N9H26 SPI Loader Reference Guide

Document Information

Abstract	Introduce the steps to build and launch SPI Loader for the N9H26 series microprocessor (MPU).
Apply to	N9H26 series

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

Table of Contents

1 Introduction

The SPI loader is a firmware stored at the SPI Flash chip for booting purpose. It will set the system clock, initialize the relevant modules, and then load the next firmware to DRAM to execute.

The SPI loader supports the following features:

- Initialize more modules such as SPU, RTC, and so on.
- Load Logo image to DRAM if it existed at SPI Flash chip.
- Load next firmware to DRAM if it existed at SPI Flash chip.
- Execute next firmware. Normally, it should be application code.

Nuvoton provides SPI loader source code within the N9H26 series microprocessor (MPU) BSP.

2 SPI Loader BSP Directory Structure

This chapter introduces the SPI loader related files and directories in the N9H26 BSP. SPI loader provide gzip decompression function.

2.1 Loaders\SPILoader

GCC	The GCC project files for the SPI loader.
KEIL	The KEIL project files for the SPI loader.
main.c	The main function for the SPI loader.
scat.scf	The scatter file for the SPI loader
SpiRead.c	SPI Flash driver of N9H26.
Rootkey.c	Root key Generator for authentication Flash
Other files	System driver of N9H26.

2.2 Loaders\SPILoader_gzip

GCC	The GCC project files for the SPI loader.
KEIL	The KEIL project files for the SPI loader.
ziplib	gzip library
cmd_bootm.c	gzip APIs
main.c	The main function for the SPI loader.
scat.scf	The scatter file for the SPI loader
SpiRead.c	SPI Flash driver of N9H26.
Rootkey.c	Root key Generator for authentication Flash
Other files	System driver of N9H26.

2.3 Loaders\Binary

N9H26_SpiLoader_xxx.bin	The binary file of the SPI loader for different project targets.

3 SPI Loader Source Code

Complete source codes are included in the *N9H26 BSP Loaders\SPILoader & Loaders\SPILoader_gzip* directories:

3.1 Development Environment

Keil IDE and Eclipse are used as Non-OS BSP development environment, which uses J-Link ICE or ULINK2 ICE (optional) for debugging. This document uses Keil IDE to describe the project structure. To support ARM9, MDK Plus or Professional edition shall be used.

Note that Keil IDE and ICE need to be purchased from vendor sources.

Feature	MDK Edition			
	Professional	Plus	Essential	Lite
	All-in-one solution including Middleware	Supports all microcontroller cores and Middleware	Supports selected Cortex-M	Free with code size limit: 32 KBytes
Device Support				
Arm Cortex-M0/M0+/M3/M4/M7	✔	✔	✔	✔
Arm Cortex-M23/M33 Non-secure only	✔	✔	✔	✘
Arm Cortex-M23/M33 Secure and non-secure	✔	✔	✘	✘
Armv8-M Architecture Models including FastModel	✔	✘	✘	✘
Arm SecurCore®	✔	✔	✘	✘
Arm7™, Arm9™, Arm Cortex-R4	✔	✔	✘	✘

Figure 3-1 Keil MDK License Chart

3.2 Project Structure

The SPI loader project includes one main function file and some driver files of N9H26. Please note that the binary code size of the SPI loader MUST less than 63 KB.

