# 1 Bootloader overview

A Bootloader is a small application that is used to load new user applications to devices. After it is loaded, the new user application is able to run in the MCU. It works with two modes, user application and Bootloader mode.

We are discussing common way to implement Bootloader on your own Arm Cortez M3. And we choose Silicon labs SIM3U167 MCU as example for making Bootloader.

# 2 Bootloader design method

## 2.1 Bootloader need to run in SRAM

### 2.1.1 Bootloader behavior

Bootloader is an application which runs from internal flash, in case of update Bootloader itself requirement, the Bootloader code needed to be copied into internal SRAM and run.

* The Bootloader code address need to set in SRAM in link file
* The vector table entry needs to update from flash to SRAM after copying Bootloader into SRAM.
* The Reset vector address we want to keep its original address, since it is hardware behavior.

### 2.1.2 Arm Cortex M3 internal memory map

* Internal flash address (0x00000000-0x1FFFFFFF).
* Internal SRAM address (0x20000000-0x3FFFFFFF).
* Peripherals address (0x40000000-0x5FFFFFFF).
* System levels address (0xE0000000-0xFFFFFFFF).

So base on above memory, we need to copy code from internal flash address to internal SRAM address. And access UART peripheral register in Peripherals address, and change vector table address need to access register in System levels address.

### 2.1.3 Link file

To make sure Bootloader code is located in SRAM, we need to specific the address setting in link file. Now we have a Bootloader\_link.ld as our link file. We would talk detail about how to write link file, we assume that you already know the basic knowledge about link file.

.text 0x20000000 : AT (0x00000000)

{

. = ALIGN(4);

\_text = .;

KEEP(\*(.isr\_vector))

\*(.text\*)

\*(.rodata\*)

\_etext = .;

}

The first line “.text 0x20000000 : AT (0x00000000)” means .text section load address start from 0x20000000, store address from 0x00000000.

Inside the description, we put “.isr\_vector”, \*(.text\*) and \*(.rodata\*) in .text section. The “.isr\_vector” is vector entry; we put it in front of Bootloader code.

And then we have other two sections .data and .bss

.data 0x20000000 + SIZEOF(.text) : AT (LOADADDR(.text) + SIZEOF(.text))

{

\_data = .;

\*(.data\*)

\_edata = .;

}

.bss 0x20000000 + SIZEOF(.text) + SIZEOF(.data) :

AT (LOADADDR(.data) + SIZEOF(.data))

{

\_bss = .;

\*(.bss\*)

\*(COMMON)

\_ebss = .;

}

And now the link file is ready.

### 2.1.4 Vector table introduction.

By default Arm Cortex M3 vector table starts at memory address 0. The vector table can be relocated to other memory locations in the code or Random Access Memroy(RAM) region where the RAM is so that we can change the handlers during run time. This is down by settting a register in the NVIC called the vector table offset register (address 0xE000ED08). You need to have the following (at a minimum).

• Initial main stack pointer value

• Reset vector

• NMI vector

• Hard fault vector

### 2.1.5 Vector table in startup file

We prepare startup\_sim3u1xx.S for our startup file. In this file, we will have the vector table, code copy from flash to SRAM, jump to user application code. Let us have a look on vector table first. We can see .isr\_vector in link file is the vector table section name.

**.section** .isr\_vector

**Vectors:**

**.word** 0x20008000 // The initial stack pointer

**.word** ResetISR - 0x20000000 // The reset handler

**.word** NMI\_Handler // The NMI handler

**.word** HardFault\_Handler // The hard fault handler

**.word** Default\_Handler // The MPU fault handler

**.word** Default\_Handler // The bus fault handler

**.word** Default\_Handler // The usage fault handler

**.word** 0 // Reserved

**.word** 0 // Reserved

**.word** 0 // Reserved

**.word** 0 // Reserved

**.word** Default\_Handler // SVCall handler

**.word** Default\_Handler // Debug monitor handler

**.word** 0 // Reserved

**.word** Default\_Handler // The PendSV handler

**.extern** SysTick\_Handler

**.word** SysTick\_Handler // The SysTick handler

Note:

* Stack pointer we set is 0x20008000, that is because SIM3U167 internal SRAM size is 32KB (0x20000000-0x20007FFF).
* Reset handler address we minus 0x20000000, that is because, Arm will check the value in second word and jump to address in it. Since all Bootloader code was locate at SRAM (0x20000000), but at the time power up, all the codes are in internal flash, and didn’t copy to SRAM yet. So we have to set ResetISR address point to flash in order Arm can execute code correctly.
* Other exception handlers. We need to keep some import exception handlers; we will see HardFault\_Handler happens many times during the Bootloader development in case the address not matches.

