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
2017-1-12	V0.3 add storage type definition	Mike Li
2017-2-17	V0.4 added Apollo2 support description	Renton Ma
2017-3-24	V0.5 updated according to setting changes	Renton Ma
2017-5-5	V1.0 created for AmbiqSuite SDK Rel 1.2.8	David Munsinger
2017-11-27	V1.1 created for AmbiqSuite SDK Rel 1.2.11	Zhongyi Chen

Table of Contents

1. Introduction

This document describes the firmware Over-The-Air (OTA) update example using BLE 4.2 for APOLLO[1] and APOLLO2 series MCUs 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/APOLLO2 series MCUs.
  - AMOTA Application (freertos_amota)
  - Bootloader (exactle_fit_amota_multi_boot)
  - OTA BLE Service (amota)
  - BLE stack (ARM Cordio BTLE Stack)
- Firmware running on BLE HCI controller device (EM9304)
- Smartphone APP on iOS or Android system. (OTA Demo)
- Makefile to generate binary files for APP to load.

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.

[1] Note: Apollo1 support will be provided as soon as EM9304 device is supported with the apollo1_evb.

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 V5.17 and and Eclipse Mars 1 Release.

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 > 3KB / sec with iOS and Android.
    (Target version: 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:

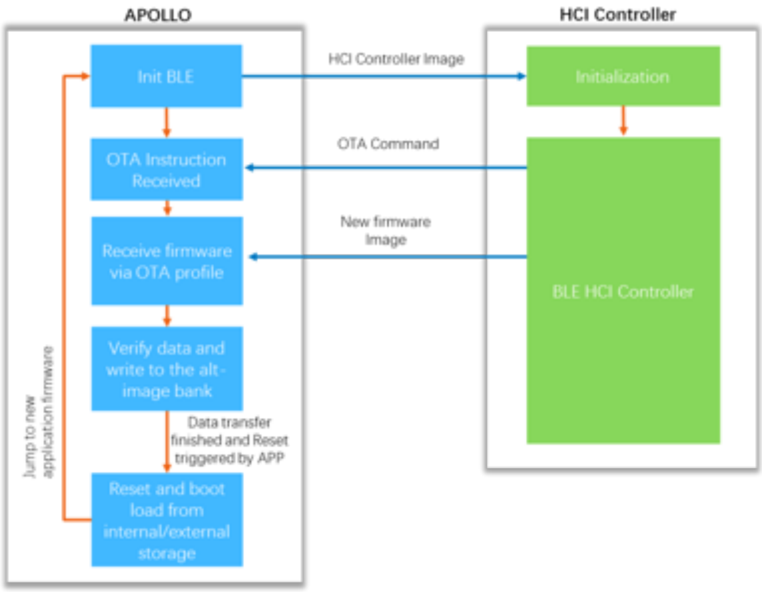


FIGURE 1 SYSTEM ARCHITECTURE AND OPERATION FLOW

The BLE stack used in this example is the ARM Cordio BTLE stack (a.k.a Wicentric BLE Stack, or exactLE Stack) software version 2.1. The upper layers (above and including HCI host layer) of the stack is running on the APOLLO/APOLLO2 series MCUs as a part of the application firmware. This leaves the BTLE module working as a standard HCI controller. This architecture utilizes the ultra-low operating power feature of the APOLLO/APOLLO2 series MCUs 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 boots 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 on the ARM Cordio BTLE stack, 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 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/APOLLO2 series MCUs 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 (default: 0x3C00 for APOLLO MCU and 0x4000 for APOLLO2 MCU) 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:

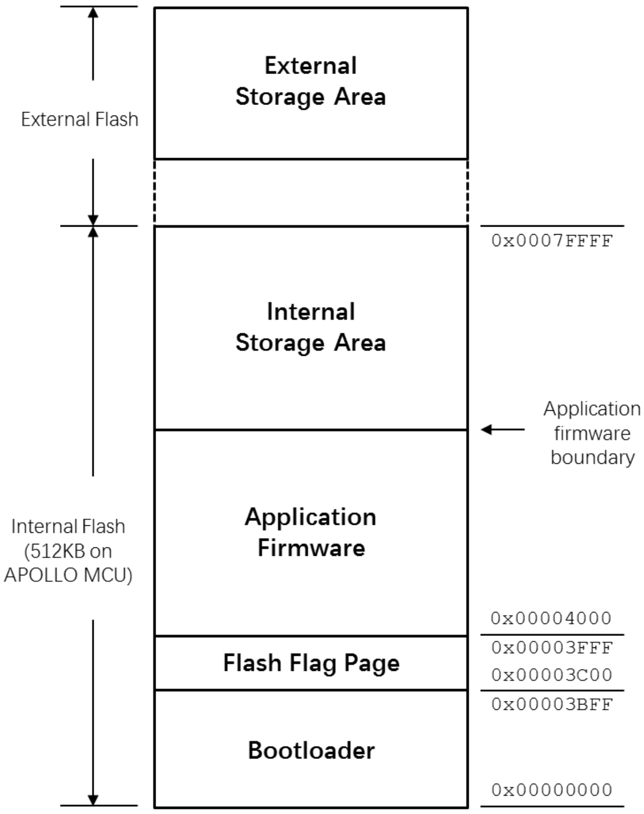


FIGURE 2 APOLLO MCU MEMORY MAP

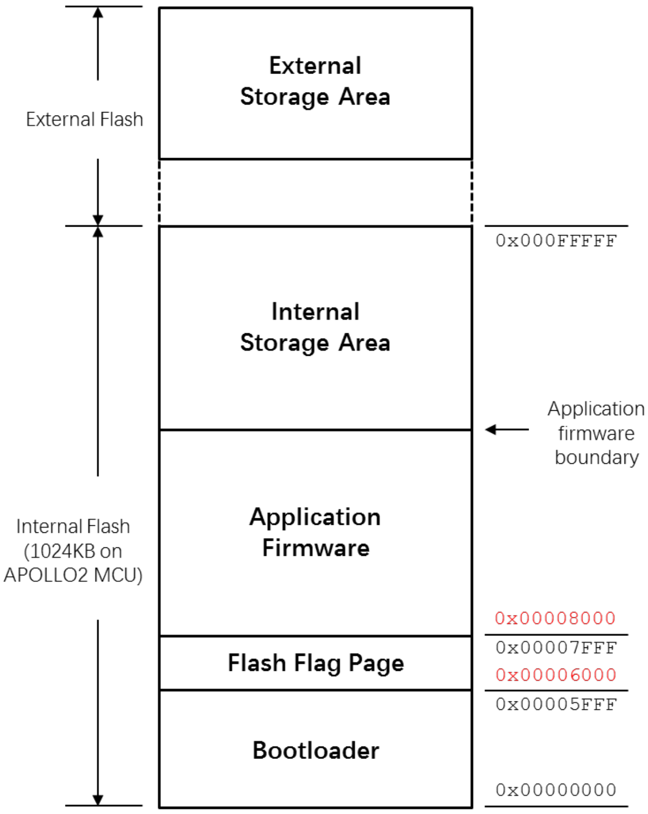


FIGURE 3 APOLLO2 MCU MEMORY MAP

Taking APOLLO as example, 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.

For above mentioned memory mapping, with APOLLO2 MCU, the addresses are different due to the size of the flash page on APOLLO2 is 8K bytes. For APOLLO2, the default flash flag page starting address is mapped to 0x6000, and the application load starting address is mapped to 0x8000.

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:

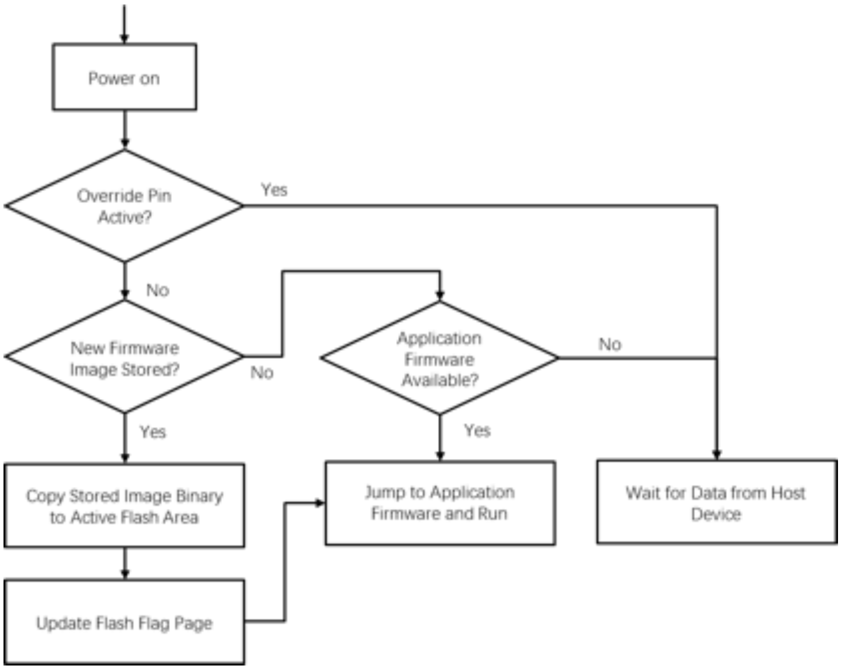


FIGURE 4 BOOT SEQUENCE

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 (ui32CRC, address offset 0x08), and load the image into the target link address (pui32LinkAddress, address offset 0x00) if the image is valid.

