HID Transfer Firmware Update Solution

Information

Application	The sample code provides a solution for HID Vendor Transfer for Firmware Update
BSP Version	N9H2x Series BSP
Hardware	N9H2x Development Board

1 Function Description

1.1 Introduction

N9H2x works as HID device and uses Vendor commands to achieve data transfer and provides a solution to avoid the product destroyed by failed update operation (Take SPI flash for example code).

Function	USB HID Device
Control Transfer	Control Pipe endpoint 0: Control IN/OUT
HID device	Interface 0 endpoint 1: Interrupt IN endpoint 2: Interrupt OUT

Table 1 HID Device Configuration

1.2 Firmware Update

1.2.1 Memory map

Turbowriter maintains image list (Last page of Block 0 for SPI flash) when user adds or removes image from storage and loader will use the information to load the images from flash to target memory space. Therefore, N9H20 series can't boot up successfully when not only loader and execute image but also image list is destroyed. Figure 1 shows the flash map for SPI flash. The sample code provides a solution to avoid the product destroyed by failed update operation.

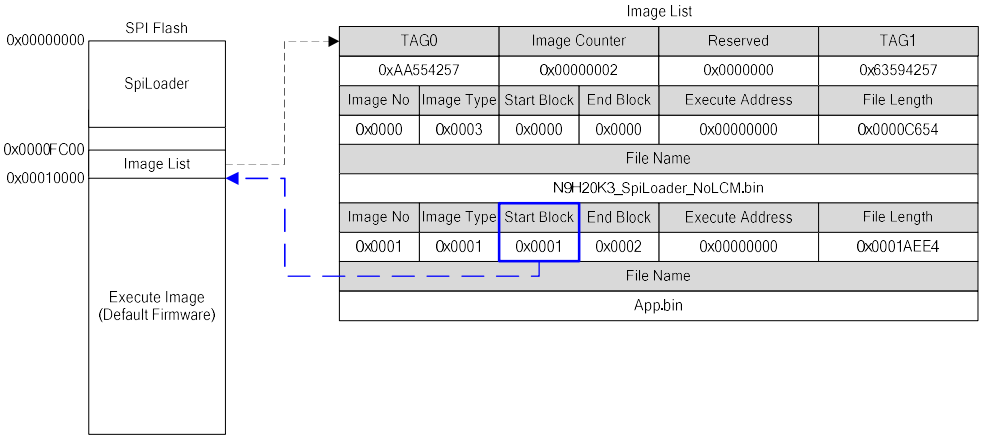


Figure 1 SPI flash map

1.2.2 Principle

The solution doesn't modify the memory space of default firmware (Block 0 & Blocks for execute image) to avoid that the product is destroyed after update failed. In order not to modify Image List, it needs to keep the information about the Updated Firmware in other block. We get the end of block of Execute Image from Image List for the start block of the Updated Firmware (end of block of Execute Image + 1) and use the firmware image format to indicate the Updated Firmware status, version, and size (See Figure 2).

START TAG0	Version	Firmware Size	START TAG1
0x4E565420	20191223	0x0000EDB0	0x2054564E
Updated Firmware			
			End TAG
			0xA55AA55A

Figure 2 Firmware image format

END TAG offset is Updated Firmware start address + PAGE_SIZE (256 Bytes) x page number of (Updated Firmware Header Size + Firmware Size), i.e., the next page of last page of firmware. When the update flow starts, updater will erase the END TAG first and write the END TAG after firmware update operation is finished correctly. Therefore, loader and HID device can check if firmware version and size are available by START and END TAGs (Updated Firmware uses the same execute address as Execute Image). The Updated Firmware is available when START TAGs and END TAG are correct. Figure 3 shows the flash map for the solution.

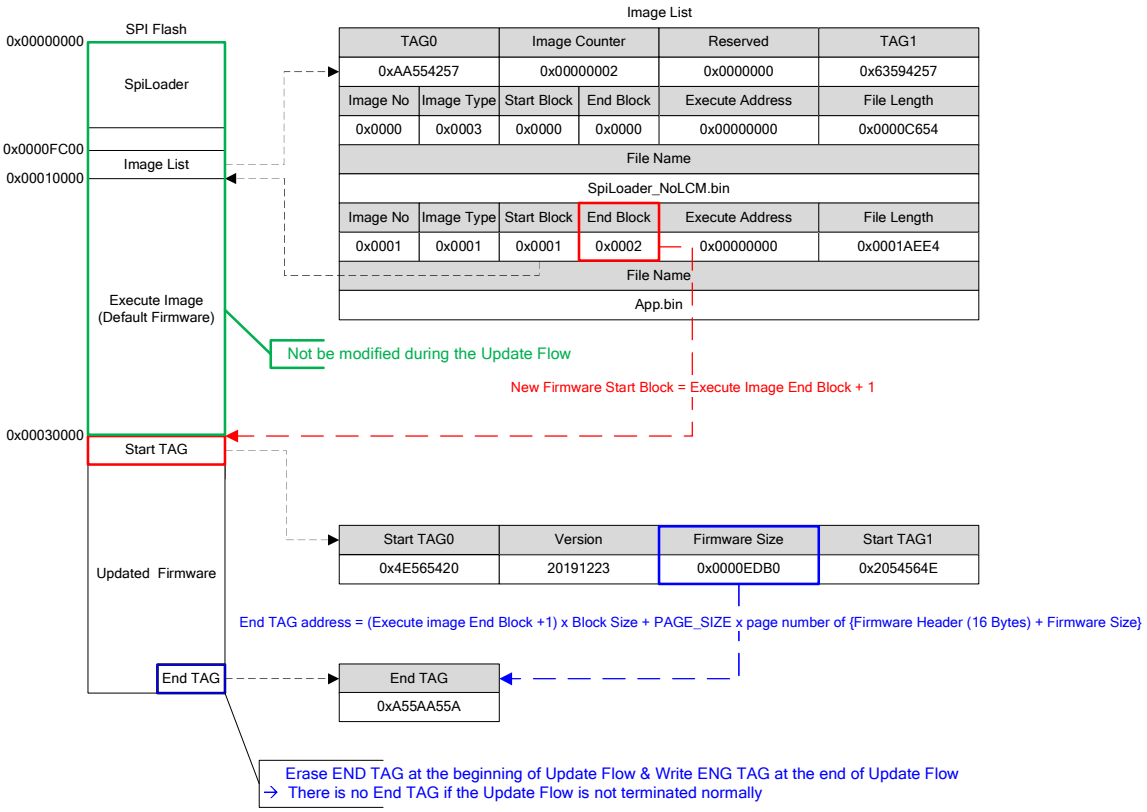


Figure 3 SPI flash map for the solution

1.3 HID Vendor Command

The solution defines a set of HID vendor command for the firmware update flow. Take packets in full speed mode for example. Table 2 shows the command structure and Table 3 shows the command set for the firmware update flow. Here takes full speed for example.

Byte	Description	Comment
0	Command	
1	Length	0xE (Not include Checksum& Reserved)
2~5	Arg 1	
6~9	Arg 2	
10~13	Signature	0x43444948
14~17	Checksum	
18~63	Reserved	18~511 for High Speed

Table 2 Command structure

Description	CMD	Arg1	Arg2	Comment
GET_VERSION	0xD3	N/A	N/A	Get Current Version
GET_START_BLOCK	0xD5	N/A	N/A	Get Firmware Start Block No.
GET_STATUS	0xD4	N/A	N/A	Get SPI Flash status
ERASE	0x71	Block No.	N/A	Erase Block
UPDATE	0xB0	N/A	N/A	Start to Update
WRITE	0xC3	Page No.	Page Count	Write page
READ	0xD2	Page No.	Page Count	Read page
EXIT	0xB1	N/A	N/A	End of Update
IMAGE_WRITE	0xC4	N/A	Data Count	Write Image to display buffer
SET_PARAM	0xC5	Arg1	Arg2	Set parameter
GET_PARAM	0xD6	Arg1	Arg2	Get parameter

Table 3 HID Vendor Command set

1.3.1 GET_VERSION

This command can get the firmware version. Device will return current version number. Table 4 and Figure 4 show the GET_VERSION command example (If there is only default firmware, it will return 0x00000000). It gets 0x013417F7 (20191223).

CMD	Length	Arg1	Arg2	Signature	Checksum
0xD3	0x0E	0x00000000	0x00000000	0x43444948	0x000001F9

Table 4 GET_VERSION command example

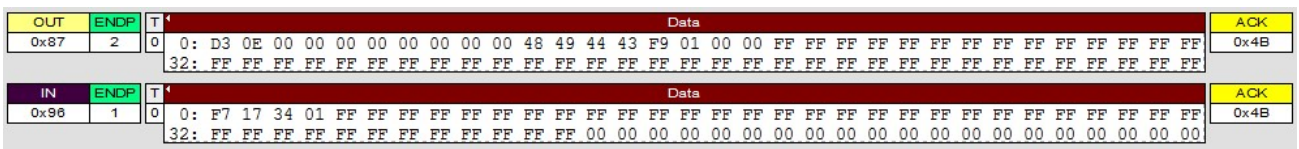


Figure 4 GET_VERSION command flow example

1.3.2 GET_START_BLOCK

This command can get the start block of Updated Firmware. Device will return the block number of Updated Firmware. Table 5 and Figure 5 show the GET_START_BLOCK command example. It gets 0x0000003 (Updated Firmware starts from Block 3).