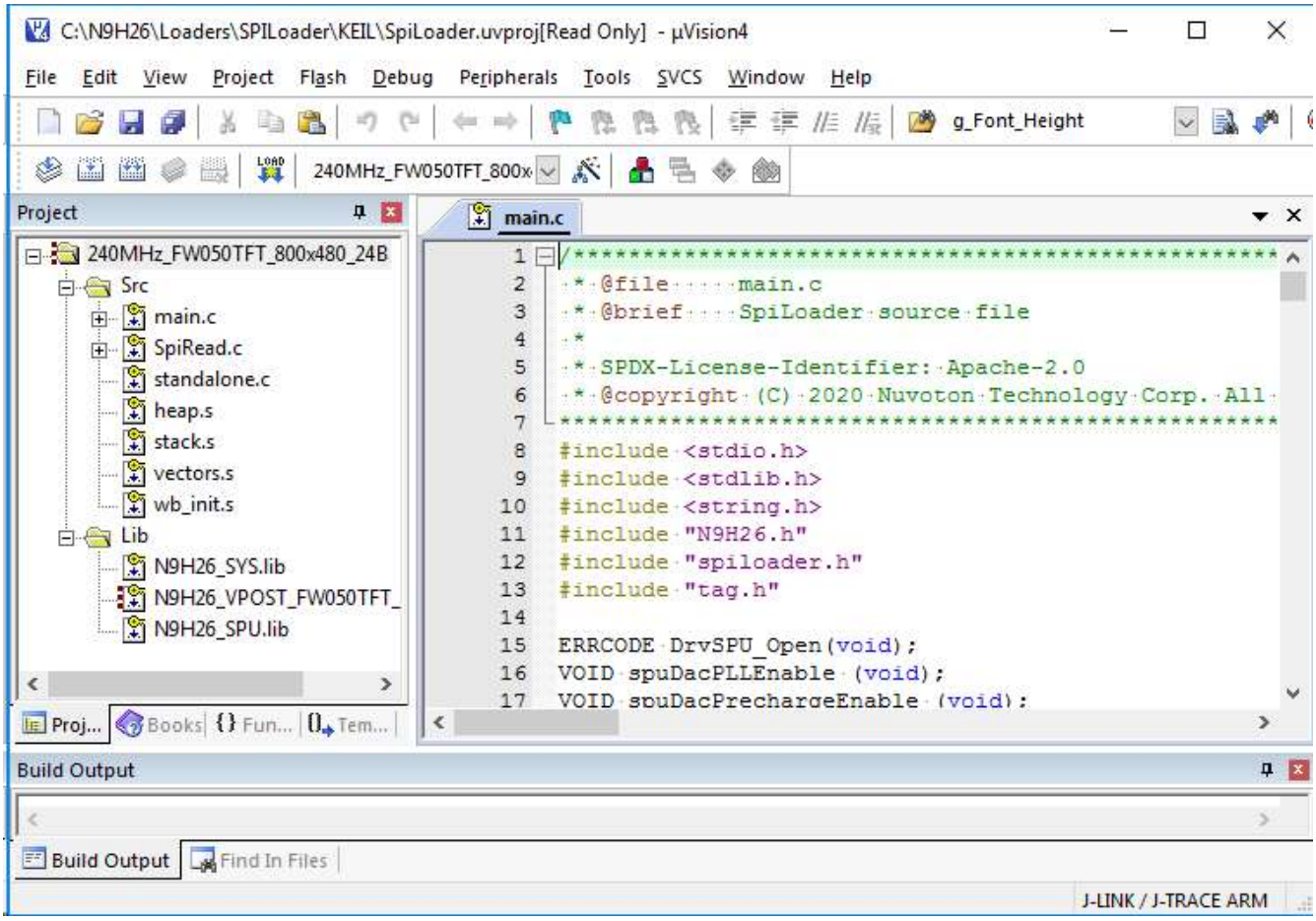

Figure 3-2 SPI Loader Project Tree on Keil MDK

The SPI loader project includes some targets that can be used in different situations.

- **240MHz_NoLCM**: Set system core clock to 240MHz.
- **240MHz_FW050TFT_800x480_24B**: Set system core clock to 240MHz and initialize VPOST for Giant Plus WVGA panel. This is the official standard target.
- **264MHz_NoLCM**: Set system core clock to 264MHz.
- **264MHz_FW050TFT_800x480_24B**: Set system core clock to 264MHz and initialize VPOST for Giant Plus WVGA panel.
- **Not_Set_Clock_NoLCM**: Not change system clock.
- **Not_Set_Clock_FW050TFT_800x480_24B**: Not change system clock and initialize VPOST for Giant Plus WVGA panel.