Storage type of the new image is specified by user when generating the OTA binary file. The smartphone APP is also able to set the target storage option.

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. (Not used in this example.)
pui32StorageAddressNewImage	4	0x28	Starting address where the new image was stored.
ui32TotalNumBytesNewImage	4	0x2C	Length of the new image being received in bytes. (Not used in this example.)
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. (Not used in this example.)
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

The flash flag page is by default located at 0x3C00 (0x4000 for APOLLO2) of the internal flash, however it is possible for user to specify the start address of the flash flag page to the last page in the internal flash. E.g. 0x7F800 for a device with 512KB internal flash. This can be done by setting the following macro to 1.

```
40 //*****************************************************************************
41 //
42 // Ignore the configured location, and use the last available flash page as the
43 // flag page.
44 //
45 //*****************************************************************************
46 #define USE_LAST_PAGE_FOR_FLAG            1
47
```

The macro is located at line 46 of ..\AmbiqSuite\\boards\apollo_evk_base\examples\multi_boot\src\multi_boot_config.h. After changing to this setting is made, both multi_boot and freertos_amota projects have to be re-built to work with the new setting.

Note: Using the last page of flash as the flash flag page will significantly increase the binary file size of the “combined” binary (boot + application + flash flag page information), due to the gap between the end of the application firmware and the flash flag page is filled with 0xFF in the binary file generated.

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 by 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)

amotaHeaderInfo_t

encrypted: 4 bytes

fwStartAddr: 4 bytes

fwLength: 4 bytes

fwCrc: 4 bytes

secInfoLen: 4 bytes

resvd1: 4 bytes

resvd2: 4 bytes

resvd3: 4 bytes

version: 4 bytes

fwDataType: 4 bytes

storageType: 4 bytes

resvd3: 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_INSUFFICIENT_FLASH,
    AMOTA_STATUS_UNKNOWN_ERROR,
    AMOTA_STATUS_FLASH_WRITE_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

