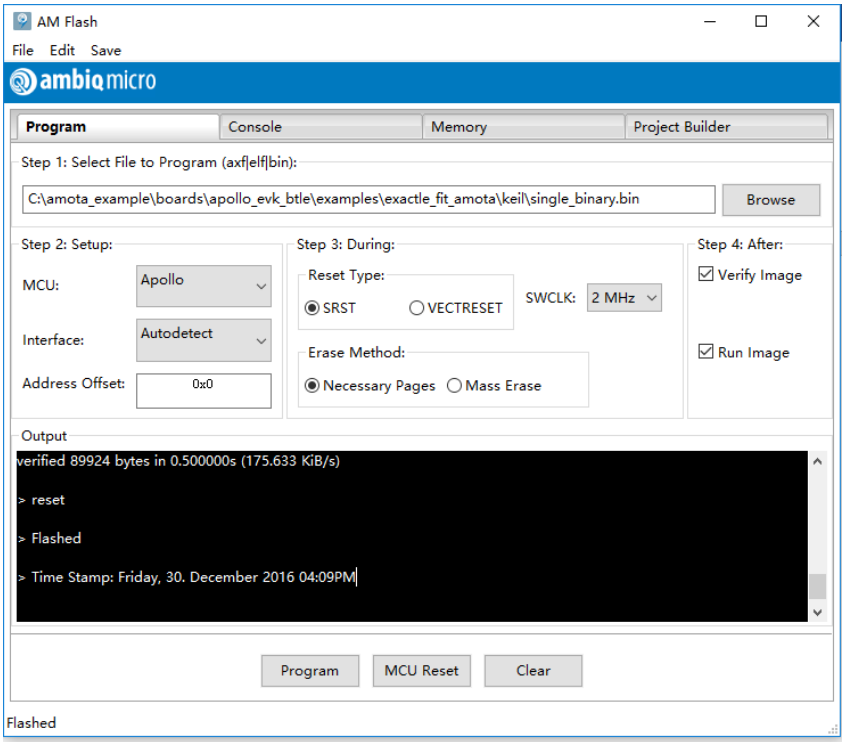
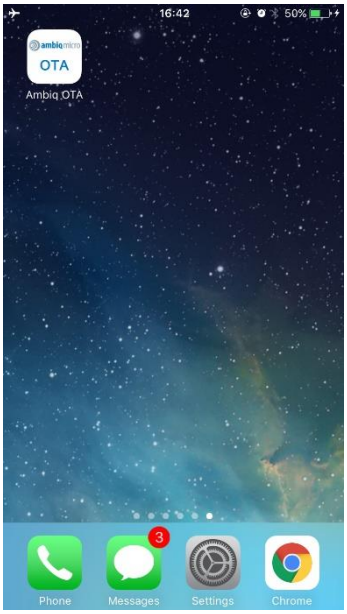


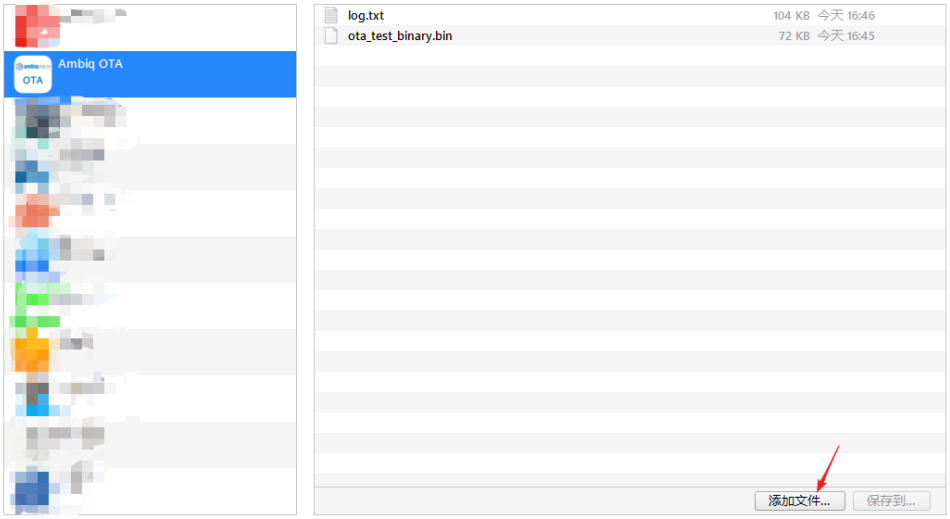
FIGURE 6 AM FLASH SCREENSHOT

After the binary file is flashed into the target device, LED arrays shall be blinking as binary counter with an adding pattern.

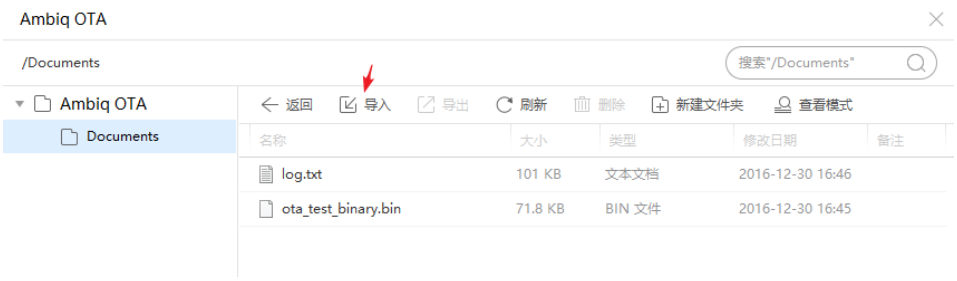
5. Install the iOS APP to the smart phone using iTools or whatever means.
   Once the APP is installed successfully, it will be shown on the home screen.