3.3 System Initialization

The system initialization code is located in main function, including system code clock setting, enable cache feature, and UART debug port setting. It also initializes some necessary peripherals during the boot process.

```
int main()
{
    unsigned int startBlock;
    unsigned int fileLen;
    unsigned int executeAddr;

    /* Omit some source code in document. */

    int count, i;
    void (*fw_func)(void);

    outp32(0xFF000000, 0);

    init();
    initClock();

    DrvSPU_Open();
    spuDacPLLEnable();
    spuDacPrechargeEnable();

#ifndef __No_LCM__
    initVPostShowLogo();
#else
    /* Turn on HCLK4_CKE clock control */
    outpw(REG_AHBCLK, inpw(REG_AHBCLK) | HCLK4_CKE);
#endif
    /* Omit some source code in document. */
}
```

3.4 RTC Function

RTC hardware power off function is enabled by default in SPI loader. User can modify *SpiLoader.h* to enable / disable RTC hardware power off function (__RTC_HW_PWOFF__) or entire RTC function (__No_RTC__).

```
#ifdef __No_RTC__
    sysprintf("* Not Config RTC\n");
    outp32(REG_APBCLK, inp32(REG_APBCLK) & ~RTC_CKE);/* disable RTC clock to save power */
#else
    #ifdef __RTC_HW_PWOFF__
    sysprintf("Enable HW Power Off\n");
    /* RTC H/W Power Off Function Configuration */
    RTC_Check();    /* waiting for RTC registers ready for access */
    outp32(PWRON, (inp32(PWRON) & ~PCLR_TIME) | 0x60005); /* Press Power Key during 6 sec
to Power off (0x'6'0005) */
    RTC_Check();
    outp32(RIIR, 0x4);
    RTC_Check();
    outp32(REG_APBCLK, inp32(REG_APBCLK) & ~RTC_CKE); /* disable RTC clock to save power
*/
    #else
    /* RTC H/W Power Off Function Configuration */
    RTC_Check();    /* waiting for RTC registers ready for access */
    outp32(PWRON, (inp32(PWRON) & ~PCLR_TIME) & ~0x4); /* Press Power Key during 6 sec to
Power off (0x'6'0005) */
    RTC_Check();
    outp32(RIIR, 0x4);
    RTC_Check();
    sysprintf("Disable HW Power Off - 0x%X\n",inp32(PWRON));
    outp32(REG_APBCLK, inp32(REG_APBCLK) & ~RTC_CKE); /* disable RTC clock to save power
*/
    #endif
#endif
```

3.5 SPI Flash Initialization

One of the major tasks of the SPI loader is to copy the next firmware on the SPI Flash to DRAM for execution. To initialize the SPI Flash driver, SPI_OpenSPI () must be called in source code.

```
    /* Initial SPI Flash interface */
    SPI_OpenSPI();
```