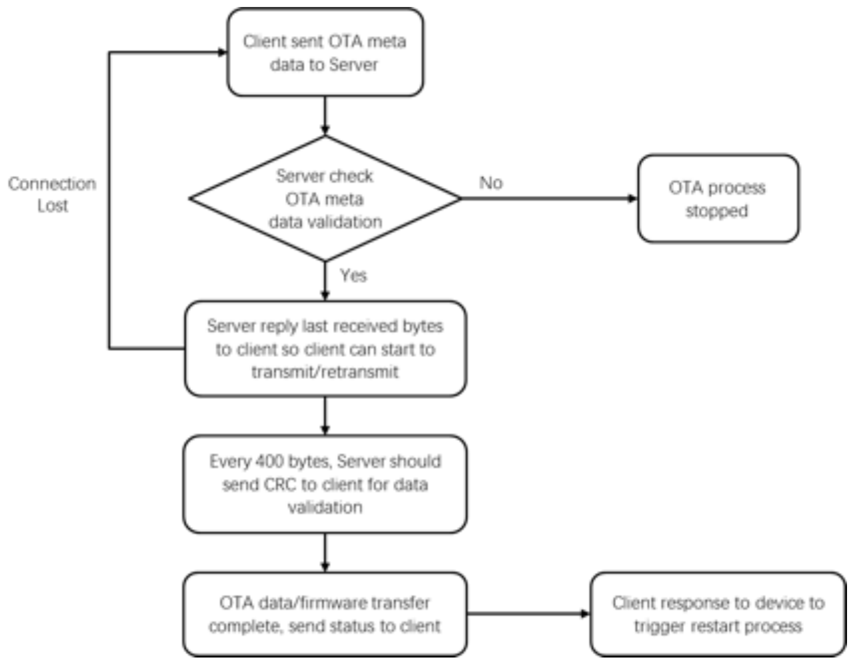


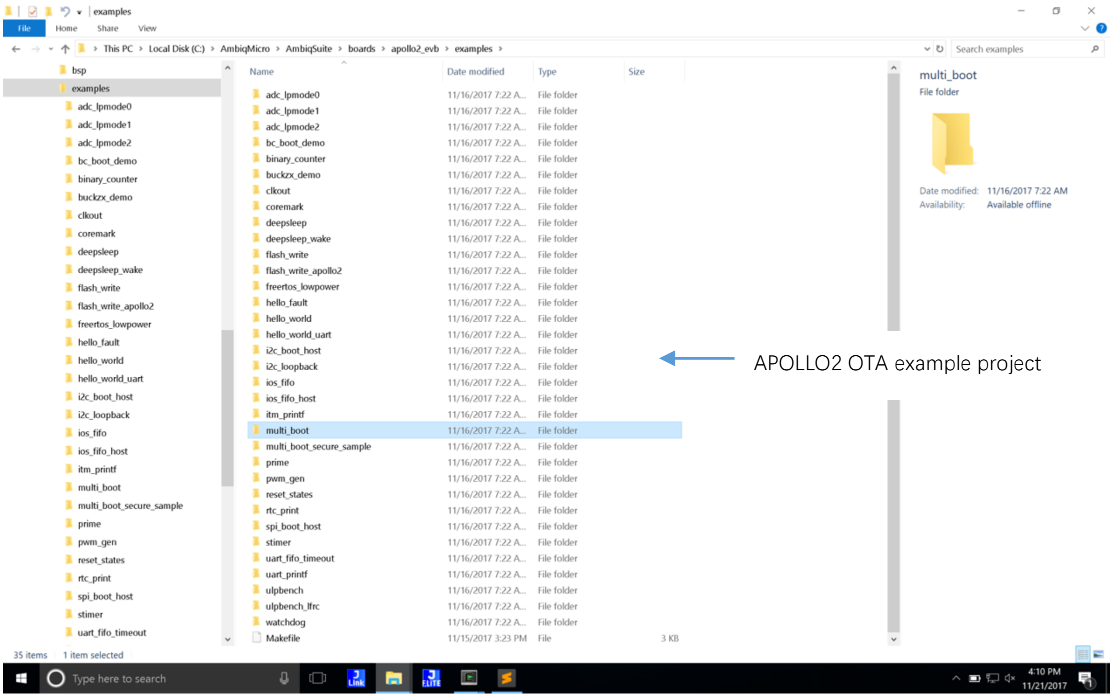
FIGURE 5 OTA SERVICE FLOW

- 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:



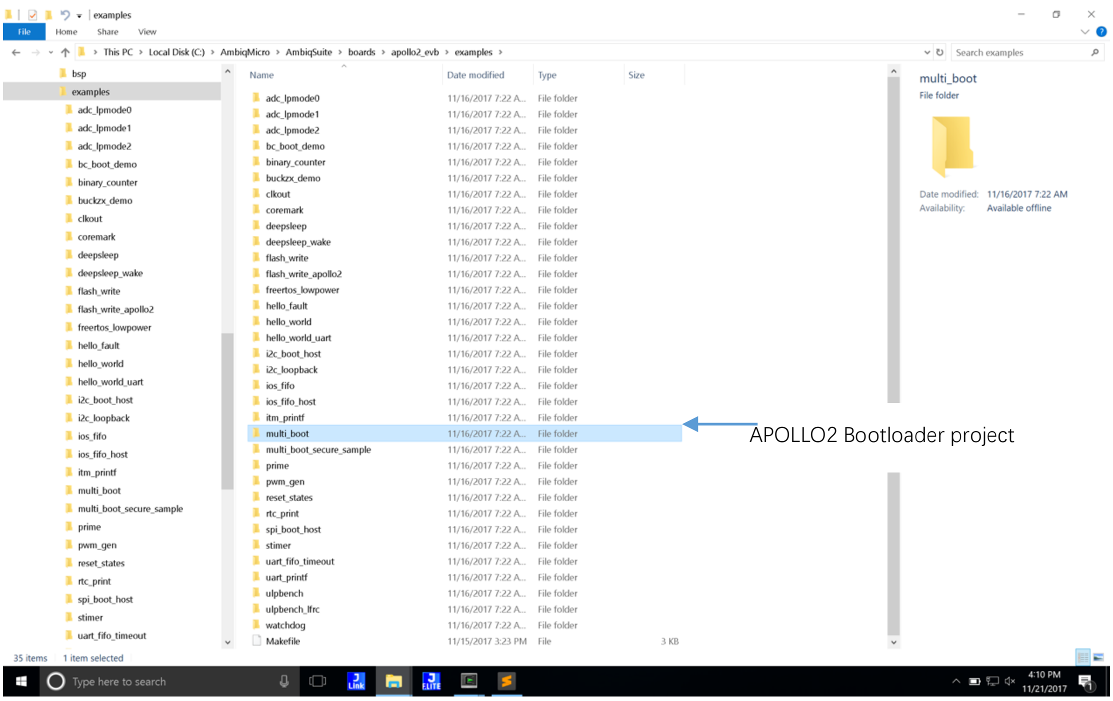


FIGURE 6 FOLDER STRUCTURE

3.2 Development Environment

- Hardware:

This example runs on APOLLO+EM9304 shield board and APOLLO2-BLUE EVB, make sure you have one available to run the example.
For details of the EVKs, please check ..\AmbiqMicro\AmbiqSuite\docs\boards\apollo_evk\apollo_evk_users_guide.pdf.
Otherwise, modifications need to be done according to the hardware setup in the BSP of both exactle_fit_amota_multi_boot and freertos_amota folders.

