

---

# N32G45X\_FR\_WB series chip IAP upgrade application note

---

## Introduction

This document mainly introduces the IAP upgrade application routine of N32G45X\_FR\_WB series chips (hereinafter referred to as N32G45X), and the problems and solutions that may be encountered during application development.

## CONTENT

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction of IAP .....</b>                   | <b>1</b>  |
| <b>2</b> | <b>IAP software implementation process.....</b>    | <b>5</b>  |
| 2.1      | Set the start ADDRESS of the APP program.....      | 5         |
| 2.2      | Interrupt vector table offset setting method ..... | 7         |
| 2.3      | In APP project, generate bin file .....            | 8         |
| 2.4      | Software implementation process .....              | 8         |
| <b>3</b> | <b>Download validation .....</b>                   | <b>13</b> |
| 3.1      | Upper computer transmission protocol .....         | 13        |
| 3.2      | Procedure download the BIN file .....              | 14        |
| 3.3      | Validation .....                                   | 15        |
| <b>4</b> | <b>Q&amp;A.....</b>                                | <b>17</b> |
| <b>5</b> | <b>Version history .....</b>                       | <b>18</b> |
| <b>6</b> | <b>Notice .....</b>                                | <b>19</b> |

# 1 Introduction of IAP

IAP is an abbreviation of In Application Programming. It refers to the burning of part of the User Flash by the User's own program during the running process. The purpose is to easily update the firmware program In the product through reserved communication ports after the product is released. Generally, when IAP is implemented, two project codes need to be written when the user program is running to update itself. The first project program does not perform normal functional operations, but only receives programs or data through certain communication methods (such as USB and UART) to update the second part of the code. The second project code is the real functional code. These two parts of project code are burned in different areas of User Flash at the same time. When the chip is powered on, the first project code starts to run, which does the following operations:

- 1. Check whether part 2 code needs to be updated;
- 2. If no update is required, go to ●4.
- 3. Perform the update operation.
- 4. Jump to the second part of code execution;

The first part of the code must be burned in by other means, such as JTAG or ISP. The second part of the code can be burned in using the IAP feature of the first part of the code, or it can be burned in with the first part of the code, and then updated with the first part of the IAP code when the application needs to be updated later.

We will call the first project code Bootloader program, the second project code called APP program, they are generally stored in N32G45X Flash different address range, generally from the lowest address area to store the Bootloader, followed by APP program. **New apps can be stored in Flash as well as Sram for execution, as illustrated in the following chapters.** So according to the above description, we need to implement two programs: Bootloader and APP. The normal program running flow of N32G45X is shown in Figure 1.0.

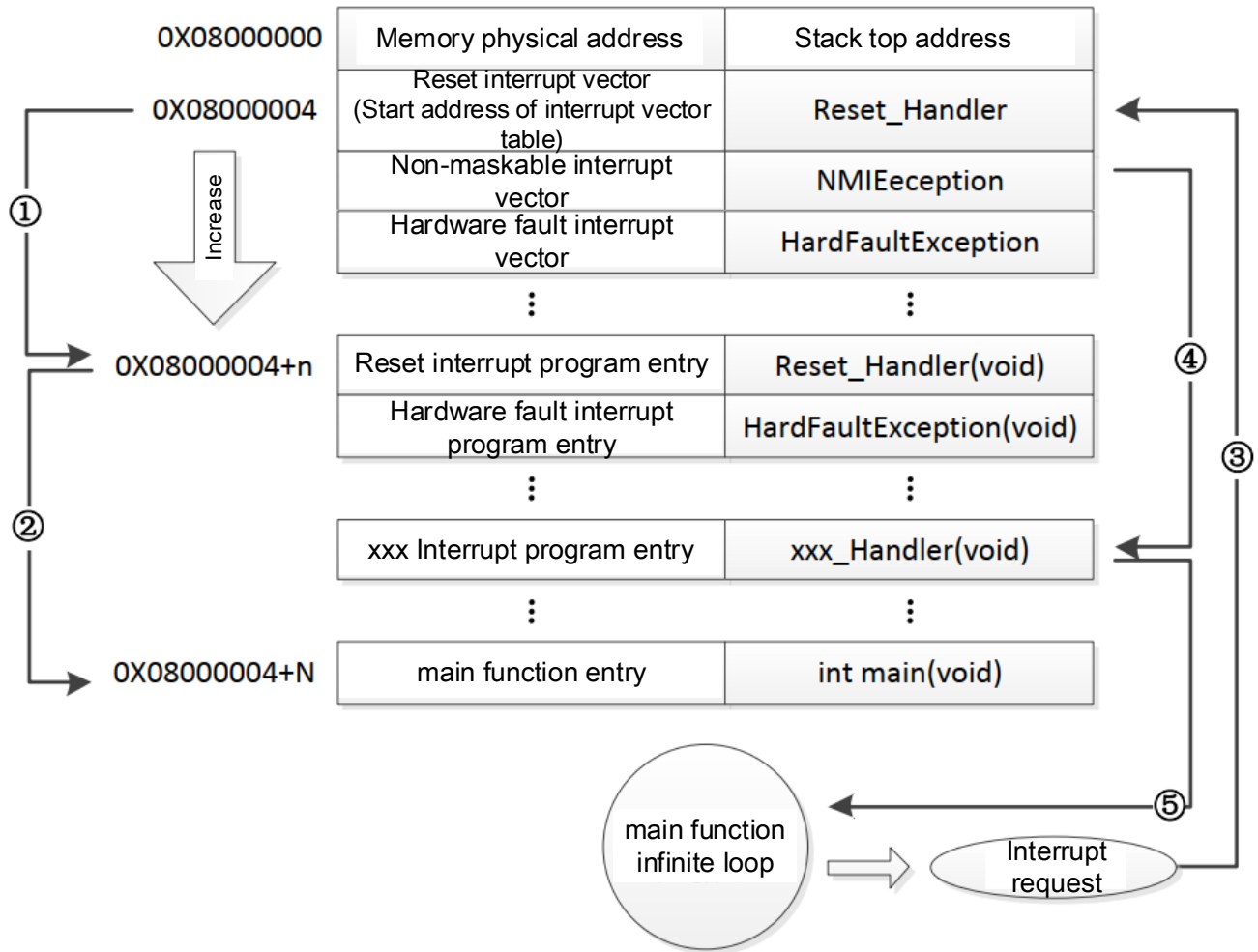


Figure 1.0

As shown in the figure above, the N32G45X's internal Flash address starts at 0x08000000, where program files are normally written. The N32G45X is a microcontroller based on the Cortex-M4F kernel, which internally responds to interrupts through an "interrupt vector table". After the program is started, it will first take out the reset interrupt vector from the "interrupt vector table" and execute the reset interrupt program to complete the startup. The starting address of this "interrupt vector table" is 0x08000004. When the interrupt comes, the internal hardware mechanism of N32G45X will automatically locate the PC pointer to the "interrupt vector table". According to the interrupt source, the corresponding interrupt vector is extracted to execute the interrupt service program.