3.6 Security Function

The security function is to provide anti-copy function to prevent the reproduction (supported by W74M series – provide an authentication mechanism to ensure the physical authenticity of the attached flash devices). If user wants to use the security function, please build SPI loader by the project file name with security.

```
#ifdef __Security__
    if ((RPMC_ReadUID(u8UID)) == -1)
    {
        sysprintf("read id error !!\n");
        return -1;
    }

    sysprintf("SPI flash uid [0x%02X%02X%02X%02X%02X%02X%02X%02X]\n",u8UID[0],
u8UID[1],u8UID[2], u8UID[3],u8UID[4], u8UID[5],u8UID[6], u8UID[7]);

    /* first stage, initial rootkey */
    /* calculate ROOTKey with UID & ROOTKeyTag by SHA256 */
    RPMC_CreateRootKey((unsigned char *)u8UID,8, ROOTKey);

    /* Second stage, update HMACKey after ever power on. without update HMACkey, Gneiss
would not function */
    /* Get random data for HMAC message, it can also be serial number, RTC information and
so on. */
    HMACMessage[0] = rand()%0x100;
    HMACMessage[1] = rand()%0x100;
    HMACMessage[2] = rand()%0x100;
    HMACMessage[3] = rand()%0x100;

    /* Update HMAC key and get new HMACKey.
       HMACKey is generated by SW using Rootkey and HMACMessage.
       RPMC would also generate the same HMACKey by HW
     */
    RPMCStatus = RPMC_UpHMACkey(KEY_INDEX, ROOTKey, HMACMessage, HMACKey);
    if(RPMCStatus == 0x80)
    {
        /* update HMACkey success */
        sysprintf("RPMC_UpHMACkey Success - 0x%02X!!\n",RPMCStatus );
    }
    else
    {
        /* write HMACkey fail, check datasheet for the error bit */
```

```
        sysprintf("RPMC_UpHMACkey Fail - 0x%02X!!\n",RPMCStatus );
    }

    /* Third stage, increase RPMC counter */
    /* input tag is send in to RPMC, it could be time stamp, serial number and so on */
    for(i= 0; i<12;i++)
        Input_tag[i] = u8UID[i%8];

    RPMCStatus = RPMC_IncCounter(KEY_INDEX, HMACKey, Input_tag);
    if(RPMCStatus == 0x80)
    {
        /* increase counter success */
        sysprintf("RPMC_IncCounter Success - 0x%02X!!\n",RPMCStatus );
    }
    else
    {
        /* increase counter fail, check datasheet for the error bit */
        sysprintf("RPMC_IncCounter Fail - 0x%02X!!\n",RPMCStatus );
        while(1);
    }

    if(RPMC_Challenge(KEY_INDEX, HMACKey, Input_tag)!=0)
    {
        sysprintf("RPMC_Challenge Fail!!\n" );
        /* return signature miss-match */
        while(1);
    }
    else
        sysprintf("RPMC_Challenge Pass!!\n" );
#endif
```

3.7 Get Image Information Table

The location of Image Information Table always is offset 63 KB in the SPI Flash. SPI loader reads and parses it to find out all images.

```
    sysprintf("Load Image ");
    /* read image information */
#ifndef __OTP_4BIT__
    SPIReadFast(0, 63*1024, 1024, (UINT32*)imagebuf);  /* offset, len, address */
#else
```

```
    outpw(REG_GPEFUN1, (inpw(REG_GPEFUN1)  & ~(MF_GPE8 |MF_GPE9)) | 0x44);

    JEDEC_Probe();
    spiFlashFastReadQuads(63*1024, 1024, (UINT32*)imagebuf);  /* offset, len, address */
#endif
```

3.8 Load Image from SPI Flash to DRAM

The location of Image Information Table always is offset 63 KB in the SPI Flash. SPI loader reads and parses it to find out all images.

```
    if (((*(pImageList+0)) == 0xAA554257) && ((*(pImageList+3)) == 0x63594257))
    {
        count = *(pImageList+1);

        pImageList=((unsigned int*)(((unsigned int)image_buffer)|0x80000000));
        startBlock = fileLen = executeAddr = 0;

        /* load logo first */
        pImageList = pImageList+4;
        for (i=0; i<count; i++)
        {
            if (((*(pImageList) >> 16) & 0xffff) == 4)    /* logo */
            {
                startBlock = *(pImageList + 1) & 0xffff;
                executeAddr = *(pImageList + 2);
                fileLen = *(pImageList + 3);
#ifndef __OTP_4BIT__
                SPIReadFast(0, startBlock * 0x10000, fileLen, (UINT32*)executeAddr);
#else
                spiFlashFastReadQuads(startBlock * 0x10000, fileLen,
                                      (UINT32*)executeAddr);
#endif
                break;
            }
            /* pointer to next image */
            pImageList = pImageList+12;
        }

        /* Omit some source code in document. */
```

```
        pImageList=((unsigned int*)(((unsigned int)image_buffer)|0x80000000));
        startBlock = fileLen = executeAddr = 0;

        /* load execution file */
        pImageList = pImageList+4;
        for (i=0; i<count; i++)
        {
            if (((*(pImageList) >> 16) & 0xffff) == 1)    /* execute */
            {
                startBlock = *(pImageList + 1) & 0xffff;
                executeAddr = *(pImageList + 2);
                fileLen = *(pImageList + 3);
#ifndef __OTP_4BIT__
                SPIReadFast(0, startBlock * 0x10000, fileLen, (UINT32*)executeAddr);
#else
                spiFlashFastReadQuads(startBlock * 0x10000, fileLen,
                                      (UINT32*)executeAddr);
#endif

                /* Omit some source code in document. */

                /* JUMP to kernel */
                sysprintf("Jump to kernel\n\n");

                /* Omit some source code in document. */

                fw_func = (void(*)(void))(executeAddr);
                fw_func();
                break;
            }
            /* pointer to next image */
            pImageList = pImageList+12;
        }
    }
```