- Software:

Install the latest AmbiqSuite.

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.
iOS:

Install iTunes PC tool for iOS device APP installation and file sharing.
Visit our APP page on Apple AppStore at: https://itunes.apple.com/us/app/ambiq-ota/id1190453962?mt=8
Or simply search for “Ambiq OTA” in the AppStore to install.

Android:

Install our Ambiq OTA app directly from the APK located at:
..\AmbiqMicro\AmbiqSuite\tools\amota

3.3 Run the example

Navigate to the /AmbiqSuite/tools/amota/scripts folder.

Run “make” in this folder:

```
$ make
python3 bootloader_binary_combiner.py --bootbin "../../../boards/apollo2_evb_em9304/exampl
es/exactle_fit_amota_multi_boot/keil/bin/exactle_fit_amota_multi_boot.bin" --appbin "../..
/../boards/apollo2_evb_em9304/examples/freertos_amota/keil/bin/freertos_amota.bin" --flag-
addr 0x6000 --load-address 0x8000 -o starter_binary
boot size  12868
pad_length  19900
load_address  0x8000 ( 32768 )
app_size  0x13dd0 ( 81360 )
crc =    0x4742cde0
python3 ota_binary_converter.py --appbin "./binary_counter.bin" --load-address 0x8000 -o u
pdate_binary
pad_length  48
load_address  0x8000 ( 0x8000 )
app_size  0x19e8 ( 6632 )
crc =    0x4147fa8d
app_ver 0 ( 0x0 )
bin_type 0 ( 0x0 )
str_type 0 ( 0x0 )
```

Note: it will take a while to build all the required binaries for Apollo1 and Apollo2-Blue EVB.

There are four final binaries generated from the above step:

1. `starter_binary_apollo1.bin`, which is used to load into Apollo1 based board.
2. `starter_binary_apollo2_blue.bin`, which is used to load into Apollo2-Blue EVB.
3. `update_binary_apollo1.bin`, which should be uploaded into smartphone app to transfer it over the air to Apollo1 based EVB.
4. `update_binary_apollo2_blue.bin`, which should be uploaded into smartphone app to transfer it over the air to Apollo2-BLUE EVB.

Load the `starter_binary_apollo1.bin` or `starter_binary_apollo2_blue.bin` into the target MCU using the J-Link Flash as follows:

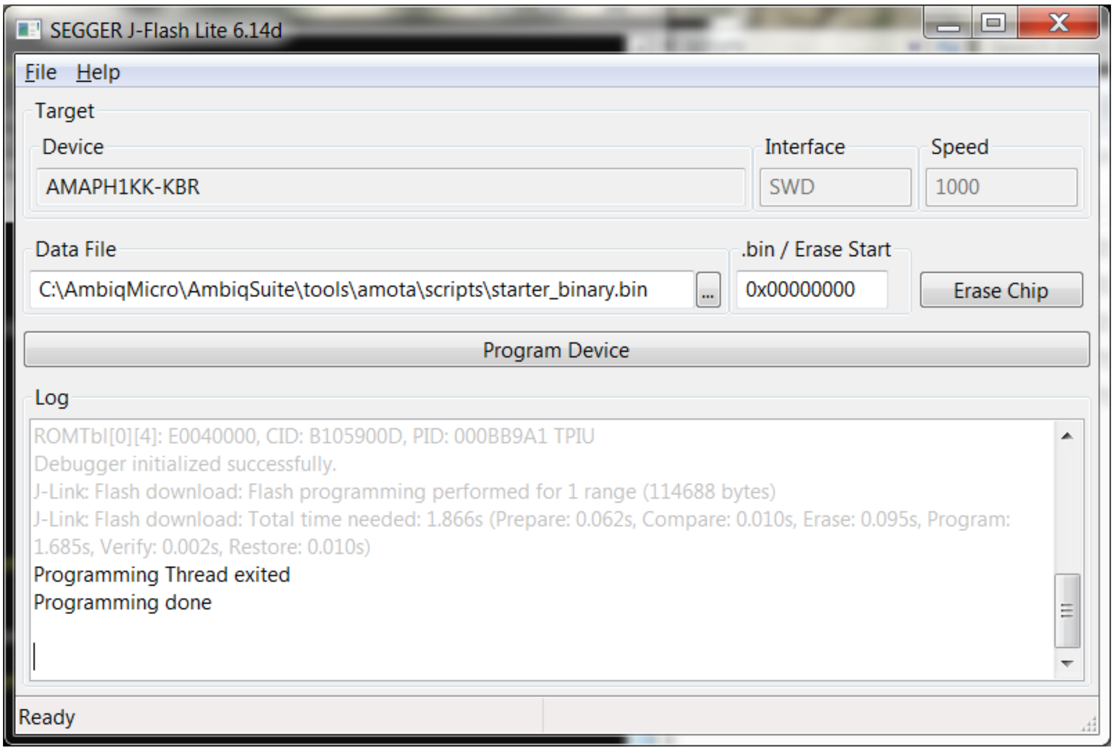
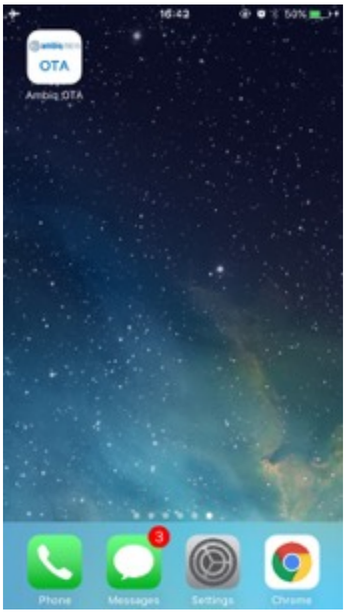