CMD	Length	Arg1	Arg2	Signature	Checksum
0xD5	0x0E	0x00000000	0x00000000	0x43444948	0x000001FB

Table 5 GET_START_BLOCK command example

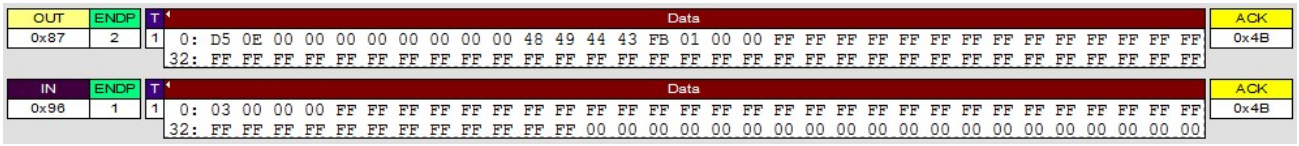


Figure 5 GET_START_BLOCK command flow example

1.3.3 GET_STATUS

This command can get SPI flash status. Device will return SPI flash status (0 means “Ready” and 1 means “Busy”. Table 6 and Figure 6 show the GET_STATUS command example.

CMD	Length	Arg1	Arg2	Signature	Checksum
0xD4	0x0E	0x00000000	0x00000000	0x43444948	0x000001FA

Table 6 GET_STATUS command example

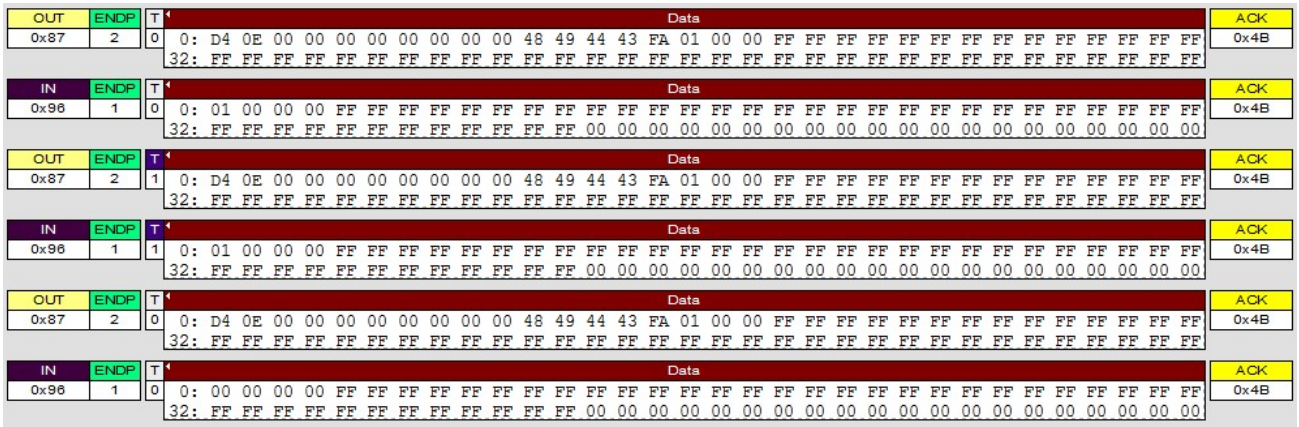


Figure 6 GET_STATUS command flow example

1.3.4 ERASE

This command can erase SPI flash block. After this command, master must call ERASE command until it gets Ready status. Table 7 and Figure 7 show the ERASE command example.

CMD	Length	Arg1	Arg2	Signature	Checksum
0x71	0x0E	0x00000003	0x00000001	0x43444948	0x0000019B

Table 7 ERASE command example

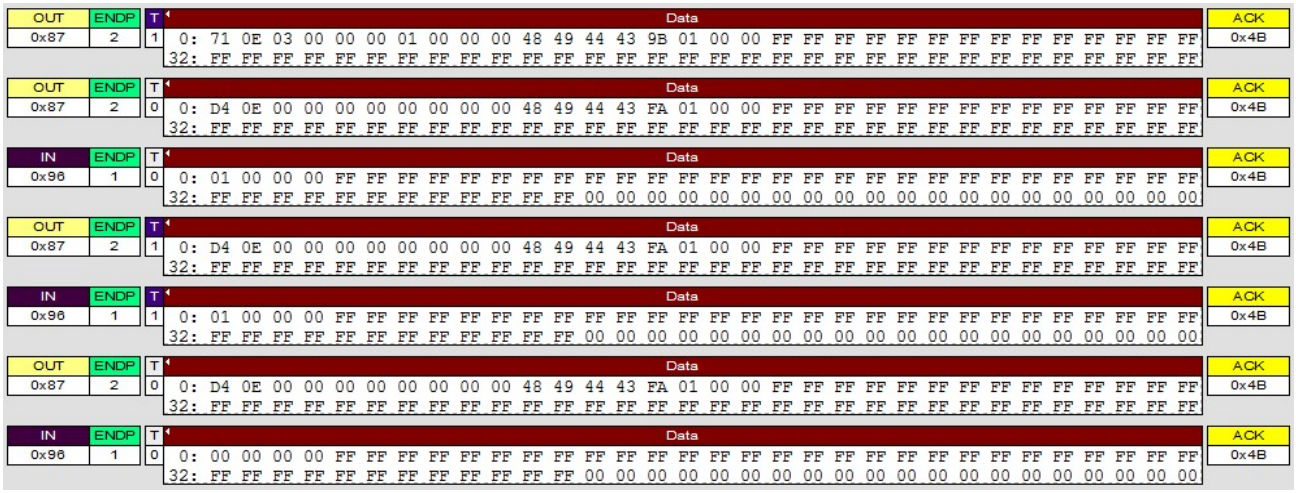


Figure 7 ERASE command flow example

1.3.5 UPDATE

This command means the update firmware flow starts. Device will erase the block that END TAG located if it exists. Table 8 and Figure 8 show the UPDATE command example.

CMD	Length	Arg1	Arg2	Signature	Checksum
0xB0	0x0E	0x00000000	0x00000000	0x43444948	0x000001D6

Table 8 UPDATE command example

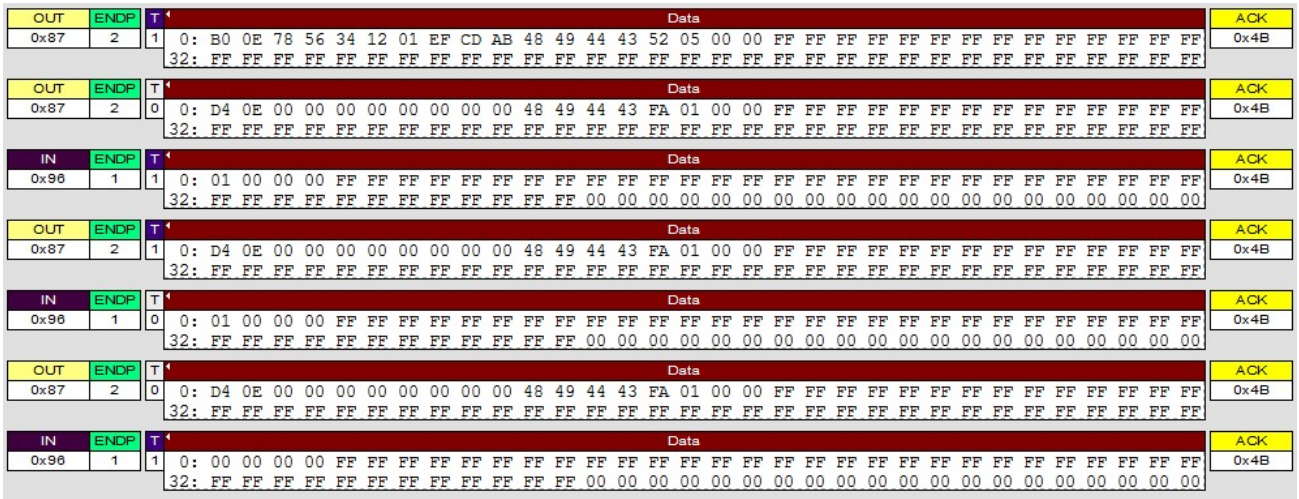


Figure 8 UPDATE command flow example

1.3.6 WRITE

This command writes data to SPI flash. After this command, master needs to send data to device according to the arguments (Arg1 is the start page number and Arg2 is the page count). Table 9 and Figure 9 show the WRITE command example. It writes 238 pages to page number 768 (block 3).