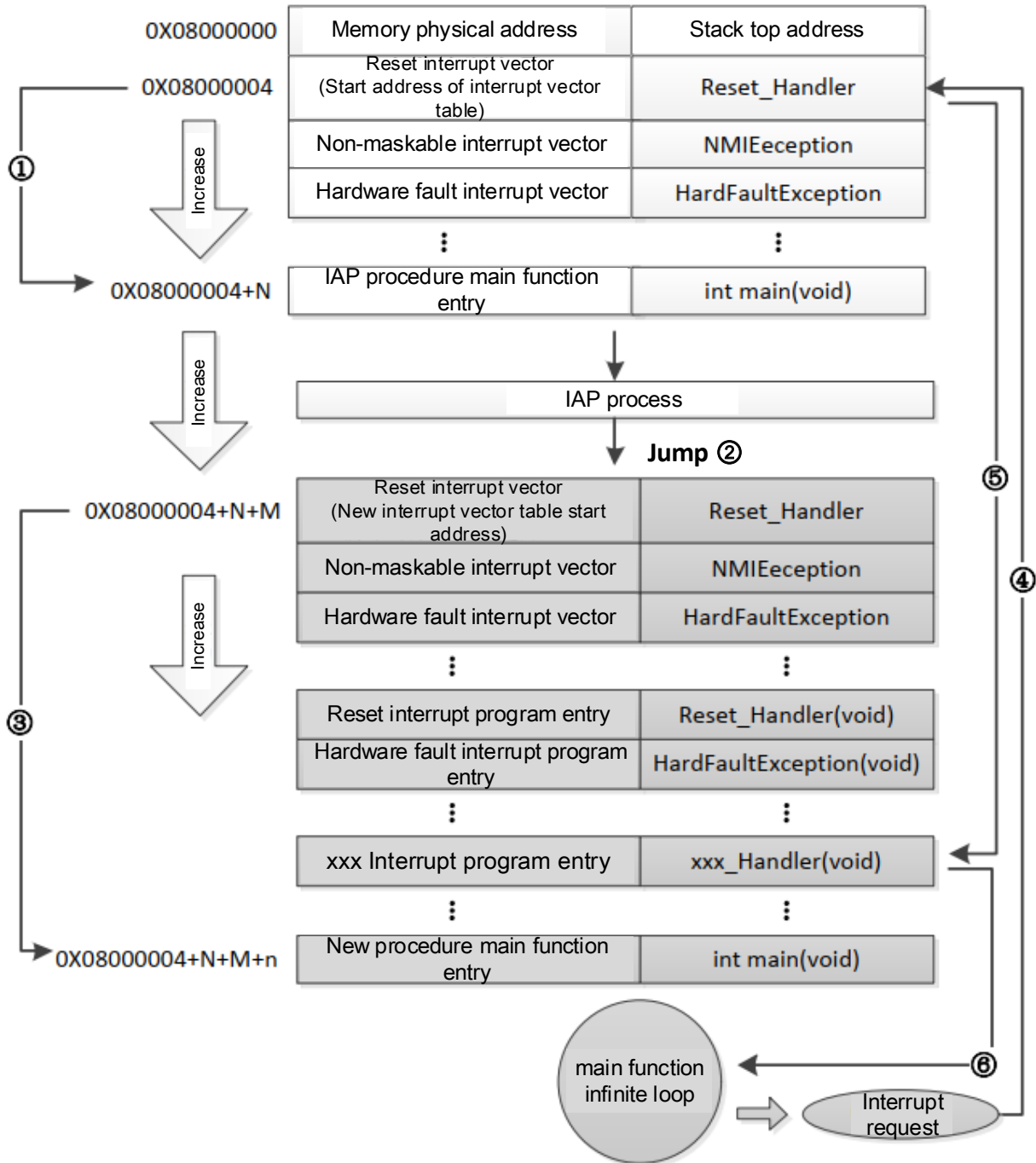


Figure 1.1

As shown in Figure 1.1, after powering on, the chip will extract the address of reset interrupt vector from the address  $0x08000004$  of Flash and jump to the interrupt reset function. After executing the interrupt reset function, the chip will jump to the MIAN function of IAP and start executing. When the main function is waiting for the upgrade, users can upgrade the APP by sending the update file through USB or UART. During the upgrade process, users can update while receiving, or update after receiving the whole package of APP. Since Flash and sram reserved by bootloader are relatively small, the example of this application note will upgrade APP by subcontracting sending and receiving while updating.

After the APP is updated, the program jumps to the reset vector table of the newly written program, takes out the address of the reset interrupt vector of the new program, jumps to the reset interrupt service program of the new program, and then jumps to the main function of the APP program, as shown in the figure (2) and (3). Similarly, main function is an infinite loop. And notice that N32G45X Flash has two interrupt vector tables in different positions.

During the execution of main function, if the CPU gets an interrupt request, the PC pointer still forcibly jumps to address 0X08000004 instead of the interrupt vector table of the new program, as shown in figure 4. The program then jumps to the new interrupt service program corresponding to the interrupt source according to the offset of the interrupt vector table set by us, as shown in figure 5. After executing the interrupt service program, the program returns the main function to continue running, as shown by ⑥ in the figure. The start address of the reset interrupt vector of the new program is  $0X08000004+N+M$ , where M is the jump offset of the new program. Subsequent chapters will explain how to set the offset in engineering.

## 2 IAP software implementation process

Through the analysis of the above two processes, we know that an IAP application must meet two requirements:

- 1) The new program must start at some address with offset X after the IAP program;
- 2) The interrupt of the new program must be moved to the corresponding table with an offset of X;

### 2.1 Set the start ADDRESS of the APP program

#### 2.1.1 SRAM\_APP Set the start ADDRESS

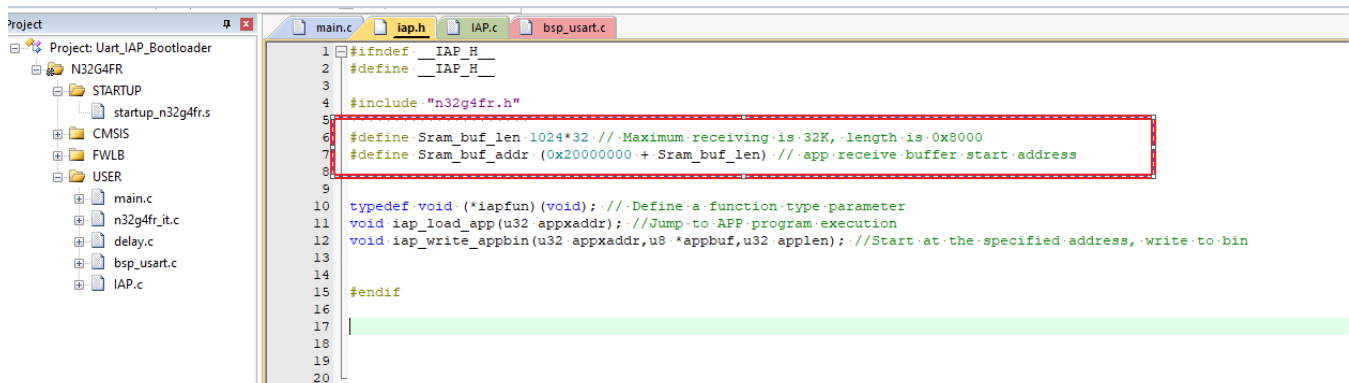


Figure 2.0

As shown in Figure 2.0, in the Bootloader project, the APP array SrAm\_buf was defined, starting at 0x20008000 and 1024\*32 in length (32K).