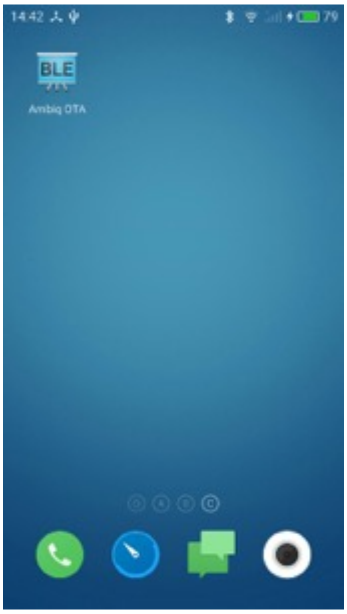
FIGURE 7 J-LINK FLASH LITE SCREENSHOT

Press the reset button to start the application.

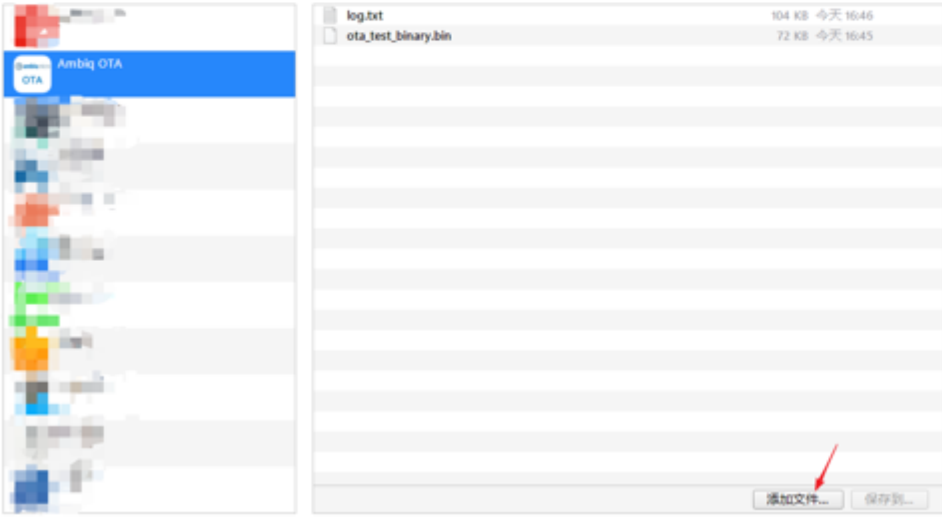
1. iOS:
   Install the iOS APP from Apple AppStore by searching for Ambiq OTA.
   Once the APP is installed successfully, it will be shown on the home screen.



   Android:
   Install the APK from the ..\tools\amota folder. Once the APK is installed successfully, it will be shown on the home screen.