CMD	Length	Arg1	Arg2	Signature	Checksum
0xC3	0x0E	0x00000300	0x000000EE	0x43444948	0x000002DA

Table 9 WRITE command example

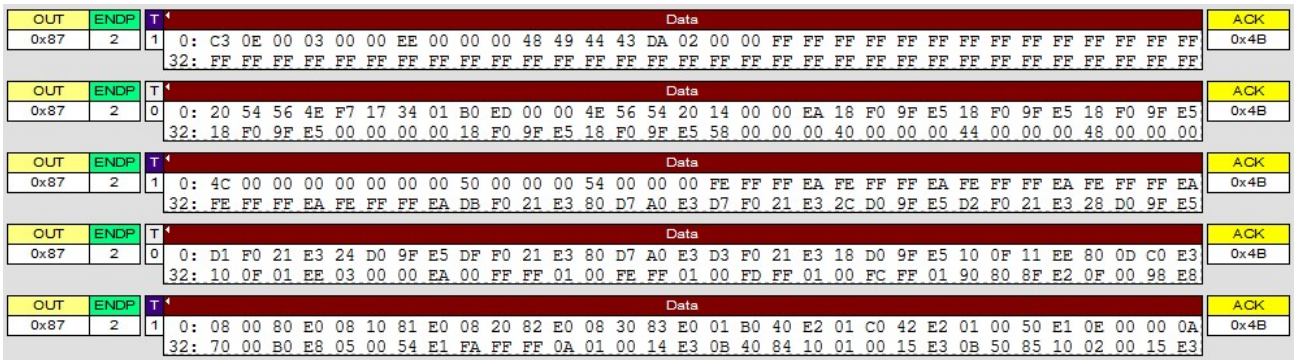


Figure 9 WRITE command flow example

1.3.7 READ

This command read data from SPI flash. After this command, device needs to send data to host according to the arguments (Arg1 is the start page number and Arg2 is the page count). Table 10 and Figure 10 show the READ command example. It reads 238 pages from page number 768 (block 3).

CMD	Length	Arg1	Arg2	Signature	Checksum
0xD2	0x0E	0x00000300	0x000000EE	0x43444948	0x000002E9

Table 10 READ command example

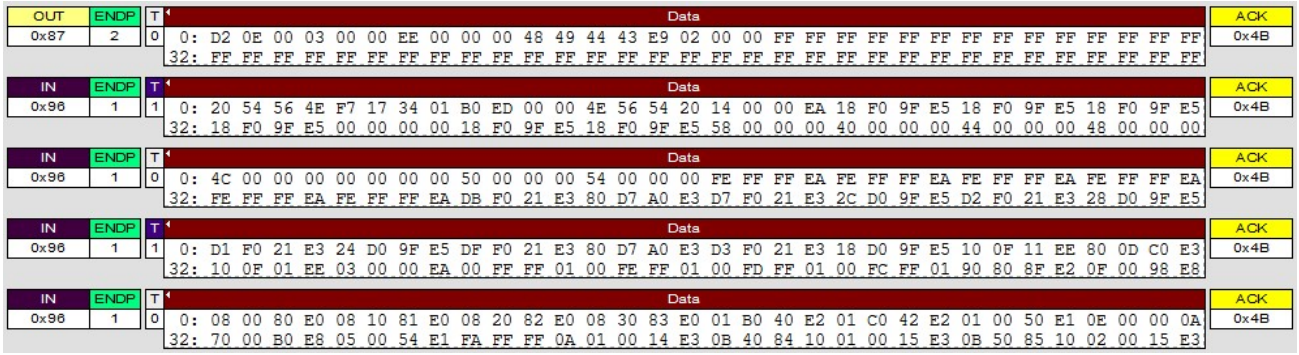


Figure 10 READ command flow example

1.3.8 EXIT

This command means the update flow is done. In this example code, N9H2x will disable the USB function and load firmware.

CMD	Length	Arg1	Arg2	Signature	Checksum
0xB1	0x0E	0x00000000	0x00000000	0x43444948	0x000001D7

Table 11 EXIT command example

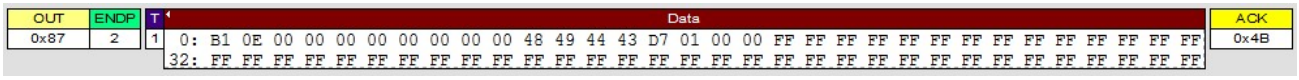


Figure 11 EXIT command flow example

1.3.9 IMAGE_WRITE

This command writes data to Image Buffer. After this command, master needs to send data to device according to the arguments (Arg2 is the data count). Table 12 and Figure 12 show the IMAGE_WRITE command example. It decodes Image file (JPEG) to image buffer (47616 Bytes).

CMD	Length	Arg1	Arg2	Signature	Checksum
0xC4	0x0E	0x0000BA00	0x00000000	0x43444948	0x000002A4

Table 12 IMAGE_WRITE command example

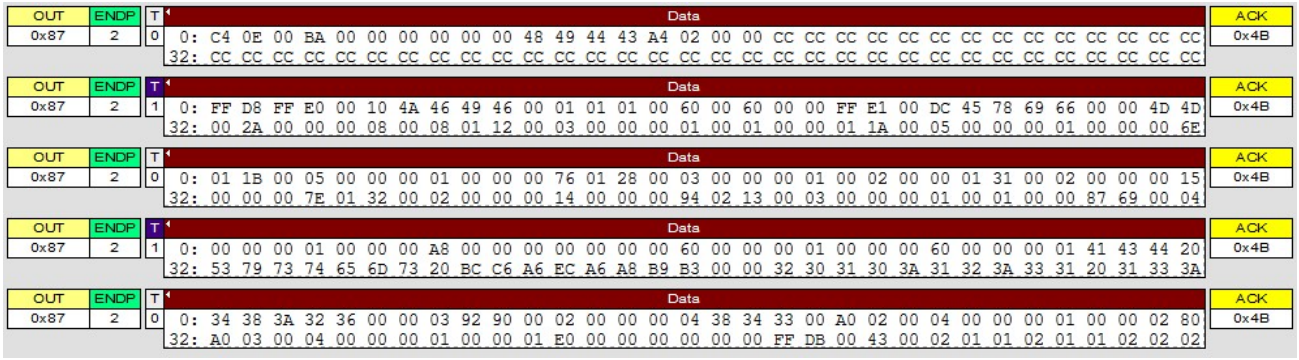


Figure 12 IMAGE_WRITE command flow example

1.3.10 SET_PARAM

This command can get the firmware version. Device will return current version number. Table 13 and Figure 13 show the SET_PARAM command example (Set parameter – Arg1 0x04 & Arg2 0x03 with data 0xBC614E).

CMD	Length	Arg1	Arg2	Signature	Checksum
0xC5	0x0E	0x00000004	0x00000003	0x43444948	0x000001F2

Table 13 SET_PARAM command example

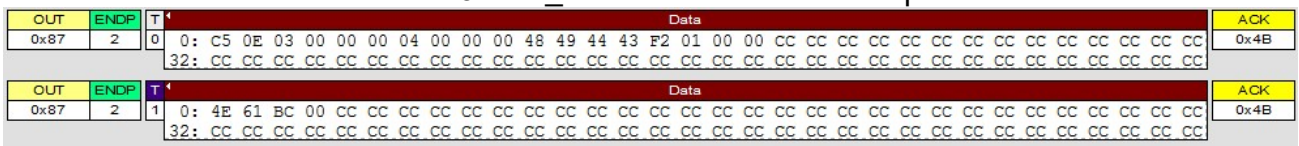


Figure 13 SET_PARAM command flow example

1.3.11 GET_PARAM

This command can get the firmware version. Device will return current version number. Table 14 and Figure 14 show the GET_PARAM command example (Get parameter – Arg1 0x04 & Arg2 0x03 with returned data 0xBC614E)..

CMD	Length	Arg1	Arg2	Signature	Checksum
0xD6	0x0E	0x00000003	0x00000004	0x43444948	0x00000203

Table 14 GET_PARAM command example

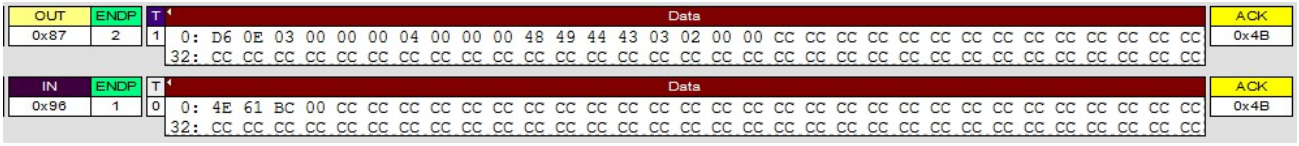


Figure 14 GET_PARAM command flow example

1.4 Firmware Update

1.4.1 Update Flow Implementation

Figure 15 shows that the flow for Firmware Update. SpiLoader needs to be modified for the Updated Firmware loading flow. The update operation will execute in USB interrupt service routine. Therefore, process (Execute Image) may not get response immediately (maximum is about 4 ms for USB full speed mode) when the update operation is in progress.