### 2.1.6 ResetISR implements.

We need to handle code that Arm will first execute, ResetISR. What does this function do?

* Copy code to SRAM and initialize variables.
* Initialize system hardware
* Check external pin status to run under Bootloader or user application mode.

Here is the cod structure of ResetISR

**.globl** ResetISR

.thumb\_func

**ResetISR:**

// Copy code from flash to SRAM

bl CopyCode2SRAM

**.extern** SystemInit

bl SystemInit

// Check if update is needed

**.extern** check\_update\_requirement

bl check\_update\_requirement

cbz r0, RunUserCode

**.extern** main

bl main

#### 2.1.6.1 Copy code to SRAM and initialize variables.

Let us take a look on CopyCode2SRAM function.

**.text**

.thumb\_func

**CopyCode2SRAM:**

// Copy the text and data sections from flash to SRAM.

movs r0, #0x00000000

ldr r1, =0x20000000

**.extern** \_bss

ldr r2, =\_bss

**copy\_loop:**

ldr r3, [r0], #4

str r3, [r1], #4

cmp r1, r2

blt copy\_loop

// Zero fill the bss segment

movs r0, #0x00000000

**.extern** \_ebss

ldr r2, =\_ebss

**zero\_loop:**

str r0, [r1], #4

cmp r1, r2

blt zero\_loop

// Set the vector table pointer to SRAM.

ldr r0, =0xe000ed08

ldr r1, =0x20000000

str r1, [r0]

// set return address to SRAM and return

orr lr, lr, #0x20000000

bx lr

The code is very simple.

* copy .text and .data from flash to SRAM
* Zero .bss segment in SRAM
* Relocate vector table address and return to SRAM to execute.

Note:

* \_bss is defined in link file, which is address after .text and .data. So the copy\_loop is copy code from flash to SRAM until it reaches \_bss address.
* \_ess is defined in linke file, which is address after .bss segment. So zero\_lopp, just write “0” into SRAM follow the .text and .data segment, until it reaches \_ebss address.
* And finally, we write new vector table address (0x20000000) into register 0xe00ed08, and set return address to SRAM and return.

#### 2.1.6.2 Initialize system hardware

The idea is very simple too, set necessary hardware environment we need in startup file. For instance, we need to check GPIO status to choose running mode. We have to enable clock to GPIO and set the GPIO as input mode and something like that. I paste the example of SystemInit.

**void** **mySystemInit**(**void**)

{

SI32\_WDTIMER\_A\_stop\_counter (SI32\_WDTIMER\_0);

// Enable the APB clock to the PB registers

SI32\_CLKCTRL\_A\_enable\_apb\_to\_modules\_0(SI32\_CLKCTRL\_0, SI32\_CLKCTRL\_A\_APBCLKG0\_PB0);

SI32\_PBSTD\_A\_set\_pins\_push\_pull\_output(SI32\_PBSTD\_1, 0x00000008);

SI32\_PBCFG\_A\_enable\_crossbar\_1(SI32\_PBCFG\_0);

SI32\_PBSTD\_A\_set\_pins\_digital\_input(SI32\_PBSTD\_2,0x00000300);

// Enable the LED drivers (P2.10, P2.11)

SI32\_PBSTD\_A\_set\_pins\_push\_pull\_output(SI32\_PBSTD\_2, 0x00000C00);

SysTick\_Config(SystemCoreClock / 1000);

// set Priority for Cortex-M0 System Interrupts.

NVIC\_SetPriority(*SysTick\_IRQn*, (1 << \_\_NVIC\_PRIO\_BITS) - 1);

}

#### 2.1.6.3 Check external pin status to run under Bootloader or user application mode

The code call *check\_update\_requirement* to check GPIO status, if the GPIO is high, we enter into user application mode; if the GPIO is low, we enter into Bootloader mode. Let’s check user application mode entry code.