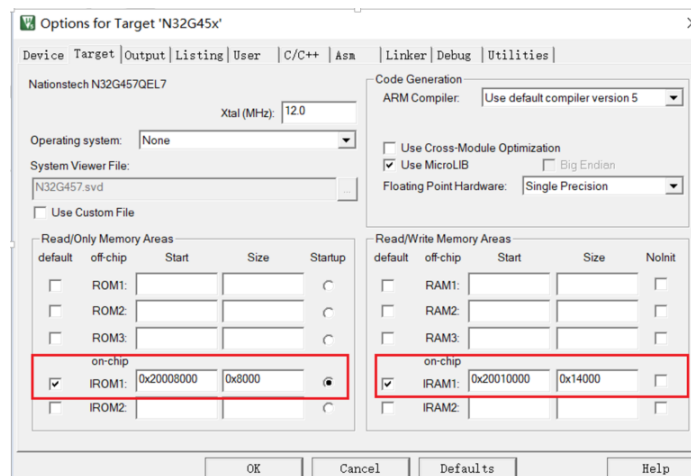


Figure 2.1

Since the SRAM of N32G45X starts at 0x20000000 and ends at 0x20024000, the size of the entire SRAM is 144K. So in the SRAM\_APP project, set the offset as shown in Figure 2.1: Click "magic wand", select "Target", enter Start 0x20008000, Size 0x8000 in the IROM1 column; Enter 0x20010000 for Start and 0x14000 for Size in the IRAM1 column.

Therefore, the resource layout of the entire SRAM is as follows: the first 32K is allocated to the Bootloader, the next 32K is used to store APP programs, and the next 80K is allocated to APP program calls.

### 2.1.2 FLASH\_APP start address setting

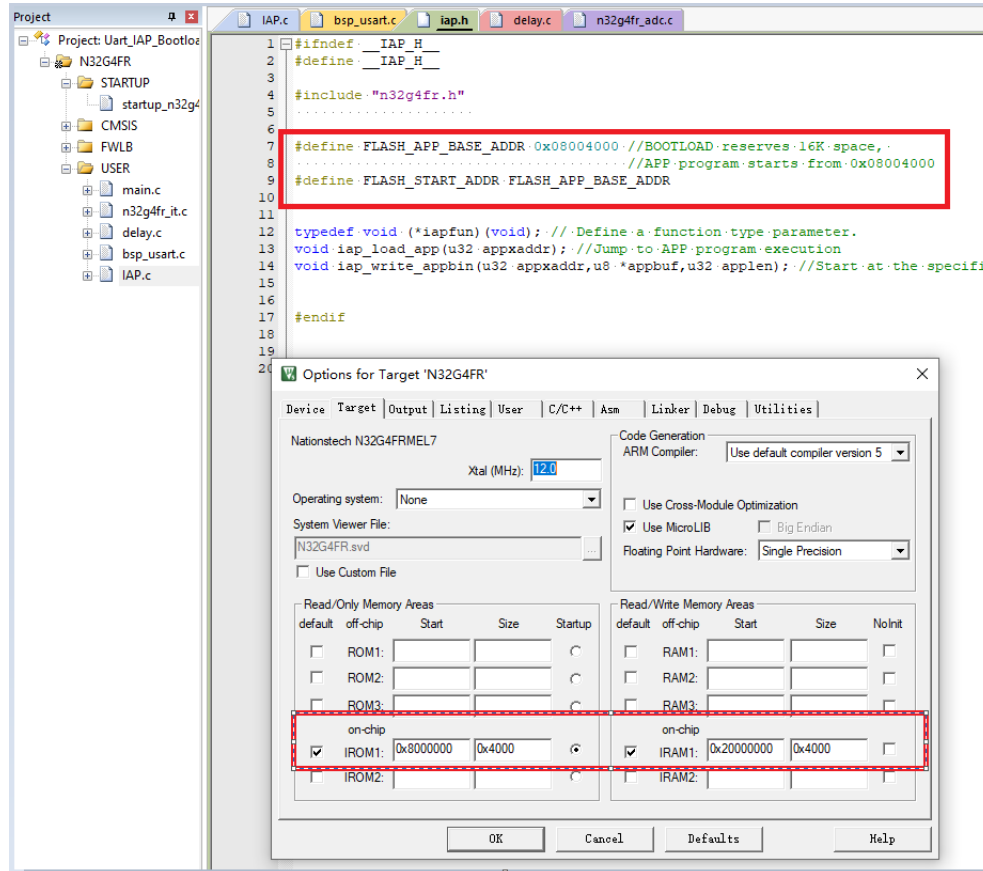


Figure 2.2

As shown in Figure 2.2, in the Bootloader project corresponding to Flash\_App, 16K Flash and 16K Sram are reserved for the small amount of code about 13K, and the Flash jump address 0x0800400 is set. Click "magic wand", select "Target", in the IROM1 column "Start" fill 0x08000000, Size fill 0x4000; Enter 0x20000000 for Start and 0x4000 for Size in the IRAM1 column.



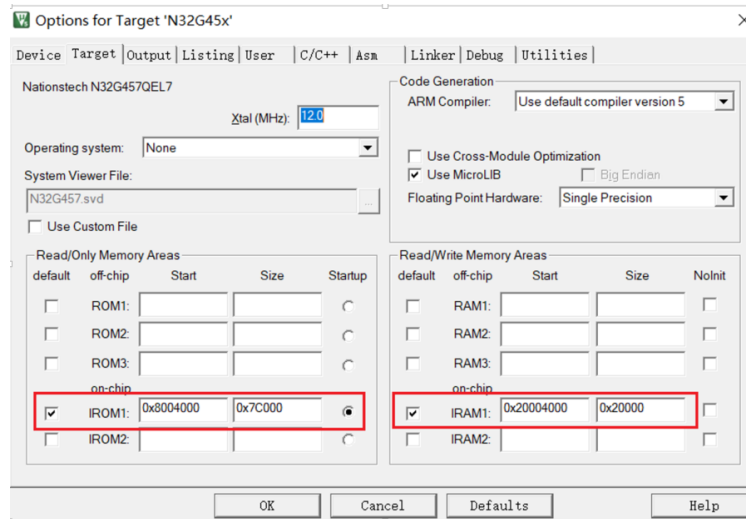


Figure 2.3

N32G45X has a maximum of 512K Flash, from address 0x08000000 to 0x08080000. The routine uses the first 16K Flash as the Bootloader, and the second 496K Flash is reserved for APP use. As shown in Figure 2.3, in Flash\_App project, click "magic wand", select "Target", fill 0x08004000 in "Start" of IROM1, and fill 0x7C000 in Size; In the IRAM1 column, enter Start 0x20004000, Size 0x20000.