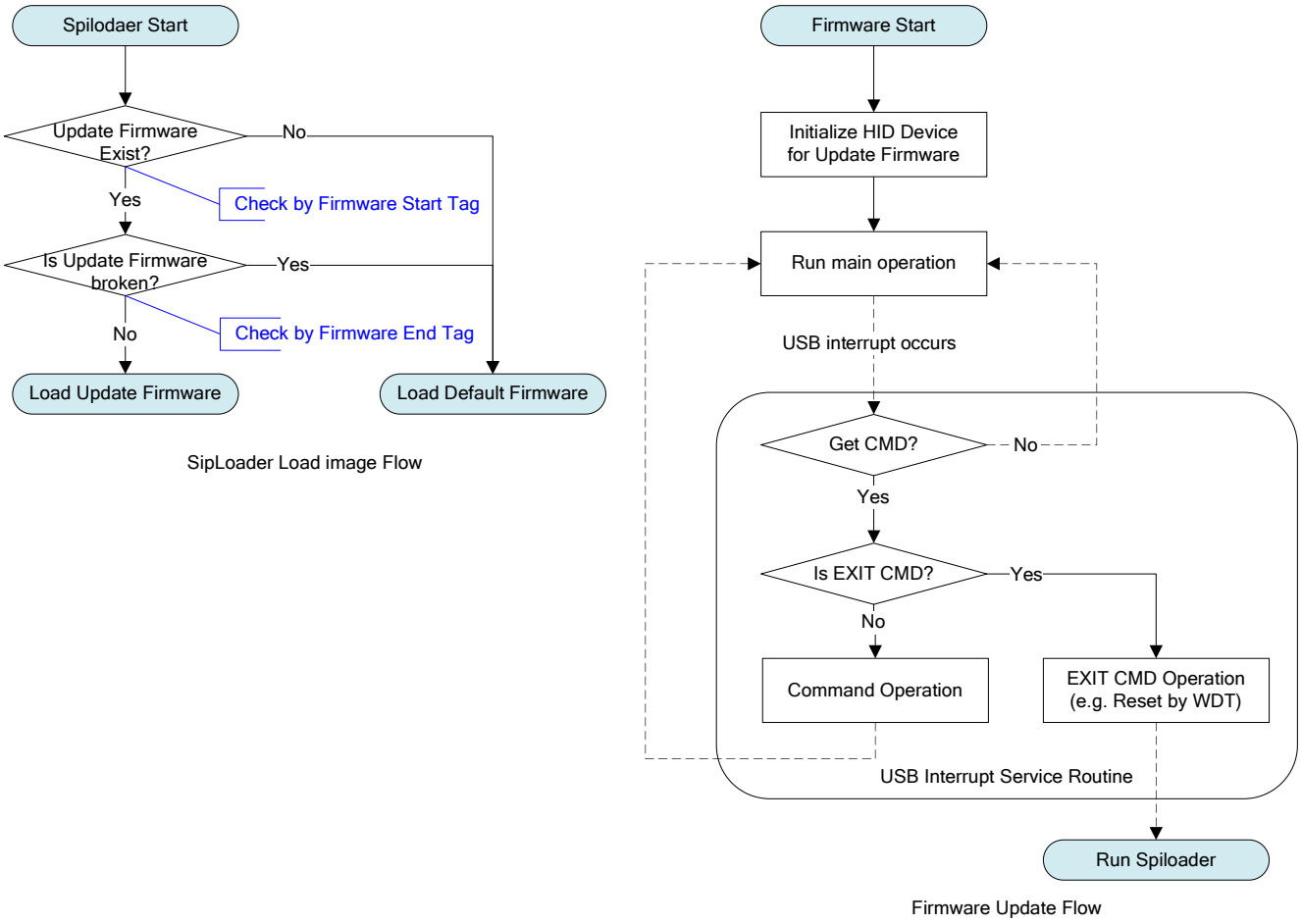


Figure 15 SpiLoader & HID Device Control Flow for Firmware Update

1.4.2 PC Tool Update CMD Flow

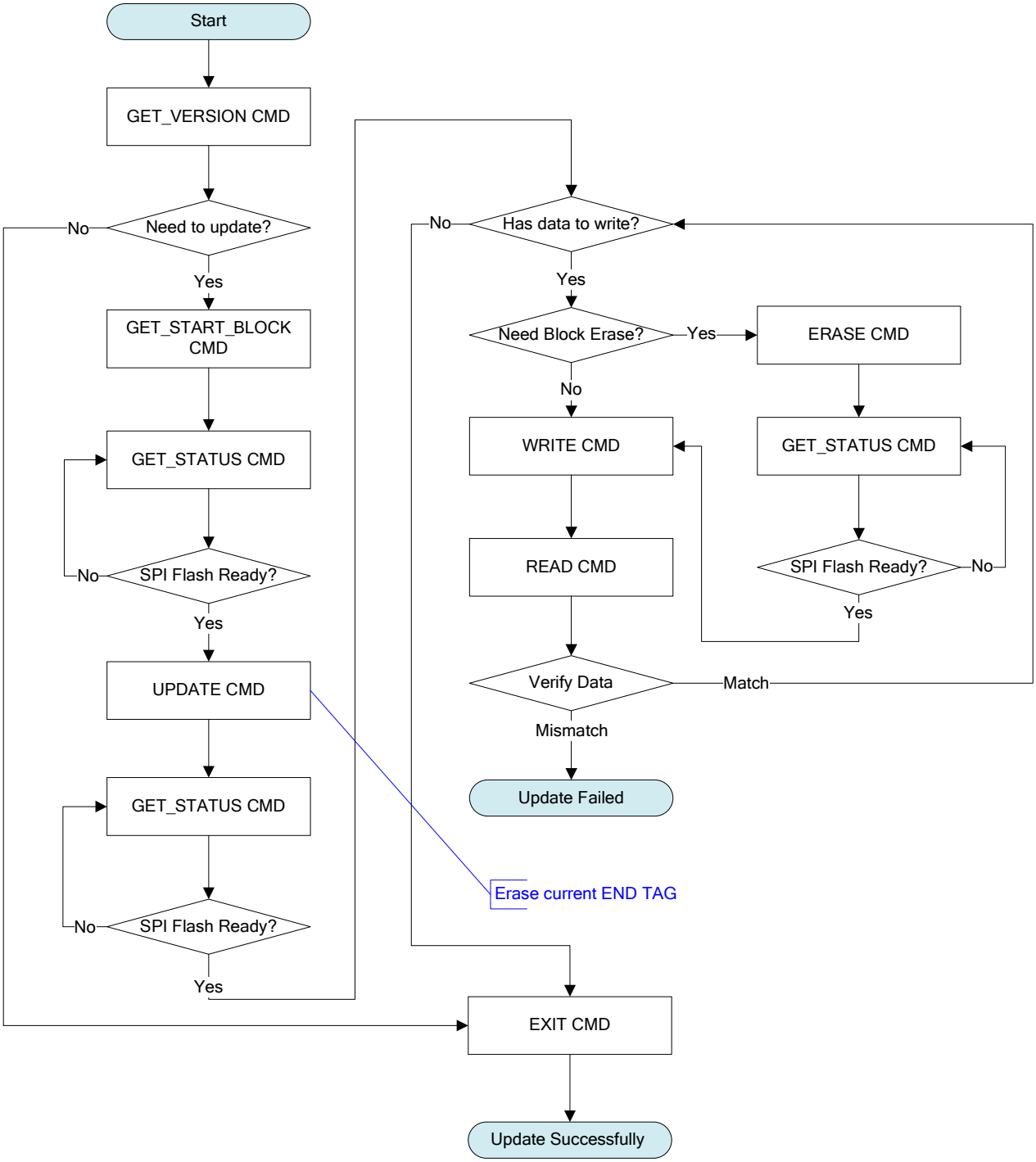


Figure 16 PC Tool CMD Flow

1.5 Demo Result

The solution provides a Command Line Tool: HID_TransferTool_HS.exe for reference. Figure 17 shows the Command format.