.thumb\_func

**RunUserCode:**

// Set the vector table address to user code address.

ldr r0, =USER\_CODE\_ADDRESS

ldr r1, =0xe000ed08

str r0, [r1]

// Update stack pointer from user code vector table

ldr r1, [r0]

mov sp, r1

// Load user code reset handler and jump to the user code

ldr r0, [r0, #4]

bx r0

Yeah, we have to relocate vector table if we want run user application. Note we have a macro “USER\_CODE\_ADDRESS” which is user application start address. At the end we load user code reset handler and jump to the user code.

OK, we will talk about Bootloader mode in next couple sections.

## 2.2 Bootloader need to get UART work to communicate with PC host tool.

We choose UART interface as communication port with PC host since it is easier to implement. Just have a look on SIM3U167 reference manual, initialize UART peripherals, and set it as 115200 baud rate. Prepare UART send and received functions.

* UART hardware peripherals initialization. Those codes can be generated by Silabs AppBuilder.

**Void** **UART0\_initialize**(**void**)

{

SI32\_PBCFG\_A\_enable\_crossbar\_0(SI32\_PBCFG\_0);

// UART PINS TO PROPER CONFIG (TX = PB1.12, RX = PB1.13)

SI32\_PBSTD\_A\_set\_pins\_push\_pull\_output(SI32\_PBSTD\_1, 0x0001000);

SI32\_PBSTD\_A\_set\_pins\_digital\_input(SI32\_PBSTD\_1, 0x00002000);

SI32\_PBSTD\_A\_write\_pbskipen(SI32\_PBSTD\_0, 0x0000FFFF);

SI32\_PBSTD\_A\_write\_pbskipen(SI32\_PBSTD\_1, 0x00000FFF);

// BRING OUT UART

SI32\_PBCFG\_A\_enable\_xbar0h\_peripherals(SI32\_PBCFG\_0, SI32\_PBCFG\_A\_XBAR0H\_UART0EN);

// ENABLE UART0 CLOCK

SI32\_CLKCTRL\_A\_enable\_apb\_to\_modules\_0(SI32\_CLKCTRL\_0,

SI32\_CLKCTRL\_A\_APBCLKG0\_UART0);

// SETUP UART. BAUD RATE (9600 baud, APB = System Clock)

SI32\_UART\_A\_enter\_full\_duplex\_mode(SI32\_UART\_0);

SI32\_UART\_A\_set\_tx\_baudrate(SI32\_UART\_0, (SystemCoreClock / (2 \* UART\_BAUD\_RATE)) – 1);

SI32\_UART\_A\_set\_rx\_baudrate(SI32\_UART\_0, (SystemCoreClock / (2 \* UART\_BAUD\_RATE)) – 1);

// SETUP TX (8-bit, 1stop, no-parity)

SI32\_UART\_A\_select\_tx\_data\_length(SI32\_UART\_0, 8);

SI32\_UART\_A\_enable\_tx\_start\_bit(SI32\_UART\_0);

SI32\_UART\_A\_enable\_tx\_stop\_bit(SI32\_UART\_0);

SI32\_UART\_A\_disable\_tx\_parity\_bit(SI32\_UART\_0);

SI32\_UART\_A\_select\_tx\_stop\_bits(SI32\_UART\_0, *SI32\_UART\_A\_STOP\_BITS\_1\_BIT*);

SI32\_UART\_A\_disable\_tx\_signal\_inversion(SI32\_UART\_0);

SI32\_UART\_A\_enable\_tx(SI32\_UART\_0);

// SETUP RX

SI32\_UART\_A\_select\_rx\_data\_length(SI32\_UART\_0, 8);

SI32\_UART\_A\_enable\_rx\_start\_bit(SI32\_UART\_0);

SI32\_UART\_A\_enable\_rx\_stop\_bit(SI32\_UART\_0);

SI32\_UART\_A\_select\_rx\_stop\_bits(SI32\_UART\_0, *SI32\_UART\_A\_STOP\_BITS\_1\_BIT*);

SI32\_UART\_A\_disable\_rx\_signal\_inversion(SI32\_UART\_0);

SI32\_UART\_A\_select\_rx\_fifo\_threshold\_1(SI32\_UART\_0);

SI32\_UART\_A\_enable\_rx(SI32\_UART\_0);

}

* UART send and receive functions.