## 2.2 Interrupt vector table offset setting method

When the system starts up, the systemInit function is called first to initialize the clock, and the systemInit function also completes the interrupt vector table.

```
#ifndef VECT_TAB_SRAM
```

```
SCB->VTOR = SRAM_BASE | VECT_TAB_OFFSET; /* Vector Table Relocation in Internal SRAM. */
```

```
#else
```

```
SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET; /* Vector Table Relocation in Internal FLASH. */
```

```
#endif
```

It can be understood from the code that the VTOR register stores the start address of the interrupt vector table.

VECT\_TAB\_SRAM is not defined by default, so perform SCB -> VTOR = FLASH\_BASE | VECT\_TAB\_OFFSET; For FLASH APP, we set it to FLASH\_BASE+ offset 0x4000, so we can add the following code before jumping to the main function of FLASH APP to reset the start address of the interrupt vector table:

```
SCB->VTOR = FLASH_BASE | 0x4000;
```

The above is the case of FLASH APP. When using SRAM APP, we set the start address as:

SRAM\_base+0x8000, same method, before jumping to the main function of SRAM APP, add the following code:

```
SCB->VTOR = SRAM_BASE | 0x8000;
```

This completes the setting of the interrupt offset to the table

## 2.3 In APP project, generate bin file

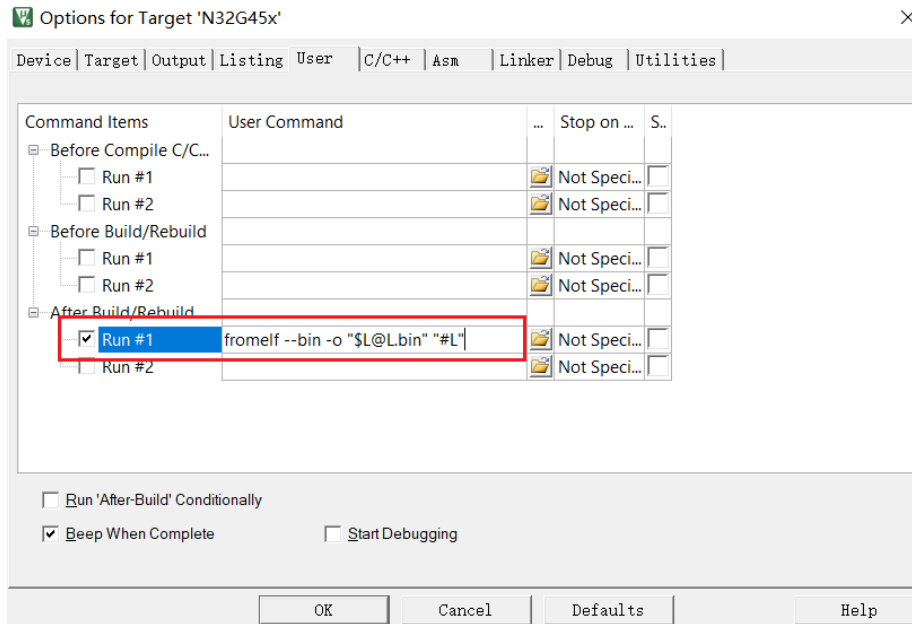


Figure 2.4

In the Sram\_App and Flash\_App projects, click "Magic Wand" and select "USER". Under "After Build/Rebuild", tick the box to the left of "RUN #1", and fill in "fromelf --bin -o \"\$L@L.bin\" \"#L\"\" in the right column, click OK After recompiling, the BIN file can be generated, and the BIN file is saved in the \\MDK-ARM\\Objects directory.

## 2.4 Software implementation process

The software process of Bootloader mainly consists of three steps:

- 1) Power on and initialize the serial port to determine whether the bin file of the App is waiting to receive.
- 2) Subcontracting receives the bin file and dumps the contents to Sram\_buf at the specified address, or writes to the specified Flash address;
- 3) Receiving bin file is complete, the program jumps;



Upgrade flow figure

## 2.4.1 Bootloader process of Sram\_App

Open Uart\_IAP\_Bootloader project, we can see that the program is mainly in main.c, IAP.c, bsp\_usart.c. The code for the three steps is detailed below.

```
int main(void)
{
    tim3_init(99, 71); //72MH/(71+1)=1M Hz; 1M Hz/(99+1)=100us
    USART_Config();
    printf("NZ3601_init success! \r\n");
    while(1)
    {
        while(receive_app_done == 0) //No APP program, waiting to receive updates
        {
            if(f_final_frame == 1)
            {
                receive_app_done = 1; //After receiving the BIN upgrade file
                m_delay_ms(500);
                break;
            }
        }

        if(receive_app_done) //App has been updated
        {
            receive_app_done = 0;
            TIM_Enable(TIM3, DISABLE); //Turn off timer interrupt
            //
            printf("APP address:%x\r\n", Sram_buf_addr);
            printf("Start to execute SRAM user code!!\r\n");

            SCB->VTOR = SRAM_BASE | 0x8000; //Set up interrupt vector table before jump
            iap_load_app(Sram_buf_addr); //Jump to the start address of the APP, during which it cannot be interrupted by other interrupts, otherwise the jump will fail
        }
    }
}
```

1. Waiting to receive the bin file

3. After the reception is completed, the program jumps

Figure 2.5

```
void USART1_IRQHandler(void)
{
    uint8_t i = 0;
    uint8_t buf_temp[256] = {0};
    uint8_t sum_check = 0;
    //
    if(USART_GetFlagStatus(DEBUG_USARTx, USART_INT_RXDNE) != RESET)
    {
        USART_ClearIntPendingBit(DEBUG_USARTx, USART_INT_RXDNE);
        if(receive_cnt <= 134)
        {
            RX_buf[receive_cnt++] = USART_ReceiveData(DEBUG_USARTx);
            current_pack_length = RX_buf[3]+5; //Calculate the data length of the current pack
            if((RX_buf[0] == 0x01)&&(RX_buf[1] == 0x01)&&(receive_cnt== current_pack_length)) //Frame header is fixed to 0x01, 0x01.
            {
                //pack length is fixed to uart_rx_buf[3] + 5 bytes,
                receive_cnt = 0; //The maximum length is 128+5 bytes
                f_receive_frame = 1;
                memcpy(buf_temp, RX_buf, 256);
                for(i = 0; i < current_pack_length -1; i++)
                {
                    sum_check = sum_check + buf_temp[i]; //Calculate SUM check
                }
                sum_check = ~sum_check + 1;
                if(sum_check == buf_temp[current_pack_length-1]) //Compare SUM, if it is different, discard the current packet and wait for the host computer to resend
                {
                    send_ack(); //Respond to the host computer
                    memcpy(&Sram_buf[rx_number*128], &RX_buf[4], current_pack_length-5); //Data transfer to Sram_buf
                    rx_number++;
                    if((current_pack_length==5)&&(RX_buf[3]==0)) //After sending the last packet of bin content, the host computer will send a 5-byte frame end
                    {
                        rx_number = 0;
                        f_final_frame = 1; //After receiving the last frame of data
                        current_pack_length = 0;
                    }
                }
                memset(RX_buf, 0x00, sizeof(RX_buf));
            }
        }
    }
}
```