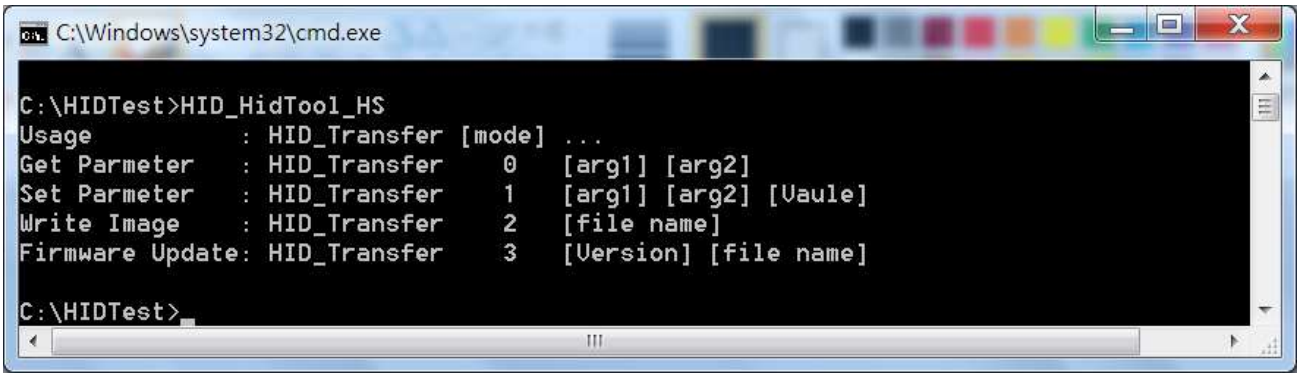


Figure 17 Command format

After the tool executes, it will wait and detect the HID Device with VID – 0x0416 and PID – 0x5020 for the update flow.

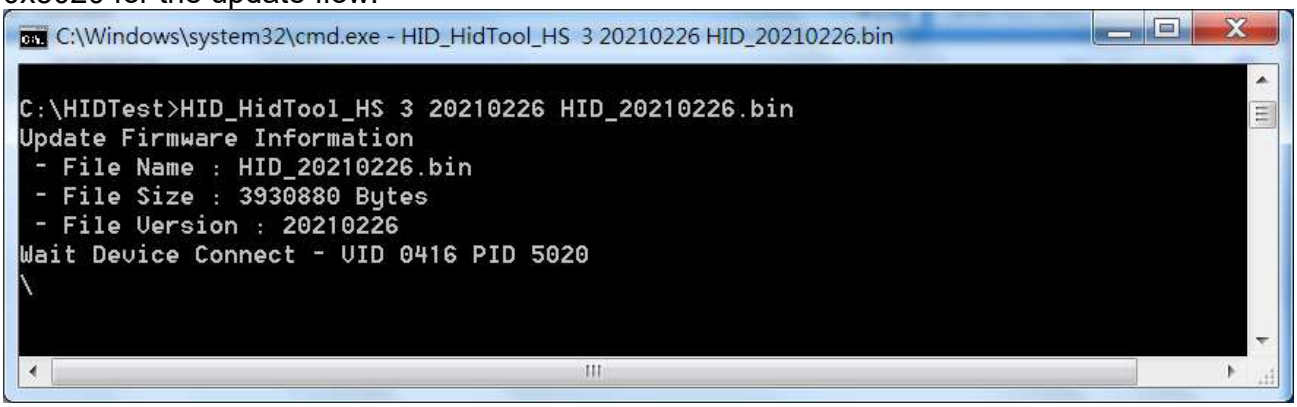


Figure 18 Detect HID Device

1.5.1 Parameter Control Demo Result

1.5.1.1 Set Parameter Control Demo Result

The Command Format is "HID_HidTool_HS 1 [Arg1] [Arg2] [Value]" as Figure 19 (Set parameter – Arg1 0x04 & Arg2 0x03 with data 12345678).

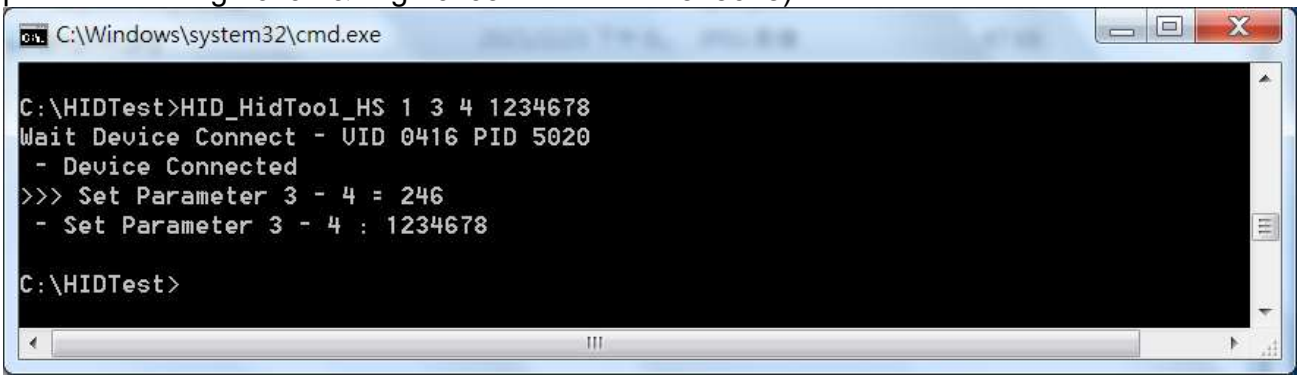


Figure 19 Set Parameter Example

1.5.1.2 Get Parameter Control Demo Result

The Command Format is "HID_HidTool_HS 0 [Arg1] [Arg2] [Value]" as Figure 20 (Get parameter – Arg1 0x04 & Arg2 0x03 with returned data 12345678).

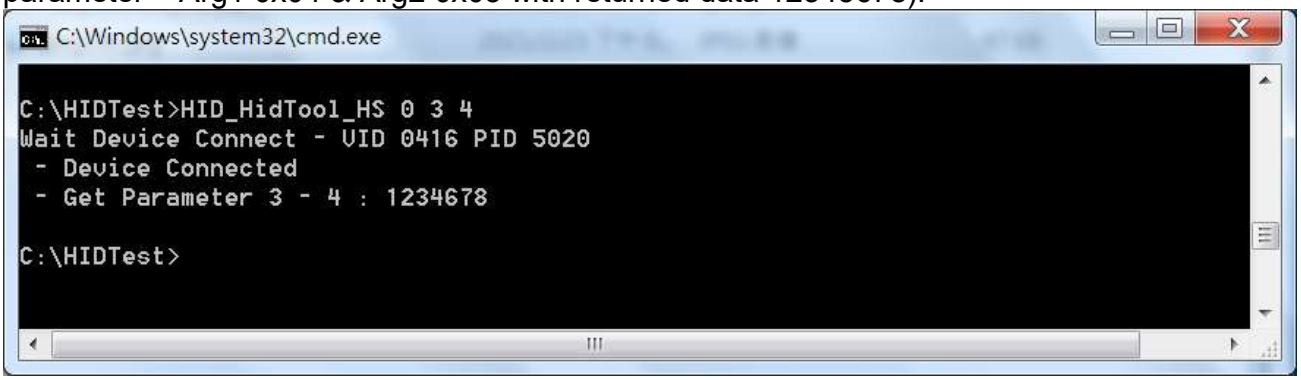


Figure 20 Get Parameter Example

1.5.2 Image Display Demo Result

The Command Format is "HID_HidTool_HS 2 [File Name]" as Figure 21 (Image Write with JPEG file – jpegDec.jpg).

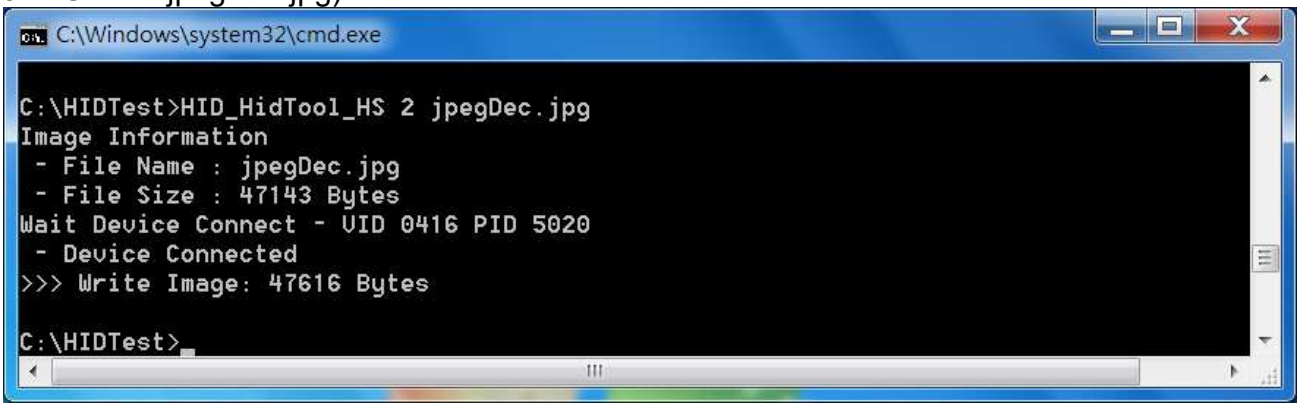


Figure 21 Get Parameter Example

1.5.3 Firmware Update Demo Result

The Command Format is "HID_HidTool_HS 3 [Version Number] [File Name]" as Figure 22. After HID device connects to PC and PC Tool will connect to the HID device. PC Tool will start to write the update firmware, write END TAG after update flow is complete correctly and send Exit command. N9H2x will reset after getting EXIT command.