**Void** **uart\_send\_data**(**unsigned** **char** \*data, **unsigned** **int** count)

{

**while**(count--)

{

// Block if the output buffer is full

**while** (SI32\_UART\_A\_read\_tx\_fifo\_count(SI32\_UART\_0) >= 4);

// Write character to the output buffer

SI32\_UART\_A\_write\_data\_u8(SI32\_UART\_0, \*data++);

}

}

**int** **uart\_get\_data**(**unsigned** **char** \*data, **unsigned** **int** count)

{

**unsigned** **int** time\_out;

**while**(count--)

{

time\_out = msTicks + UART\_TIME\_OUT;

**while**(SI32\_UART\_A\_read\_rx\_fifo\_count(SI32\_UART\_0) == 0)

{

**if**(time\_out < msTicks)

{

**return** -1;

}

}

\*data++ = SI32\_UART\_A\_read\_data\_u8(SI32\_UART\_0);

}

**return** 0;

}

## 2.3 Bootloader need to burn the firmware gets from PC host into internal flash.

The UART driver was ready. We need to write user application into flash and flash device driver is needed now.

* Flash hardware peripherals initialization.

**Void** **flash\_initialize**(**void**)

{

// ENABLE FLASH CLOCK

SI32\_CLKCTRL\_A\_enable\_apb\_to\_modules\_0(SI32\_CLKCTRL\_0,

SI32\_CLKCTRL\_A\_APBCLKG0\_FLCTRLCEN\_ENABLED\_U32);

// 1. Enable VDD Supply Monitor and set as a reset source

SI32\_VMON\_A\_enable\_vdd\_supply\_monitor(SI32\_VMON\_0);

SI32\_RSTSRC\_A\_enable\_vdd\_monitor\_reset\_source(SI32\_RSTSRC\_0);

}

* Flash operation functions

**void** **flash\_erase\_page**(**unsigned** **int** addr)

{

SI32\_FLASHCTRL\_A\_exit\_multi\_byte\_write\_mode(SI32\_FLASHCTRL\_0);

SI32\_FLASHCTRL\_A\_write\_wraddr(SI32\_FLASHCTRL\_0, addr);

SI32\_FLASHCTRL\_A\_enter\_flash\_erase\_mode(SI32\_FLASHCTRL\_0);

SI32\_FLASHCTRL\_A\_write\_flash\_key(SI32\_FLASHCTRL\_0, 0xA5);

SI32\_FLASHCTRL\_A\_write\_flash\_key(SI32\_FLASHCTRL\_0, 0xF1);

SI32\_FLASHCTRL\_A\_write\_wrdata(SI32\_FLASHCTRL\_0, 0x0000);

**while**(SI32\_FLASHCTRL\_A\_is\_buffer\_full(SI32\_FLASHCTRL\_0));

}

**int** **flash\_write\_data**(**unsigned** **char** \* c,**unsigned** **int** addr, **unsigned** **int** num)

{

**int** i;

**unsigned** **short** \*buf = (**unsigned** **short**\*)c;

SI32\_FLASHCTRL\_A\_exit\_flash\_erase\_mode(SI32\_FLASHCTRL\_0);

SI32\_FLASHCTRL\_A\_write\_wraddr(SI32\_FLASHCTRL\_0, addr);

SI32\_FLASHCTRL\_A\_enter\_multi\_byte\_write\_mode(SI32\_FLASHCTRL\_0);

SI32\_FLASHCTRL\_A\_write\_flash\_key(SI32\_FLASHCTRL\_0, 0xA5);

SI32\_FLASHCTRL\_A\_write\_flash\_key(SI32\_FLASHCTRL\_0, 0xF2);

// Write all the half-words in the array

**for**(i = 0; i < num / 2; i++)

{

SI32\_FLASHCTRL\_A\_write\_wrdata(SI32\_FLASHCTRL\_0, buf[i]);

**while**(SI32\_FLASHCTRL\_A\_is\_flash\_busy(SI32\_FLASHCTRL\_0));

}

**while**(SI32\_FLASHCTRL\_A\_is\_buffer\_full(SI32\_FLASHCTRL\_0));

SI32\_FLASHCTRL\_A\_write\_flash\_key(SI32\_FLASHCTRL\_0, 0x5A);

**return** 0;

}

# 3 Bootloader UART communication protocol

## 3.1 We can define our own protocol as we like. Also choose exist one is suitable. We list a protocol below just for reference. From view of host side

* Command package

CMD

CMD ^ 0xFF

Wait for ACK

* Address package

ADDR3

ADDR2

ADDR1

ADDR0

CRC(=ADDR3^ADDR2^ADDR1^ADDR0)