Subcontract to receive bin file and verify

2. Cache the bin file to Sram and return ACK

Figure 2.6

As shown in Figure 2.5 and Figure 2.6, in main function, after initialization, there are two while(1) loops, waiting for receiving and jumping to Sram execution program respectively; BIN upgrade file is received in the serial port interrupt USART1\_IRQHandler(void) function. In order to minimize the use of Bootloader resources, be compatible with receiving large BIN files and ensure the integrity of BIN files, we split BIN into small packages for sending, sending 128 bytes each time. Therefore, after receiving a packet of data, we will verify it according to the transmission

protocol. If the verification fails, the current packet is discarded and the host computer resends the current packet. Transport protocols are described in detail in a later section.

```
asm void MSR_MSP(u32 addr)
{
    MSR_MSP, r0 //set Main Stack value
    BX r14
}

/**=====
    APP jump
    appaddr: The starting address of the user code.
=====*/
void iap_load_app(u32 appxaddr)
{
    if(((* (vu32*) appxaddr) & 0x0FFFFFFF) < 1024*144) // Check whether the top address of the stack is legal.
    {
        jump2app = (iapfun) * (vu32*) (appxaddr+4);
        MSR_MSP(* (vu32*) appxaddr); // initialize the stack pointer
        jump2app(); // Jump to APP.
    }
}

/**=====
=====*/
```

set offset and jump

Figure 2.7

As shown in Figure 2.7, after receiving the BIN file in its full form, we jump to iap\_load\_app(Sram\_buf\_addr) in Figure 2.5; Sram\_buf\_addr is the starting Sram\_buf address 0x20008000 that we set in Figure 2.0.

## 2.4.2 Flash\_App Bootloader program flow

```
int main(void)
{
    tim3_init(99, 71); //72MH/(71+1)=1M.Hz; 1M.Hz/(99+1)=100us
    USART_Config();
    printf("M23601_init success!\r\n");
    while(1)
    {
        if(FLASH_ReadWord(app_update_flag_addr) == 0x12345678) //Whether the power-on detection needs to jump directly
        {
            receive_app_done = 1;
            while(receive_app_done == 0) //No APP program, waiting to receive updates
            {
                if(iap_flashing == 1)
                {
                    TIM_Enable(TIM3, DISABLE);
                    USART_Enable(DEBUG_USARTx, DISABLE);
                    //
                    IAP_UPDATE_APP(); //Update the received pack package
                    f_iap_flashing = 0;
                    f_receive_frame = 0; //Clear the receive frame flag
                    if(f_final_frame == 1)
                    {
                        f_final_frame = 0;
                        receive_app_done = 1; //Update is complete
                        app_flag_write(0x12345678, app_update_flag_addr); //Write IAP upgrade flag
                    }
                    TIM_Enable(TIM3, ENABLE);
                    USART_Enable(DEBUG_USARTx, ENABLE);
                }
            }
            if(receive_app_done) //App has been updated
            {
                receive_app_done = 0;
                TIM_Enable(TIM3, DISABLE); //Turn off timer interrupt
                printf("APP address: %x\r\n", (FLASH_START_ADDR));
                printf("Start to execute Flash user code!\r\n");
                iap_load_app(FLASH_START_ADDR); //Jump to the start address of the APP, during which it cannot be interrupted by other interrupts, otherwise the jump will fail
            }
        }
    }
}
```

Figure 2.8

As shown in Figure 2.8, in the main() function, the program will determine whether it needs to jump directly after initialization, because the Flash program will not be lost in power down and can be maintained all the time after the update, but the data will be lost after Sram power down, so there is no such judgment. If the program has not been

updated outside the Bootloader area, it will wait for the serial port to receive the BIN file for update. Since the Flash page of N32G45X is 2K, in order to avoid too much address judgment, the example is to write the Flash once after receiving the packet of 2K size. It can avoid occupying too much Sram resources. After the BIN packet of the last frame is written, a flag will be written into the Flash, and the next power-on will directly jump to the APP program.

```
void USART1_IRQHandler(void)
{
    uint8_t i = 0;
    uint8_t buf_temp[256] = {0};
    uint8_t sum_check = 0;
    //
    if(USART_GetFlagStatus(USART1, USART_FLAG_RXNE) != RESET)
    {
        USART_ClearPendingBit(USART1, USART_FLAG_RXNE);
        slot_timer = 0;
        if(receive_cnt <= 134)
        {
            RX_buf[receive_cnt++] = USART_ReceiveData(USART1);
            current_pack_length = RX_buf[3]+5;
            if((RX_buf[0] == 0x01)&&(RX_buf[1] == 0x01)&&(receive_cnt== current_pack_length))
            {
                //Calculate the data length of the current pack
                //Frame header is fixed to 0x01, 0x01
                //pack length is fixed to uart_rx_buf[3] + 5 bytes
                //Maximum 128+5 bytes
                receive_cnt = 0;
                f_receive_frame = 1;
                memcpy(buf_temp, RX_buf, 256);
                for(i = 0; i < current_pack_length -1; i++)
                {
                    sum_check = sum_check + buf_temp[i];
                }
                sum_check = ~sum_check + 1;
                if((sum_check == buf_temp[current_pack_length-1])&&(f_IAP_flashing==0))
                {
                    //Compare SUM, if flash is being written, discard the current packet, and wait for the host computer to resend
                    send_ack();
                    memcpy(&flash_buf[rx_number*128], &RX_buf[4], current_pack_length-5);
                    rx_number += 4;
                    //Respond to the host computer
                    //Dump data to flash_buf
                    if(rx_number >= 16)
                    {
                        //After receiving 16 times for a total of 2K, write a flash
                        rx_number = 0;
                        f_IAP_flashing = 1;
                        f_IAP_start = 1;
                    }
                    else if((current_pack_length==5)&&(RX_buf[3]==0))
                    {
                        //After sending the last packet of bin content, the host computer will send a 5-byte frame end
                        rx_number = 0;
                        f_IAP_flashing = 1;
                        f_final_frame = 1;
                    }
                    current_pack_length = 0;
                }
            }
            memset(RX_buf, 0x00, sizeof(RX_buf));
        }
    }
}
```