3.9 Load Image from SPI Flash to DRAM with gzip function

SPI lodaer with gzip will load first 64byte of execute image to check if there is an u-Boot header. The u-Boot image header is as follows. SPI loader only checks the fields ih_magic (0x56190527) and ih_comp (0x01).

```
typedef struct image_header {
        uint32_t        ih_magic;               /* Image Header Magic Number       */
        uint32_t        ih_hcrc;                /* Image Header CRC Checksum       */
        uint32_t        ih_time;                /* Image Creation Timestamp        */
        uint32_t        ih_size;                /* Image Data Size                 */
        uint32_t        ih_load;                /* Data  Load  Address             */
        uint32_t        ih_ep;                  /* Entry Point Address             */
        uint32_t        ih_dcrc;                /* Image Data CRC Checksum         */
        uint8_t         ih_os;                  /* Operating System                */
        uint8_t         ih_arch;                /* CPU architecture                */
        uint8_t         ih_type;                /* Image Type                      */
        uint8_t         ih_comp;                /* Compression Type                */
        uint8_t         ih_name[IH_NMLEN];      /* Image Name                      */
} image_header_t;
```

If the u-Boot header exists, it will load compressed image to temporary and decompress the image to target address. Otherwise, it will load image to target address directly.

```
        /* load execution file */
        pImageList = pImageList+4;
        for (i=0; i<count; i++)
        {
            if (((*(pImageList) >> 16) & 0xffff) == 1)    /* execute */
            {
                UINT32 u32Result;
                startBlock = *(pImageList + 1) & 0xffff;
                executeAddr = *(pImageList + 2);
                fileLen = *(pImageList + 3);

#ifndef __OTP_4BIT__
                SPIReadFast(0, startBlock * 0x10000, 64, (UINT32*)IMAGE_BUFFER);
#else
                spiFlashFastReadQuads(startBlock * 0x10000, 64, (UINT32*)IMAGE_BUFFER);
#endif
```

```
                u32Result = do_bootm(IMAGE_BUFFER, 0, CHECK_HEADER_ONLY);

                if(u32Result)        /* Not compressed */
                {
#ifndef __OTP_4BIT__
                    SPIReadFast(0, startBlock * 0x10000, fileLen, (UINT32*)executeAddr);
#else
                    spiFlashFastReadQuads(startBlock * 0x10000, fileLen,
                                          (UINT32*)executeAddr);
#endif
                }
                else                /* compressed */
                {
#ifndef __OTP_4BIT__
                    SPIReadFast(0, startBlock * 0x10000, fileLen, (UINT32*)IMAGE_BUFFER);
#else
                    spiFlashFastReadQuads(startBlock * 0x10000, fileLen,
                                          (UINT32*)IMAGE_BUFFER);
#endif
                    do_bootm(IMAGE_BUFFER, executeAddr, LOAD_IMAGE);
                }
                 /* Omit some source code in document. */

                /* JUMP to kernel */
                sysprintf("Jump to kernel\n\n");

                 /* Omit some source code in document. */

                fw_func = (void(*)(void))(executeAddr);
                fw_func();
                break;
            }
            /* pointer to next image */
            pImageList = pImageList+12;
        }
```

3.10 Build SPI Loader Project

Normally, the SPI loader doesn't need to modify. If the SPI loader is modified, clicking the **Rebuild** icon as shown below or press **F7** function key to rebuilt it in Keil MDK.