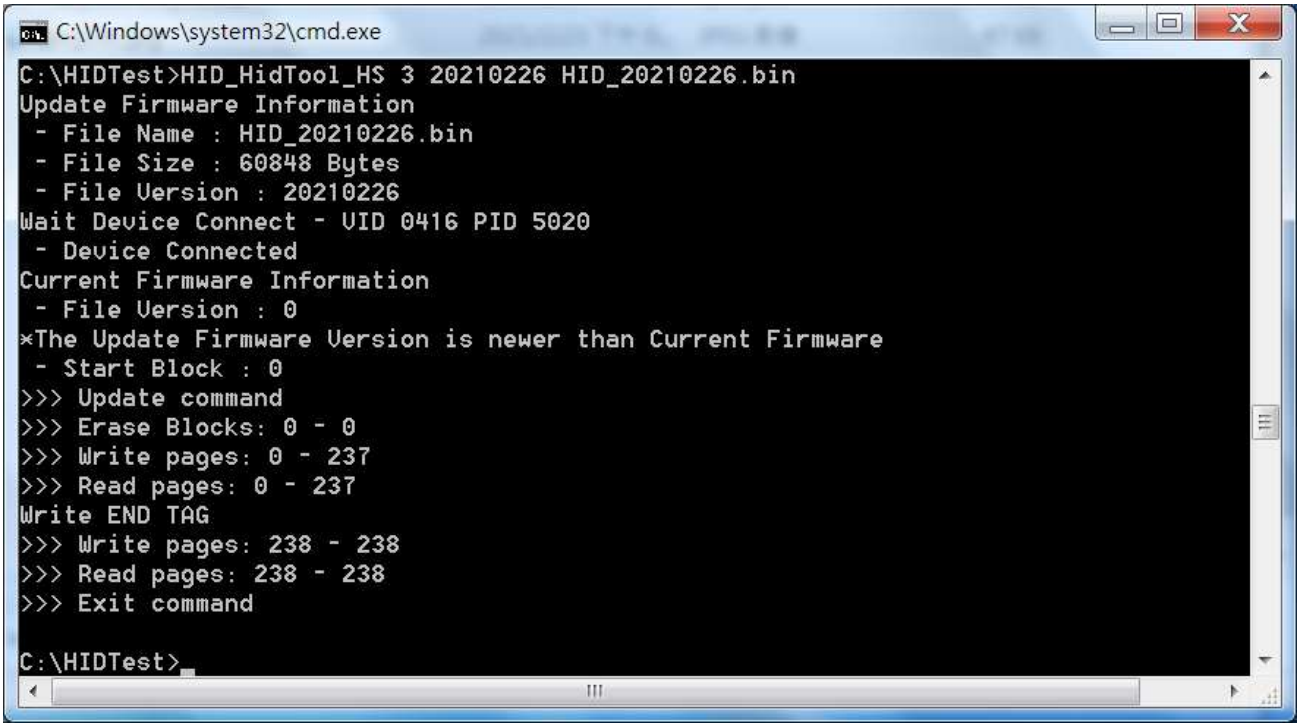


Figure 22 Update Firmware Flow Example

2 Code Description

2.1 HID Device Initialization

he following code include the USB Engine initialization.

```
/* HID High Speed Init */
void hidHighSpeedInit(void)
{
    usbdInfo.usbdMaxPacketSize = 0x40;
    outp32(EPA_MPS, 0x40);                    /* mps */
    while(inp32(EPA_MPS) != 0x40);            /* mps */

    /* bulk in */
    outp32(EPA_IRQ_ENB, 0x00000008);          /* tx transmitted */
    outp32(EPA_RSP_SC, 0x000000000);          /* auto validation */
    outp32(EPA_MPS, EPA_MAX_PKT_SIZE);        /* mps 512 */
    outp32(EPA_CFG, 0x0000001b);              /* bulk in ep no 1 */
    outp32(EPA_START_ADDR, EPA_BASE);
    outp32(EPA_END_ADDR, EPA_BASE + EPA_MAX_PKT_SIZE - 1);

    /* bulk out */
    outp32(EPB_IRQ_ENB, 0x00000010);          /* data pkt received */
    outp32(EPB_RSP_SC, 0x00000000);           /* auto validation */
    outp32(EPB_MPS, EPB_MAX_PKT_SIZE);        /* mps 512 */
    outp32(EPB_CFG, 0x00000023);              /* bulk out ep no 2 */
    outp32(EPB_START_ADDR, EPB_BASE);
    outp32(EPB_END_ADDR, EPB_BASE + EPB_MAX_PKT_SIZE - 1);
    g_u32EPA_MXP = EPA_MAX_PKT_SIZE;
    g_u32EPB_MXP = EPB_MAX_PKT_SIZE;
    g_u32ReadWriteSize = EPA_MAX_PKT_SIZE;
    usbdInfo.pu32HIDRPTDescriptor[0] =  (PUINT32) &HID_DeviceReportDescriptor;
    usbdInfo.u32HIDRPTDescriptorLen[0] = sizeof(HID_DeviceReportDescriptor);
}

/* HID Full Speed Init */
void hidFullSpeedInit(void)
{
    usbdInfo.usbdMaxPacketSize = 0x40;
    outp32(EPA_MPS, 0x40);                    /* mps */
    while(inp32(EPA_MPS) != 0x40);            /* mps */
```

```
    /* bulk in */
    outp32(EPA_IRQ_ENB, DATA_TxED_IE);        /* tx transmitted */
    outp32(EPA_RSP_SC, 0x000000000);          /* auto validation */
    outp32(EPA_MPS, EPA_OTHER_MAX_PKT_SIZE);  /* mps 64 */
    outp32(EPA_CFG, 0x0000001b);              /* bulk in ep no 1 */
    outp32(EPA_START_ADDR, EPA_OTHER_BASE);
    outp32(EPA_END_ADDR, EPA_OTHER_BASE + EPA_OTHER_MAX_PKT_SIZE - 1);

    /* bulk out */
    outp32(EPB_IRQ_ENB, 0x00000010);          /* data pkt received */
    outp32(EPB_RSP_SC, 0x00000000);           /* auto validation */
    outp32(EPB_MPS, EPB_OTHER_MAX_PKT_SIZE);  /* mps 64 */
    outp32(EPB_CFG, 0x00000023);              /* bulk out ep no 2 */
    outp32(EPB_START_ADDR, EPB_OTHER_BASE);
    outp32(EPB_END_ADDR, EPB_OTHER_BASE + EPB_OTHER_MAX_PKT_SIZE - 1);

    g_u32EPA_MXP = EPA_OTHER_MAX_PKT_SIZE;
    g_u32EPB_MXP = EPB_OTHER_MAX_PKT_SIZE;
    g_u32ReadWriteSize = SPI_PAGE_SIZE;
#ifdef __FORCE_FULLSPEED__
    outp32(OPER, 0);
#endif
    usbdInfo.pu32HIDRPTDescriptor[0] =  (PUINT32) &HID_DeviceReportDescriptor_FS;
    usbdInfo.u32HIDRPTDescriptorLen[0] = sizeof(HID_DeviceReportDescriptor_FS);
}
```

The following code include the HID Device initialization and the update operation will be executed when HID Device gets command.

```
void HIDStart(void)
{
    /* Enable USB */
    udcOpen();
    spiFlashInit();
    usiCheckBusy();
    GetInfo();
    hidInit();
    udcInit();
}
```

In the function of hidClassOUT(), PC will issue a SET_IDLE request to device and N9H2x sets timeout setting to stop the HID device.

```
void hidClassOUT(void)
{
    if(_usb_cmd_pkt.bRequest == HID_SET_IDLE)
    {
//        sysprintf("\rSet IDLE\n");
    }
    else if(_usb_cmd_pkt.bRequest == HID_SET_REPORT)
    {
        u32Ready = 1;
        sysprintf("\rSET_REPORT 0x%X\n",inp8(CEP_DATA_BUF));
    }
}
```

2.2 HID Transfer

The HID transfer operation executes in USB interrupt service routine.

```
void EPA_Handler(UINT32 u32IntEn,UINT32 u32IntStatus)  /* Interrupt IN handler */
{
    HID_SetInReport();
}
void EPB_Handler(UINT32 u32IntEn,UINT32 u32IntStatus)  /* Interrupt OUT handler */
{
    HID_GetOutReport();
}
```