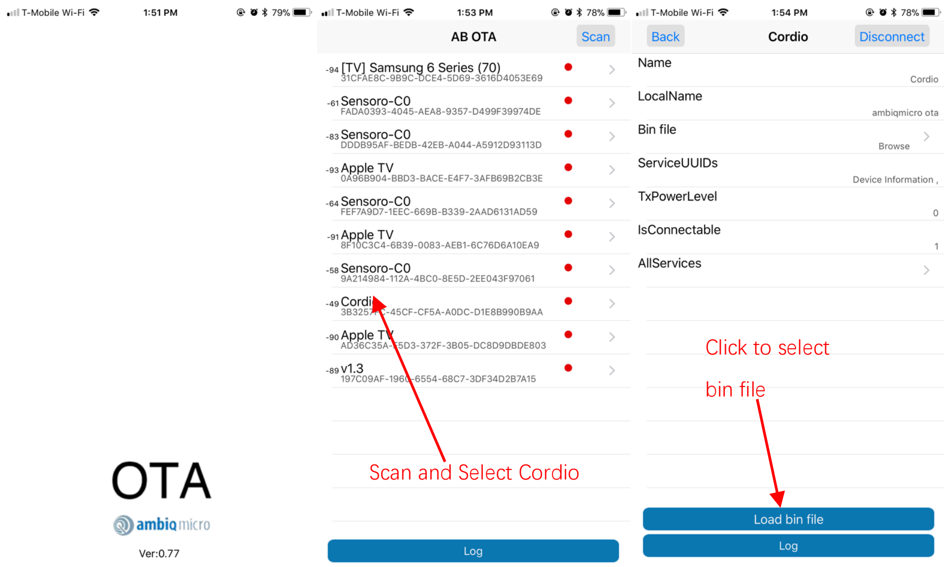


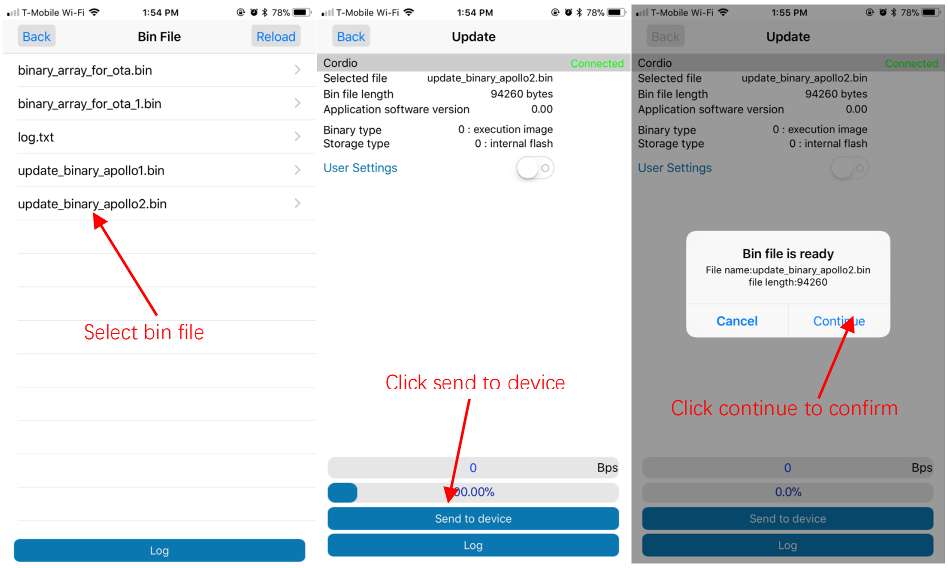
2. Load the `update_binary_apollo1.bin` and `update_binary_apollo2_blue.bin` into the APP.
   iOS: Connect the smart phone with PC, start iTunes and load the binary file into OTA APP. With iTunes:

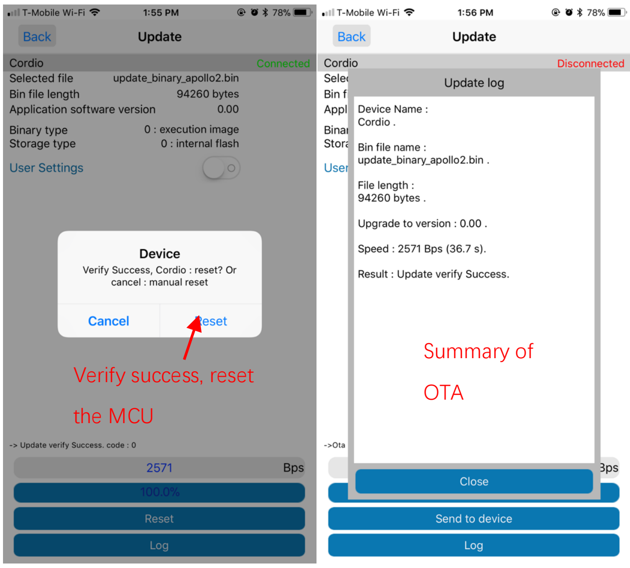


Android: Enable the USB storage media device when the smart phone is connected to PC and move the target binary file into any visible storage directory. E.g. to ..\storage\emulated\0\Debug\ota_binary