Figure 3-3 Shortcut Icon to Rebuild the SPI Loader on Keil MDK

The binary file of SPI loader will be copied to the *Loaders\Binary* folder with the file name *N9H26_SpiLoader_xxx.bin*. The "*xxx*" is depend on the project target. For the **240MHz_FW050TFT_800x480_24B** project target, the binay file name is *N9H26_SpiLoader_240MHz_FW050TFT_800x480_24B.bin*.

Please note that the binary code size of the SPI loader MUST less than 63 KB.

3.11 Download SPI Loader Binary to SPI Flash

The SPI loader binary on SPI Flash can be programmed by the tool *TurboWriter* and here are the steps. Further information about *TurboWriter* can be found at BSP *Tools/PC_Tools/TurboWriter Tool User Guide.pdf*.

1. Power off device.
2. Plug in USB cable to PC/NB.
3. Power on device under Recovery mode.
4. Run TurboWriter for N9H26 version on PC/NB.
5. Wait for the TurboWriter message to change to "**Mass Storage Connected !**". If not, press the "**Re-Connect**" button to reconnect the device.
6. Select "**SPI**" on the option "**Please choose type**".
7. Select SPI loader binary file on the option "**Image Name**".
8. Select "**System Image**" on the option "**Image Type**".
9. Press "**Burn**" button to burn the SPI loader binary into SPI Flash.
10. After burning completed, check the SPI loader information in the left table.