* Write flash command

CMD(0x31)

ADDR(address)

Wait for ACK

(LEN-1) & 0XFF

DATA0

…

DATA(LEN-1)

CRC(=DATA0^DATA1…^DATA(LEN-1))

Wait for ACK

* Read flash command

CMD(0x11)

ADDR(address)

Wait for ACK

(LEN-1) & 0XFF

CRC (((LEN-1) & 0XFF) ^ 0xFF)

Wait for ACK

DATA0

…

DATA(LEN-1)

* Erase flash command

CMD(0x43)

LEN-1

SEC0

…

SEC(LEN-1)

CRC(SEC0^SEC1…^SEC(LEN-1))

Wait for ACK

The device side control logic was list below.

**While** (1)

{

SI32\_PBSTD\_A\_toggle\_pins(SI32\_PBSTD\_2, 0xC00);

// received CMD from host

**if** (bl\_get\_cmd() == 0)

{

**switch** (data\_buf[0])

{

**case** CMD\_WRITE\_FLASH:

bl\_write\_flash();

**break**;

**case** CMD\_READ\_FLASH:

bl\_read\_flash();

**break**;

**case** CMD\_ERASE\_FLASH:

bl\_erase\_flash();

**break**;

**default**:

**break**;

}

}

}

## 3.2 And the protocol functions list below:

**#define** FLASH\_SECTOR\_SIZE 0x400

**#define** CMD\_WRITE\_FLASH 0x31

**#define** CMD\_READ\_FLASH 0x11

**#define** CMD\_ERASE\_FLASH 0x43

**#define** CMD\_UNKNOW 0xFF

**#define** RES\_ACK 0x79

**#define** RES\_NACK 0x1F

**#define** RES\_UNKNOW 0xEE

**void** **bl\_send\_ack**(**unsigned** **char** ack)

{

**unsigned** **char** res = ack;

uart\_send\_data(&res, 1);

}

* Get command from host

**int** **bl\_get\_cmd**(**void**)

{

**unsigned** **char** \*ptr = data\_buf;

**if** (uart\_get\_data(ptr, 2) == 0)

{

**if** (ptr[0] != (ptr[1] ^ 0xFF))

{

bl\_send\_ack(RES\_NACK);

**return** -2;

}

**else**

{

bl\_send\_ack(RES\_ACK);

**return** 0;

}

}

**return** -1;

}

* Handle write flash command

**void** **bl\_write\_flash**()

{

**int** i;

**unsigned** **int** addr, len;

**unsigned** **char** crc;

**unsigned** **char** \*ptr = data\_buf;

// get address

**if** (uart\_get\_data(ptr, 5) == 0)

{

crc = ptr[0] ^ ptr[1] ^ ptr[2] ^ ptr[3];

**if** (crc == ptr[4])

{

addr = ptr[3] | (ptr[2] << 8) | (ptr[1] << 16) | (ptr[0] << 24);

}

}

// send ack

bl\_send\_ack(RES\_ACK);

// get data len

uart\_get\_data(ptr, 1);

len = ptr[0] + 1;

// get data and crc

**if** (uart\_get\_data(ptr, len + 1) == 0)

{

crc = 0xFF;

**for** (i = 0; i < len; i++)

{

crc = crc ^ ptr[i];

}

}

//write flash

**if** (crc == ptr[len])

{

**if** ((addr >= USER\_CODE\_ADDRESS) && addr < TOP\_CODE\_ADDRESS)

{

flash\_write\_data(ptr, addr, len);

}

}

// send ack

bl\_send\_ack(RES\_ACK);

}

* Handle read flash command

**void** **bl\_read\_flash**()

{

**int** i;

**unsigned** **int** addr, len = 0;

**unsigned** **char** crc;

**unsigned** **char** \*ptr = data\_buf;

// get address

**if** (uart\_get\_data(ptr, 5) == 0)

{

crc = ptr[0] ^ ptr[1] ^ ptr[2] ^ ptr[3];

**if** (crc == ptr[4])

{

addr = ptr[3] | (ptr[2] << 8) | (ptr[1] << 16) | (ptr[0] << 24);

}

}

// send ack

bl\_send\_ack(RES\_ACK);