Figure 2.9

As shown in Figure 2.9, Flash reception is slightly different from Sram reception in that the Flash Bootloader defines a 2K cache BUF that will be written to Flash after 2K reception.

```
/*=====
//Upgrade APP
=====*/

void IAP_UPDATE_APP(void)
{
    ready_write_addr = FLASH_APP_BASE_ADDR + pages_number*2048;
    //
    while(app_flash_write((uint32_t *)flash_buf, ready_write_addr)); //IAP upgrade 2K each time
    //
    memset(flash_buf, 0x00, 2048);
    pages_number++;
}

//0x08004000
FLASH_APP_BASE_ADDR
```

Figure 2.10

As shown in Figure 2.10, IAP\_UPDATE\_APP(void) is called once each time after 2K or the last frame of data packet is received. The starting address FLASH\_APP\_BASE\_ADDR is 0x08004000.

### 3 Download validation

#### 3.1 Upper computer transmission protocol

The upper computer tool used for verification is XCOM V2.6, whose transmission protocol has a frame header of 2 bytes and can be flexibly configured. It supports ACK response and subcontracting to send BIN files. The maximum length of each packet is 255 bytes, and it has SUM, CRC16 and other verification methods.

| Protocol format | Frame head 1 | Frame head 2 | Frame number | Length of the frame | Data   | Data   | Data   | Data   | Checksum |
|-----------------|--------------|--------------|--------------|---------------------|--------|--------|--------|--------|----------|
|                 | 0x01         | 0x01         | n            | length              | Data 0 | Data 1 | Data 2 | Data n | SUM      |

The protocol is a frame header whose first four bytes are 2 bytes, the current frame number, and the frame length. The frame header can be set at will. When the frame number exceeds 255, it will continue to increase from 0. The protocol frame header of the routine is 0x01, and the frame length is 0x80. The SUM mode is checked and selected. After the frame sequence is increased to 255, the next frame will be counted from 0.

| ACK format | Frame head 1 | Frame head 2 | Frame number | Length of the frame | Checksum |
|------------|--------------|--------------|--------------|---------------------|----------|
|            | 0x01         | 0x01         | n            | 0                   | SUM      |

After receiving a complete packet, the chip will reply the host computer with an ACK signal. If no ACK is replied, the host computer will send the packet of the current frame repeatedly.

## 3.2 Procedure download the BIN file

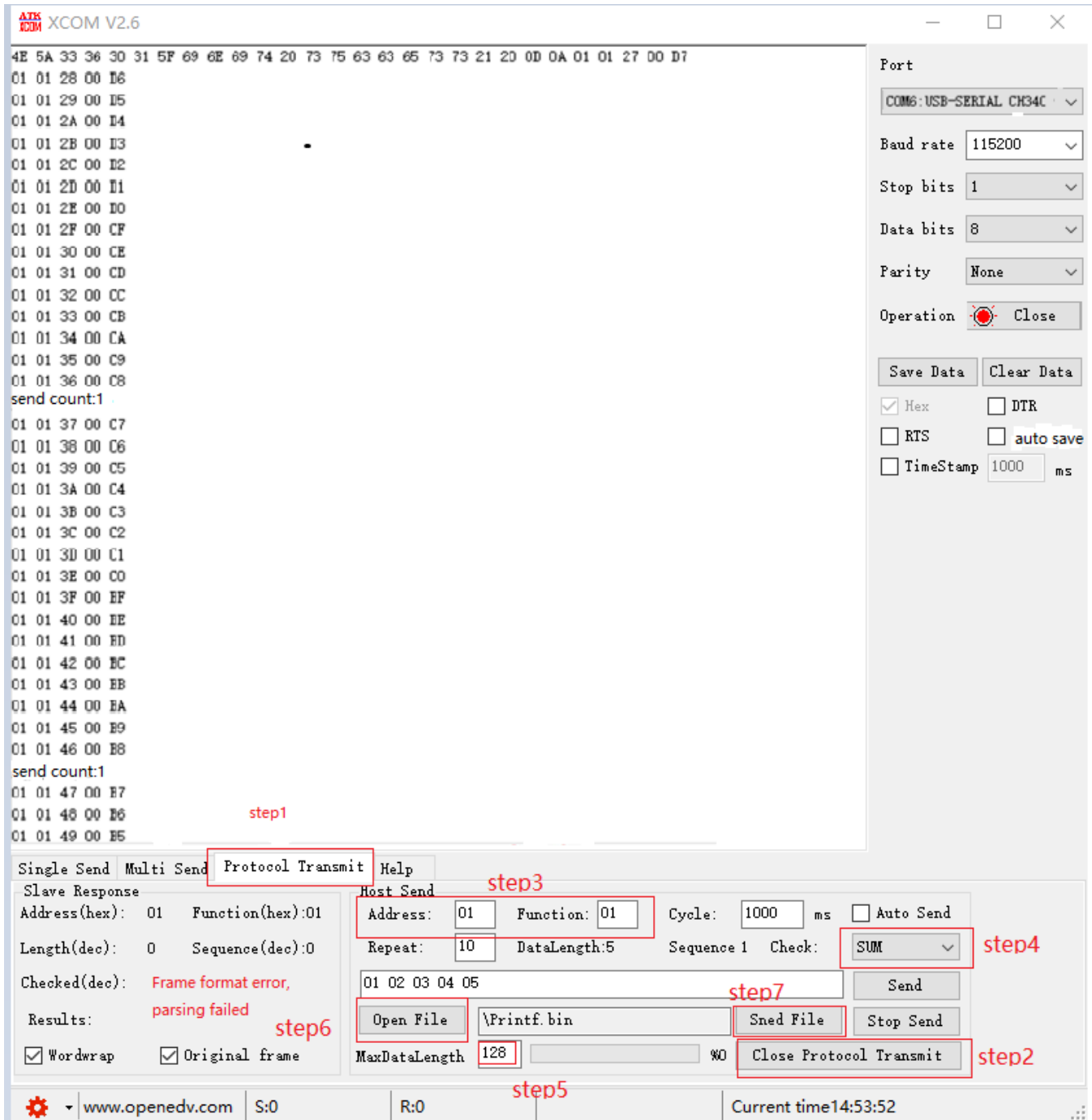


Figure 3.0

As shown in Figure 3.0, there are 7 steps to deliver BIN file through the host computer:

Step1: open XCOM V2.6 and select "protocol transfer";

Step2: click "open protocol transmission";



Step3: configure a 2-byte frame header and fill it with 0x01;

Step4: Select SUM as the test method;

Step5: Set the frame length to 128;

Step6: Open the selection BIN file;

Step7: Click send file;