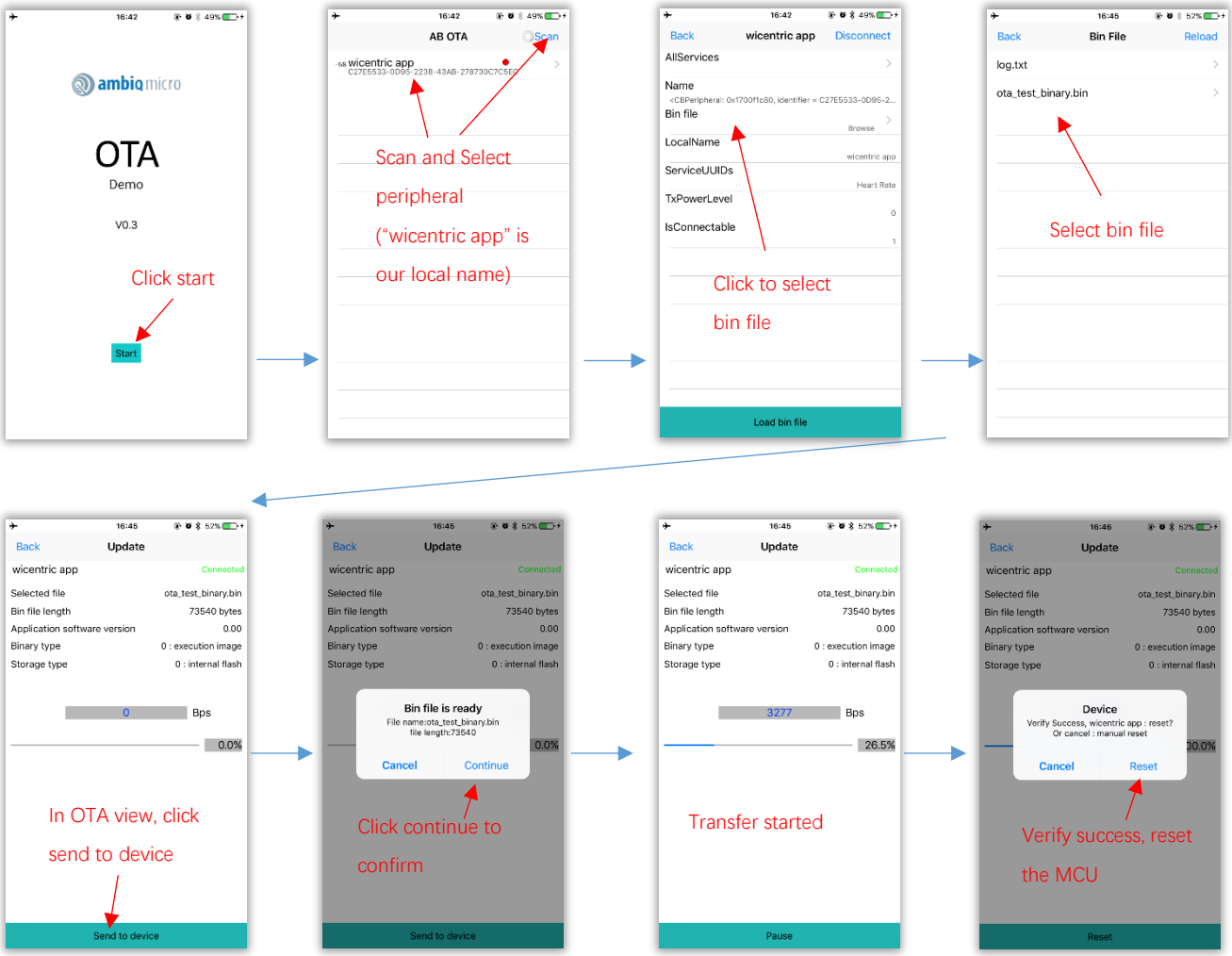
6. Load the ota_test_binary.bin file into the APP.
   Connect the smart phone with PC, start iTunes or iTools and load the binary file into OTA APP.
   With iTunes:



   With iTools:



7. Start OTA APP from smart phone and send the ota_test_binary.bin via BLE.



Once reset command is sent to MCU, the MCU resets and enters bootloader, it will take several seconds to load the new image into the flash.

After the new image is up and running, the LED array will blink the binary counter with a sub pattern, which indicates the operation is successfully done.

4. Characteristics

	Item	Typ.	Unit	Remark
Data Transfer speed				
	Storage in internal flash	3.5	KB/s	
	Storage in external flash	3.4	KB/s	
Resource Consumption				
	Bootloader code size	9.72	KB	
	Bootloader RAM size	5.32	KB	
	Flash flag page	1	KB	
	OTA project code size	71.82	KB	
	OTA project RAM size	6.2	KB	Including 1KB of heap.
Execution Timing				
	Boot from internal flash	TBD	sec	71.82KB image
	Boot from external flash	TBD	sec	71.82KB image
	Flash write to internal flash	TBD	sec	512bytes
	Flash write to external flash	TBD	sec	256bytes