In HID_GetOutReport function, it parses Command and handles the Write & Image Write command flow (It uses the member "u8Cmd" of pCmd to command flow status). When Write Command, it writes data to SPI flash if data is enough. When Image Write Command, it decode data (JPEG bitstream) to display buffer.

```
void HID_GetOutReport(void)
{
    UINT8  u8Cmd;
    UINT32 u32StartPage;
    UINT32 u32Pages;
    UINT32 u32PageCnt;
    UINT32 u32TotalCnt;
    UINT32 u32DataCnt;
    UINT32 u32Size = inp32(EPB_DATA_CNT) & 0xFFFF;
    /* Get command information */
    u8Cmd        = pCmd->u8Cmd;
    u32StartPage = pCmd->u32Arg1;
```

```
    u32TotalCnt  = pCmd->u32Arg1;     /* The word is used to target data count for
IMAGE_WRITE CMD */
    u32Pages     = pCmd->u32Arg2;
    u32PageCnt   = pCmd->u32Signature; /* The signature word is used to count pages for
WRITE CMD */
    u32DataCnt   = pCmd->u32Signature; /* The signature word is used to count data for
IMAGE_WRITE CMD */

    /* Check if it is in the data phase of write command */
    if((u8Cmd == HID_CMD_WRITE) && (u32PageCnt < u32Pages))
    {
        while(usbdInfo.USBModeFlag)
        {
            if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
                break;
        }
        /* Process the data phase of write command */
        outp32(DMA_CTRL_STS, 0x02);    /* bulk out, dma write, ep2 */
        if(g_u32EPA_MXP == EPA_MAX_PKT_SIZE)
            outp32(AHB_DMA_ADDR, (UINT32)&g_u8PageBuff[0]);
        else
            outp32(AHB_DMA_ADDR, (UINT32)&g_u8PageBuff[g_u32BytesInPageBuf]);
        outp32(DMA_CNT, u32Size);
        outp32(DMA_CTRL_STS, inp32(DMA_CTRL_STS)|0x00000020);
        while(usbdInfo.USBModeFlag)
        {
            if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
                break;
        }
        g_u32BytesInPageBuf += u32Size;

        if(g_u32BytesInPageBuf >= g_u32ReadWriteSize)
        {
            spiFlashWrite((u32StartPage + u32PageCnt) * SPI_PAGE_SIZE, g_u32ReadWriteSize,
                          (UINT32 *)g_u8PageBuff);

            if(g_u32EPA_MXP == EPA_MAX_PKT_SIZE)
                u32PageCnt+=2;
            else
                u32PageCnt++;
            /* Write command complete! */
            if(u32PageCnt >= u32Pages)
```

```
                u8Cmd = HID_CMD_NONE;
            g_u32BytesInPageBuf = 0;
        }
        /* Update command status */
        pCmd->u8Cmd        = u8Cmd;
        pCmd->u32Signature = u32PageCnt;
    }
    /* Check if it is in the data phase of image write command */
    else if((u8Cmd == HID_CMD_IMAGE_WRITE) &&  (u32DataCnt < u32TotalCnt))
    {
        while(usbdInfo.USBModeFlag)
        {
            if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
                break;
        }
        /* Process the data phase of write command */
        outp32(DMA_CTRL_STS, 0x02);    /* bulk out, dma write, ep2 */
        outp32(AHB_DMA_ADDR, (UINT32)&g_u8JPEGBuff[g_u32BytesInJPEGBuf]);
        outp32(DMA_CNT, u32Size);
        outp32(DMA_CTRL_STS, inp32(DMA_CTRL_STS)|0x00000020);
        while(usbdInfo.USBModeFlag)
        {
            if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
                break;
        }
        g_u32BytesInJPEGBuf += u32Size;

        /* Write command complete! */
        if(g_u32BytesInJPEGBuf >= u32TotalCnt)
        {
            pCmd->u8Cmd = HID_CMD_NONE;
            JpegDec((UINT32)g_u8JPEGBuff,(UINT32) g_pu8FrameBuffer);
        }
        else
        {
            /* Update command status */
            pCmd->u8Cmd        = u8Cmd;
            pCmd->u32Signature = u32DataCnt;
        }
    }
    /* Check if it is in the data phase of image set parameter command */
```

```
    else if(u8Cmd == HID_CMD_SET_PARAM)
    {
        while(usbdInfo.USBModeFlag)
        {
            if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
                break;
        }
        /* Trigger HID IN */
        outp32(DMA_CTRL_STS, 0x02);    /* bulk out, dma read, ep2 */
        outp32(AHB_DMA_ADDR, (UINT32)g_Temp);
        outp32(DMA_CNT, g_u32EPA_MXP);
        outp32(DMA_CTRL_STS, inp32(DMA_CTRL_STS)|0x00000020);
        while(usbdInfo.USBModeFlag)
        {
            if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
            break;
        }
        SetInfo(pCmd->u32Arg1, pCmd->u32Arg2, (UINT32)g_Temp);
        /* The data transfer is complete. */
        pCmd->u8Cmd = HID_CMD_NONE;
    }
    else
    {
        /* Check and process the command packet */
        if(ProcessCommand(u32Size))
        {
            sysprintf("\rUnknown HID command!\n");
        }
    }
}
```

In HID_SetInReport function, it handles the READ command flow. If previous page has sent out, it reads new page to page buffer.

```
void HID_SetInReport(void)
{
    UINT32 u32StartPage;
    UINT32 u32TotalPages;
    UINT32 u32PageCnt;
    UINT8  u8Cmd;

    u8Cmd        = pCmd->u8Cmd;
    u32StartPage = pCmd->u32Arg1;
```

```
    u32TotalPages= pCmd->u32Arg2;
    u32PageCnt   = pCmd->u32Signature;

    /* Check if it is in data phase of read command */
    if(u8Cmd == HID_CMD_READ)
    {
        /* Process the data phase of read command */
        if((u32PageCnt >= u32TotalPages) && (g_u32BytesInPageBuf == 0))
        {
            /* The data transfer is complete. */
            u8Cmd = HID_CMD_NONE;
        }
        else
        {
            if(g_u32BytesInPageBuf == 0)
            {
                /* The previous page has sent out. Read new page to page buffer */
                spiFlashRead((u32StartPage + u32PageCnt) * SPI_PAGE_SIZE,
                             g_u32ReadWriteSize , (UINT32 *)g_u8PageBuff);
                if(g_u32EPA_MXP == EPA_MAX_PKT_SIZE)
                    u32PageCnt+=2;
                else
                {
                    g_u32BytesInPageBuf = SPI_PAGE_SIZE;

                    /* Update the page counter */
                    u32PageCnt++;
                }
            }
            while(usbdInfo.USBModeFlag)
            {
                if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
                    break;
            }
            /* Prepare the data for next HID IN transfer */
            outp32(DMA_CTRL_STS, 0x11);    /* bulk in, dma read, ep1 */
            if(g_u32EPA_MXP == EPA_MAX_PKT_SIZE)
                outp32(AHB_DMA_ADDR, (UINT32)&g_u8PageBuff[0]);

            else
                outp32(AHB_DMA_ADDR, (UINT32)&g_u8PageBuff[SPI_PAGE_SIZE -
```

```
                        g_u32BytesInPageBuf]);
            outp32(DMA_CNT, g_u32EPA_MXP);
            outp32(DMA_CTRL_STS, inp32(DMA_CTRL_STS)|0x00000020);
            while(usbdInfo.USBModeFlag)
            {
                if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
                    break;
            }
            if(g_u32EPA_MXP == EPA_OTHER_MAX_PKT_SIZE)

                g_u32BytesInPageBuf -= g_u32EPA_MXP;
        }
    }

    pCmd->u8Cmd        = u8Cmd;
    pCmd->u32Signature = u32PageCnt;
}
```

Command Parser - ProcessCommand() is called in HID_GetOutReport function.

```
INT32 ProcessCommand(UINT32 u32BufferLen)
{
    UINT32 u32sum;
    while(usbdInfo.USBModeFlag)
    {
        if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
            break;
    }
    /* Read CMD for OUT Endpoint */
    outp32(DMA_CTRL_STS, 0x02);    /* bulk out, dma write, ep2 */
    outp32(AHB_DMA_ADDR, (UINT32)pCmd);
    outp32(DMA_CNT, u32BufferLen);
    outp32(DMA_CTRL_STS, inp32(DMA_CTRL_STS)|0x00000020);
    while(usbdInfo.USBModeFlag)
    {
        if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
            break;
    }

    /* Check size */
    if((pCmd->u8Size > sizeof(gCmd)) || (pCmd->u8Size > u32BufferLen))
        return -1;
```

```
    /* Check signature */
    if(pCmd->u32Signature != HID_CMD_SIGNATURE)
        return -1;

    /* Calculate checksum & check it*/
    u32sum = CalCheckSum((UINT8 *)pCmd, pCmd->u8Size);
    if(u32sum != pCmd->u32Checksum)
        return -1;

    switch(pCmd->u8Cmd)
    {
        case HID_CMD_ERASE:
        {
            HID_CmdEraseBlocks(pCmd);
            break;
        }
        case HID_CMD_READ:
        {
            HID_CmdReadPages(pCmd);
            break;
        }
        case HID_CMD_GET_STS:
        {
            HID_CmdGetStatus(pCmd);
            break;
        }
        case HID_CMD_GET_START_BLOCK:
        {
            GetInfo();
            HID_CmdGetStartBlock(pCmd);
            break;
        }
        case HID_CMD_UPDATE:
        {
            /* TODO: To erase the Block of storage */
            if(g_EndTagBlock!= 0)
            {
                sysprintf("\rErase Block %d\n",g_EndTagBlock);
                spiFlashEraseBlockNb(g_EndTagBlock * BLOCK_SIZE);
            }
            break;
```

```
        }
        case HID_CMD_GET_VER:
        {
            GetInfo();
            HID_CmdGetVersion(pCmd);
            break;
        }
        case HID_CMD_WRITE:
        {
            HID_CmdWritePages(pCmd);
            break;
        }
        case HID_CMD_EXIT:
        {
            sysEnableWatchDogTimer();
            sysEnableWatchDogTimerReset();
            while(1);

        }
        case HID_CMD_SET_PARAM:
        {
            break;
        }
        case HID_CMD_GET_PARAM:
        {
            HID_CmdGetParameter(pCmd);
            break;
        }
        case HID_CMD_IMAGE_WRITE:
        {
            HID_CmdImageWrite(pCmd);
            break;
        }
        case HID_CMD_TEST:
        {
            HID_CmdTest(pCmd);
            break;
        }
        default:
            return -1;
    }
```

```
    return 0;
}
```