### 3.3 Validation

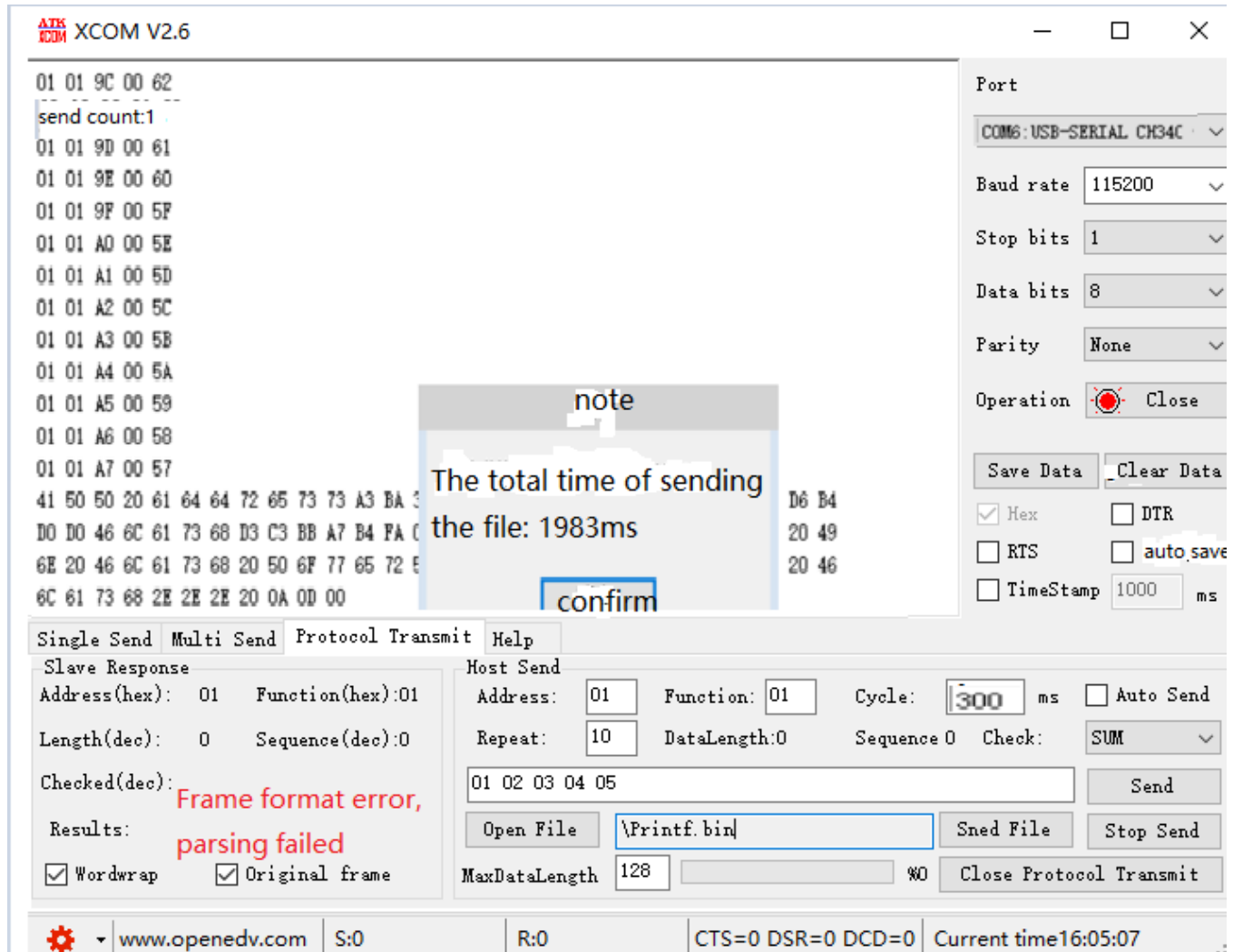


Figure 3.1

As shown in Figure 3.1, after the successful sending, the message "file has been sent, total time: XXXX ms" will be displayed.