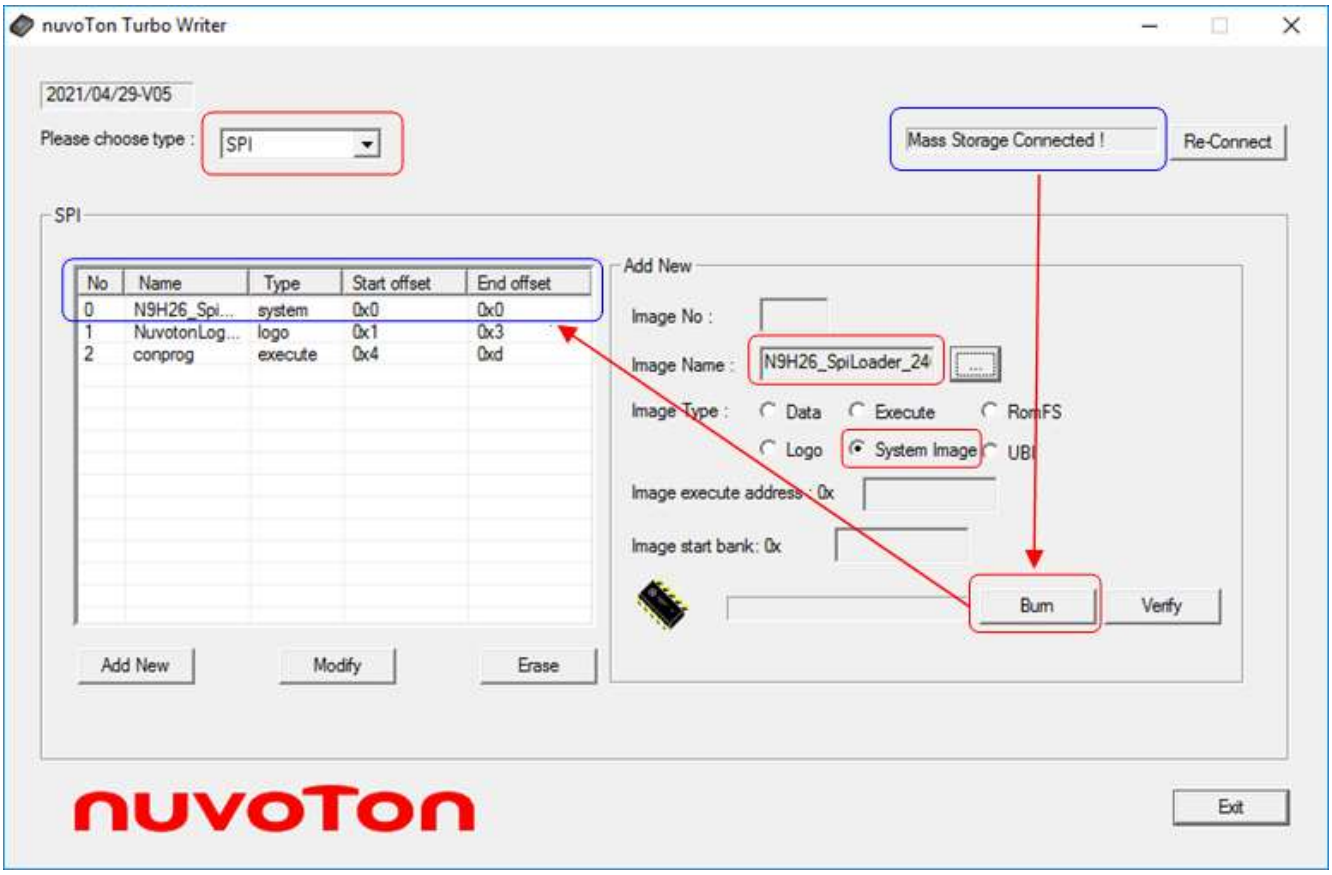

Figure 3-4 Programmed SPI Loader by TurboWriter

11. Remove USB device safely.
12. Plug out USB cable from PC/NB.
13. Reset the device under Normal mode.

3.12 Run SPI Loader

N9H26 has built-in 16K bytes IBR (Internal Booting ROM) where is the boot loader to initialize chip basically when power on, and then try to find out the next stage loader from different type of storage. It could be SD card, NAND Flash, SPI Flash, or USB storage. The booting sequence by the IBR as Figure 3-5.