// get data len and crc

**if**(uart\_get\_data(ptr, 2) == 0)

{

**if**((ptr[0] ^ 0xFF) == ptr[1])

{

len = ptr[0] + 1;

}

}

// send ack

bl\_send\_ack(RES\_ACK);

// send data

**for**(i =0; i < len; i++)

{

ptr[i]= \*(**unsigned** **char** \*)(addr++);

}

uart\_send\_data(ptr,len);

}

* Handle erase flash command

**void** **bl\_erase\_flash**()

{

**int** i;

**unsigned** **int** addr, len;

**unsigned** **char** crc;

**unsigned** **char** \*ptr = data\_buf;

// get len

**if** (uart\_get\_data(ptr, 1) == 0)

{

len = (ptr[0] + 1) & 0xFF; // maximum sector number is 256

}

// get sector sequence and crc

**if** (uart\_get\_data(ptr, len + 1) == 0)

{

crc = 0xFF;

**for** (i = 0; i < len; i++)

{

crc = crc ^ ptr[i];

}

}

// erase flash sectors

**if** (crc == ptr[len])

{

**for** (i = 0; i < len; i++)

{

addr = ptr[i] \* FLASH\_SECTOR\_SIZE;

**if** ((addr >= USER\_CODE\_ADDRESS) && addr < TOP\_CODE\_ADDRESS)

{

flash\_erase\_page(addr);

}

}

}

// send ack

bl\_send\_ack(RES\_ACK);

}

# 4 Choose Python as host UART tool

It is time to pick PC host UART tool. For easier usage purpose, we choose Python. I had tried to use Ruby serial port gem before Python. However, it is not stable. So Python become the first choice. And very lucky, I found a Python Bootloader UART source code.

* Install Python 2.6/2.7

Go to web site <http://www.python.org> to download python and install it under windows.

* Download Python Serial Port Extension

Get it from web site <http://pypi.python.org/pypi/pyserial>

* Install Python Serial Port Extension

Extract the package under Python install directory; execute command below to install the Serial Port Extension.

$python setup.py install

* Get stm32loader.py and make you own modification.

Get it from web site <https://github.com/jsnyder/stm32loader>.

# 5 Build Bootloader code

We choose Silabs Precision32 UDP MCU Card as our hardware environment.

* Build Bootloader with Precision32 IDE
* Download Bootloader into SIM3U167 internal flash via Serial Wire.
* Power up MCU card and pressing SW2, it enters Bootloader mode, DS3 and DS2 are blinking

# 6 Develop User application with Bootloader

User applications runs on Flash, but start address is not 0x00000000, we have a default value 0x1000. We need to make this modification in user application’s link file. Just change .text segment load address to 0x1000. And we also want to generate binary file in post build stage of the user application.

# 7 Bootloader functional test

We were doing the test on Silabs Precision32 UDP MCU Card.

* Connect USB cable with J10 and PC USB port, PC will found COM4 after power on.
* Copy stm32loader.py to C:\Python26 (your Python install directory).
* Copy user code binary file to C:\Python26 directory, for blinky example code, we have a sim3u1xx\_Blinky.bin.
* Run Python tool.

C:\Python26>python sim32loader.py -wv sim3u1xx\_Blinky.bin

sim3u1xx\_Blinky.bin

[4, 5, 6, 7]

File length 3076 bytes

Write 256 bytes at 0x1000

Write 256 bytes at 0x1100

Write 256 bytes at 0x1200

Write 256 bytes at 0x1300

Write 256 bytes at 0x1400

Write 256 bytes at 0x1500

Write 256 bytes at 0x1600

Write 256 bytes at 0x1700

Write 256 bytes at 0x1800

Write 256 bytes at 0x1900

Write 256 bytes at 0x1A00

Write 256 bytes at 0x1B00

Write 256 bytes at 0x1C00

Read 256 bytes at 0x1000

Read 256 bytes at 0x1100

Read 256 bytes at 0x1200

Read 256 bytes at 0x1300

Read 256 bytes at 0x1400

Read 256 bytes at 0x1500

Read 256 bytes at 0x1600

Read 256 bytes at 0x1700

Read 256 bytes at 0x1800

Read 256 bytes at 0x1900

Read 256 bytes at 0x1A00

Read 256 bytes at 0x1B00

Read 256 bytes at 0x1C00

Verification OK

* Run user application

Press Reset key, we can see DS3 and DS2 are alternate blinking, that means user code Blinky is running.