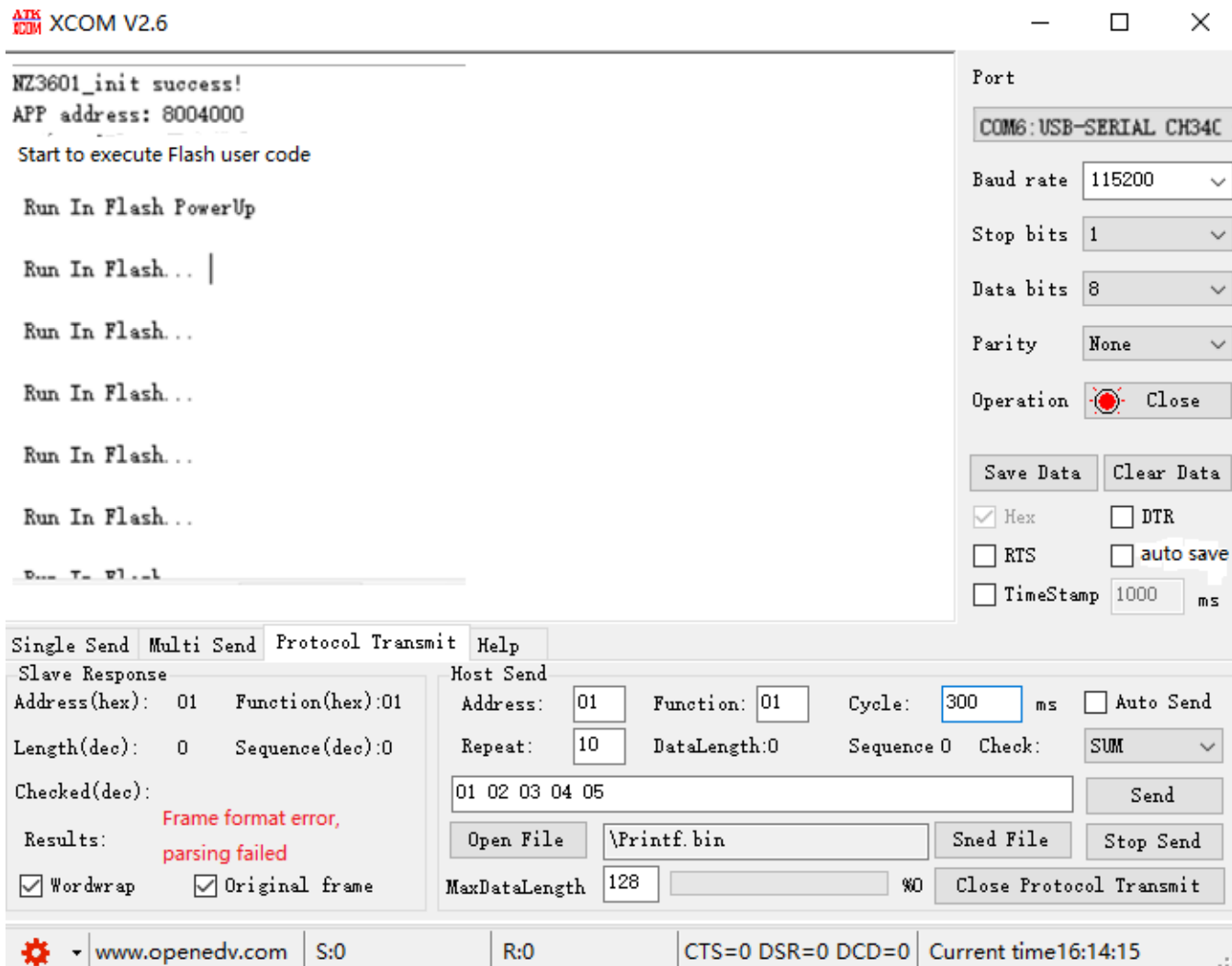


Figure 3.2

As shown in Figure 3.2, after initialization, the program jumps to APP\_address: 0x08004000 to start executing the FLASH program.

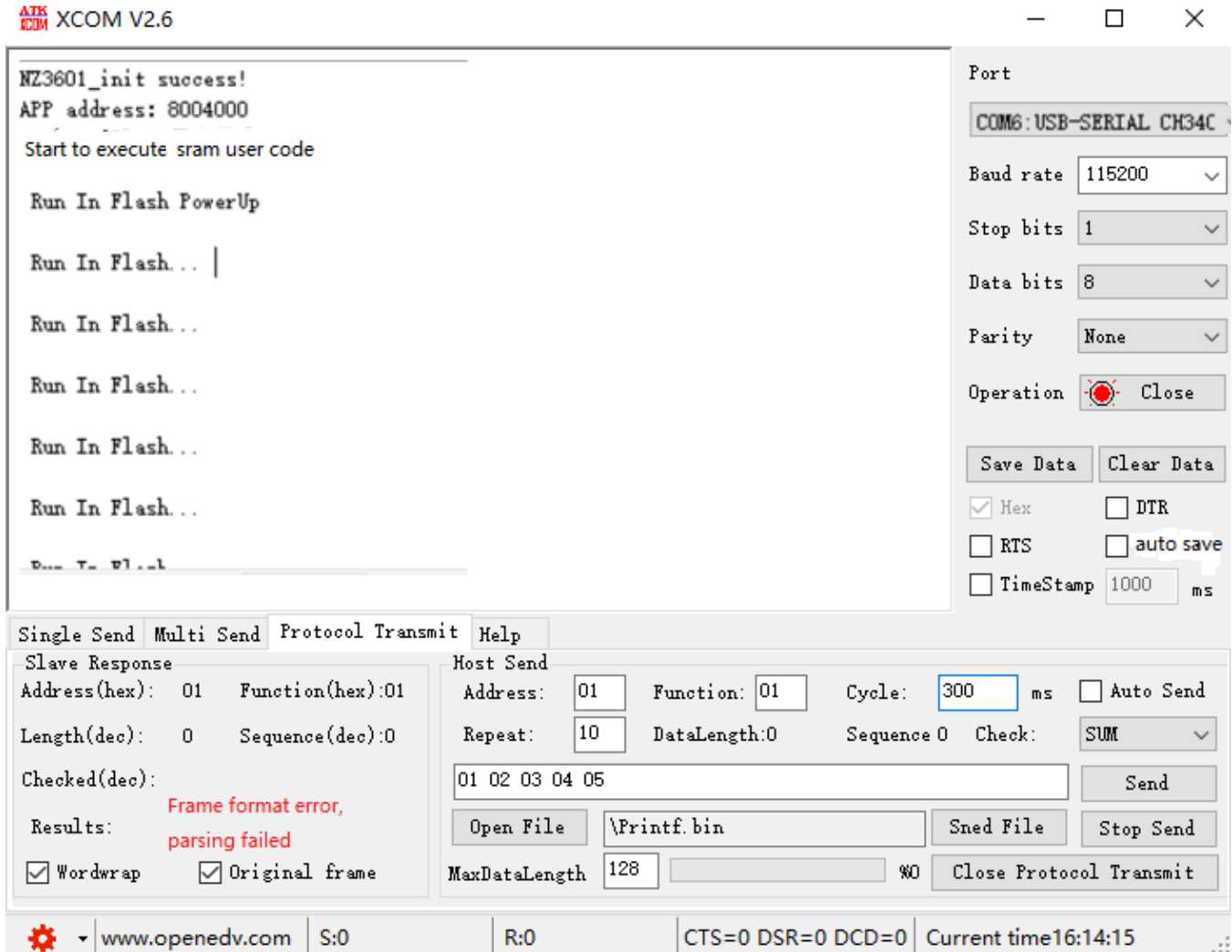


Figure 3.3

As figure 3.3 shows, after receiving the BIN file, the program successfully jumped to APP\_address: 0x20008000 to execute the code in SRAM.

## 4 Q&A

1. Q: The BIN file cannot be received, and the verification fails.

A: Check whether the baud rate is consistent and whether SUM is selected as the verification mode.

2. Q: The APP fails to jump;

A: Check whether the address set by the project is consistent with the address jump above the program; At the same time, all interrupts are closed before the jump.

## 5 Version history

| Version | Date     | Note                           |
|---------|----------|--------------------------------|
| V1.0    | 2020.9.2 | New document                   |
| V1.1    | 2021.7.1 | Added IAP software flow figure |

## 6 Notice

This document is the exclusive property of Nations Technologies Inc. (Hereinafter referred to as NATIONS). This document, and the product of NATIONS described herein (Hereinafter referred to as the Product) are owned by NATIONS under the laws and treaties of the People's Republic of China and other applicable jurisdictions worldwide. NATIONS does not grant any license under its patents, copyrights, trademarks, or other intellectual property rights. Names and brands of third party may be mentioned or referred thereto (if any) for identification purposes only. NATIONS reserves the right to make changes, corrections, enhancements, modifications, and improvements to this document at any time without notice. Please contact NATIONS and obtain the latest version of this document before placing orders.

Although NATIONS has attempted to provide accurate and reliable information, NATIONS assumes no responsibility for the accuracy and reliability of this document.

It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. In no event shall NATIONS be liable for any direct, indirect, incidental, special, exemplary, or consequential damages arising in any way out of the use of this document or the Product.

NATIONS Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at user's risk. User shall indemnify NATIONS and hold NATIONS harmless from and against all claims, costs, damages, and other liabilities, arising from or related to any customer's Insecure Usage. Any express or implied warranty with regard to this document or the Product, including, but not limited to, the warranties of merchantability, fitness for a particular purpose and non-infringement are disclaimed to the fullest extent permitted by law.

Unless otherwise explicitly permitted by NATIONS, anyone may not use, duplicate, modify, transcribe or otherwise distribute this document for any purposes, in whole or in part.