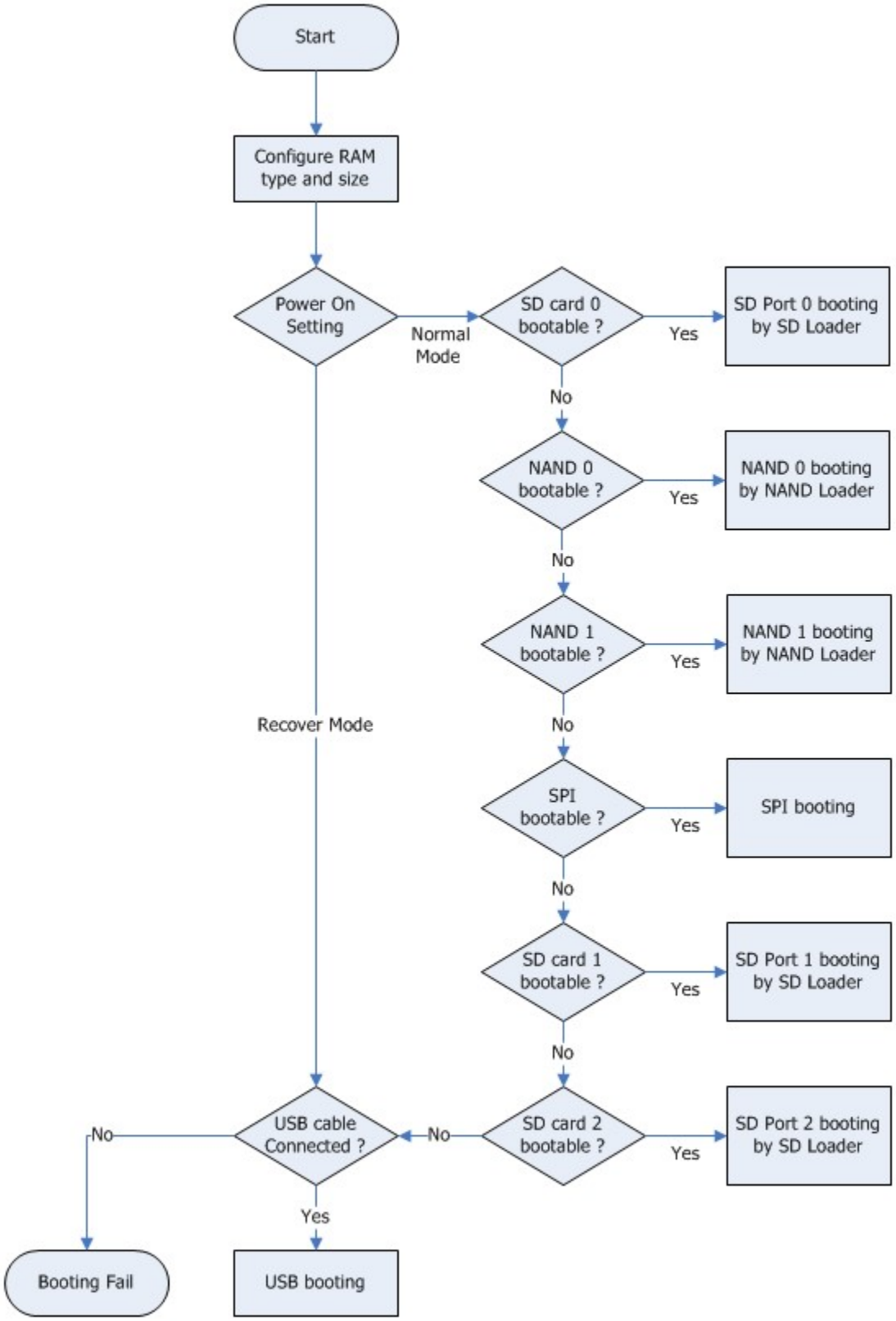

Figure 3-5 IBR Booting Sequence after Power On

The IBR will execute the SPI loader if SD loader on SD card 0 is invalid and NAND

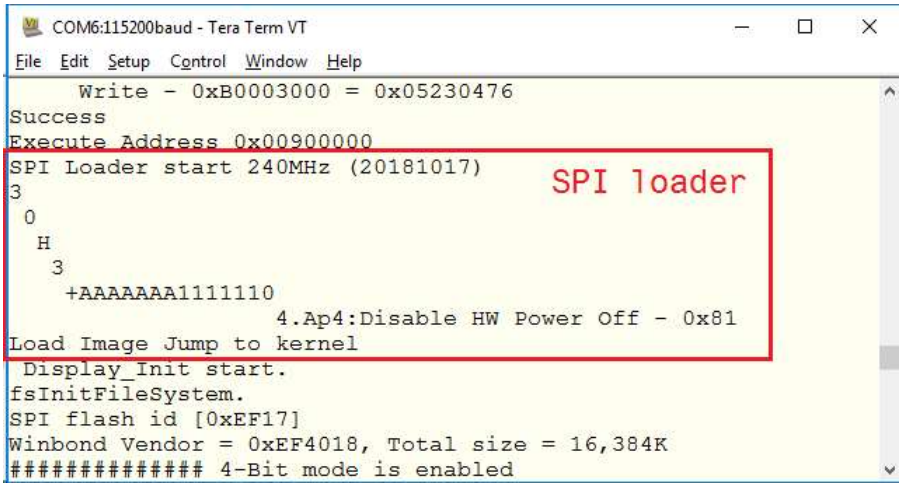

Figure 3-6 The SPI loader Runs on N9H26

3.13 gzip Performance Test Result

Here is an example for SPI loader w/o gzip (240 MHz). The execute image is 2.45 MB and becomes 2.46 MB after compression. SPI loader with gzip can be used when the SPI Flash size is critical.

Loader	Execute Image Size	Load image Time	Unzip time	Total Time
N9H26_SpiLoader_240MHz_NoLCM (23.7KB)	4.52 MB	4.243 s	N/A	5.023 s
N9H26_SpiLoader_gzip_240MHz_NoLCM (40.3KB)	2.46 MB	2.309 s	1.17 s	4.306 s

4 Supporting Resources

The N9H26 system related issues can be posted in Nuvoton's forum:

- ARM7/9 forum at: http://forum.nuvoton.com/viewforum.php?f=12.

Revision History

Date	Revision	Description
2021.6.4	1.01	1. Modify document structure.
2018.4.5	1.00	1. Initially issued.

Important Notice

Nuvoton 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, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.