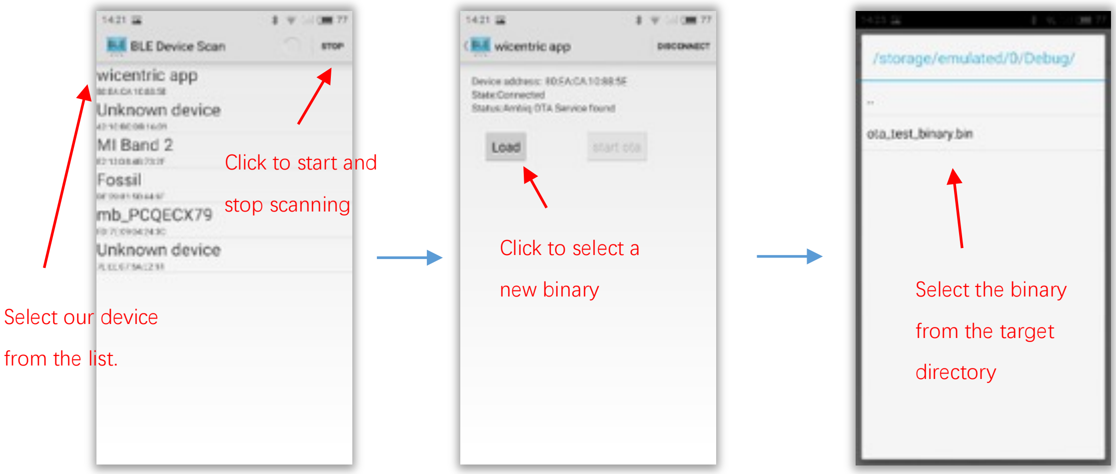
3. Start OTA APP from smart phone and send the `update_binary_apollo1.bin` or `update_binary_apollo2_blue.bin` via BLE.
   With iOS

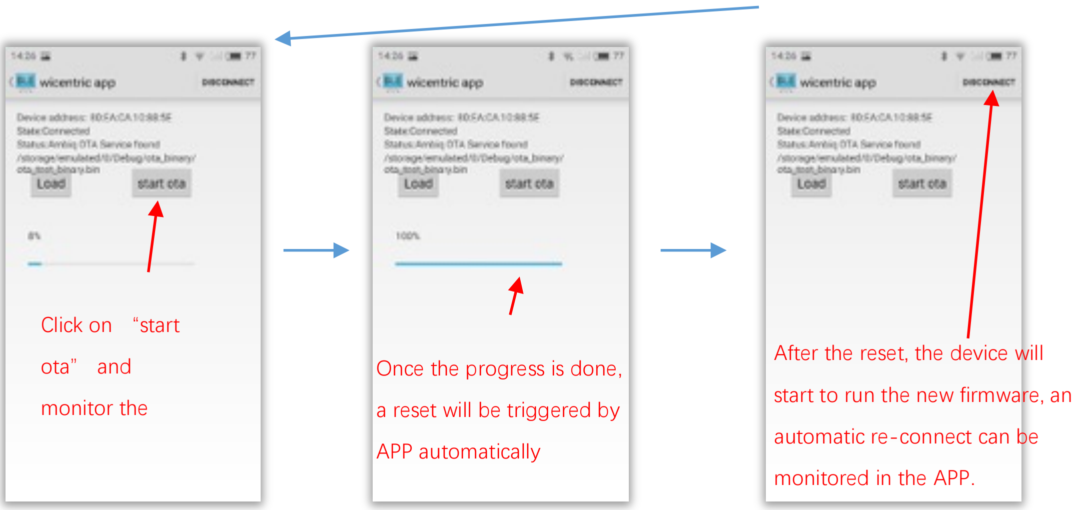






With Android:





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

4.1 For APOLLO MCU

	Item	Typ.	Unit	Remark
Data Transfer speed				
	Storage in internal flash	4.9	KB/s	with iOS App V0.51
	Storage in external flash	4.8	KB/s	with iOS App V0.51
Resource Consumption				
	Bootloader code size	10.27	KB	
	Bootloader RAM size	5.2	KB	
	Flash flag page	1	KB	224 bytes
	OTA project code size	89.70	KB	
	OTA project RAM size	7.27	KB	2KB stack
Execution Timing				
	Boot from internal flash	1.98	sec	89.70KB image
	Boot from external flash	2.45	sec	89.70KB image@8MHz
	Flash write to internal flash	4.8	msec	512bytes
	Flash write to external flash	6.4	msec	256bytes @ 8MHz

4.2 For APOLLO2-BLUE MCU

	Item	Typ.	Unit	Remark
Data Transfer speed				
	Storage in internal flash	4.9	KB/s	with iOS App V0.51
	Storage in external flash	3.4	KB/s	with iOS App V0.51
Resource Consumption				
	Bootloader code size	14.96	KB	
	Bootloader RAM size	17.60	KB	Buffer for 1 flash page
	Flash flag page	8	KB	224 bytes
	OTA project code size	89.85	KB	
	OTA project RAM size	7.8	KB	2KB stack.
Execution Timing				
	Boot from internal flash	0.624	sec	91.32KB image
	Boot from external flash	1.01	sec	91.32KB image@8MHz
	Flash write to internal flash	1.8	msec	512bytes
	Flash write to external flash	4.1	sec	256bytes@8MHz