2.3 HID Vendor Commands

HID_CmdEraseBlocks calls spiFlashEraseBlockNb to erase certain block.

```
INT32 HID_CmdEraseBlocks(CMD_T *pCmd)
{
    UINT32 u32StartBlock;

    u32StartBlock = pCmd->u32Arg1;

    sysprintf("\rErase command - Block: %d\n", u32StartBlock);

    /* TODO: To erase the Block of storage */
    spiFlashEraseBlockNb(u32StartBlock * BLOCK_SIZE);

    /* To note the command has been done */
    pCmd->u8Cmd = HID_CMD_NONE;

    return 0;
}
```

HID_CmdReadPages calls spiFlashRead to read certain page and sends data to host.

```
INT32 HID_CmdReadPages(CMD_T *pCmd)
{
    UINT32 u32StartPage;
    UINT32 u32Pages;

    u32StartPage = pCmd->u32Arg1;
    u32Pages     = pCmd->u32Arg2;

    sysprintf("\rRead command - Start page: %d    Pages Numbers: %d\n", u32StartPage,
                u32Pages);

    if(u32Pages)
    {
        /* Update data to page buffer to upload */
        spiFlashRead(u32StartPage * SPI_PAGE_SIZE, g_u32ReadWriteSize ,
                    (UINT32 *) g_u8PageBuff);
        /* The signature word is used as page counter */
        if(g_u32EPA_MXP == EPA_MAX_PKT_SIZE)
        {
            pCmd->u32Signature = 2;
            g_u32BytesInPageBuf = 0;
```

```
        }
        else
        {
            g_u32BytesInPageBuf = SPI_PAGE_SIZE;
            /* The signature word is used as page counter */
            pCmd->u32Signature = 1;
            g_u32BytesInPageBuf -= g_u32EPA_MXP;

        }
        while(usbdInfo.USBModeFlag)
        {
            if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
                break;
        }
        /* Trigger HID IN */
        outp32(DMA_CTRL_STS, 0x11);    /* bulk in, dma read, ep1 */
        outp32(AHB_DMA_ADDR, (UINT32)g_u8PageBuff);
        outp32(DMA_CNT, g_u32EPA_MXP);
        outp32(DMA_CTRL_STS, inp32(DMA_CTRL_STS)|0x00000020);
        while(usbdInfo.USBModeFlag)
        {
            if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
                break;
        }
    }

    return 0;
}
```

HID_CmdGetStatus calls usiCheckBusyNb to get SPI flash status and sends the status to host.

```
INT32 HID_CmdGetStatus(CMD_T *pCmd)
{
    *((unsigned int *)g_Temp) = usiCheckBusyNb();
    while(usbdInfo.USBModeFlag)
    {
        if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
            break;
    }
    /* Trigger HID IN */
    outp32(DMA_CTRL_STS, 0x11);    /* bulk in, dma read, ep1 */
    outp32(AHB_DMA_ADDR, (UINT32)g_Temp);
    outp32(DMA_CNT, g_u32EPA_MXP);
```

```
    outp32(DMA_CTRL_STS, inp32(DMA_CTRL_STS)|0x00000020);
    while(usbdInfo.USBModeFlag)
    {
        if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
            break;
    }
    return 0;
}
```

HID_CmdGetStartBlock sends the value of g_UpdateStartBlock that is get from image list.

```
INT32 HID_CmdGetStartBlock(CMD_T *pCmd)
{
    *((unsigned int *)g_Temp) = g_UpdateStartBlock;
    while(usbdInfo.USBModeFlag)
    {
        if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
            break;
    }
    /* Trigger HID IN */
    outp32(DMA_CTRL_STS, 0x11);    /* bulk in, dma read, ep1 */
    outp32(AHB_DMA_ADDR, (UINT32)g_Temp);
    outp32(DMA_CNT, g_u32EPA_MXP);
    outp32(DMA_CTRL_STS, inp32(DMA_CTRL_STS)|0x00000020);
    while(usbdInfo.USBModeFlag)
    {
        if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
            break;
    }
    return 0;
}
```

HID_CmdGetVersion sends the value of g_Version that is get from Updated Firmware Header.

```
INT32 HID_CmdGetVersion(CMD_T *pCmd)
{
    sysStopTimer(TIMER0);
    *((unsigned int *)g_Temp) = g_Version;
    while(usbdInfo.USBModeFlag)
    {
        if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
            break;
    }
    /* Trigger HID IN */
```

```
    outp32(DMA_CTRL_STS, 0x11);    /* bulk in, dma read, ep1 */
    outp32(AHB_DMA_ADDR, (UINT32)g_Temp);
    outp32(DMA_CNT, g_u32EPA_MXP);
    outp32(DMA_CTRL_STS, inp32(DMA_CTRL_STS)|0x00000020);
    while(usbdInfo.USBModeFlag)
    {
        if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
            break;
    }
    return 0;
}
```

HID_CmdWritePages sets to the member "u32Signature" of pCmd and g_u32BytesInPageBuf to 0 to start the writing page flow. It will receive and write data to SPI flash in HID_GetOutReport.

```
INT32 HID_CmdWritePages(CMD_T *pCmd)
{
    UINT32 u32StartPage;
    UINT32 u32Pages;

    u32StartPage = pCmd->u32Arg1;
    u32Pages     = pCmd->u32Arg2;

    sysprintf("\rWrite command - Start page: %d    Pages Numbers: %d\n", u32StartPage,
              u32Pages);

    g_u32BytesInPageBuf = 0;

    /* The signature is used to page counter */
    pCmd->u32Signature = 0;

    return 0;
}
```

HID_CmdGetParameter sends the value of the array - g_Parameter according to u32Arg1 & u32Arg2.

```
void GetInfo(UINT32 u32Arg1, UINT32 u32Arg2, UINT32 u32Address)
{
     *((unsigned int *)u32Address) = g_Parameter[u32Arg1][u32Arg2];
     sysprintf("\rGetInfo %d %d = %d\n",u32Arg1, u32Arg2, g_Parameter[u32Arg1][u32Arg2]);

}
INT32 HID_CmdGetParameter(CMD_T *pCmd)
```

```
{
    GetInfo(pCmd->u32Arg1, pCmd->u32Arg2, (UINT32)g_Temp);

    while(usbdInfo.USBModeFlag)
    {
        if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
            break;
    }
    /* Trigger HID IN */
    outp32(DMA_CTRL_STS, 0x11);    /* bulk in, dma read, ep1 */
    outp32(AHB_DMA_ADDR, (UINT32)g_Temp);
    outp32(DMA_CNT, g_u32EPA_MXP);
    outp32(DMA_CTRL_STS, inp32(DMA_CTRL_STS)|0x00000020);
    while(usbdInfo.USBModeFlag)
    {
        if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
            break;
    }
    return 0;
}
```

HID_CmdSetParameter sends the value of the array - g_Parameter according to u32Arg1 & u32Arg2.

```
void SetInfo(UINT32 u32Arg1, UINT32 u32Arg2, UINT32 u32Address)
{
     g_Parameter[u32Arg1][u32Arg2] = *((unsigned int *)u32Address);
     sysprintf("\rSetInfo %d %d = %d\n",u32Arg1, u32Arg2, g_Parameter[u32Arg1][u32Arg2]);
}
INT32 HID_CmdSetParameter(CMD_T *pCmd)
{
    while(usbdInfo.USBModeFlag)
    {
        if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
            break;
    }
    /* Trigger HID IN */
    outp32(DMA_CTRL_STS, 0x02);    /* bulk out, dma read, ep2 */
    outp32(AHB_DMA_ADDR, (UINT32)g_Temp);
    outp32(DMA_CNT, g_u32EPA_MXP);
    outp32(DMA_CTRL_STS, inp32(DMA_CTRL_STS)|0x00000020);
    while(usbdInfo.USBModeFlag)
    {
```

```
        if((inp32(DMA_CTRL_STS) & 0x00000020) == 0)
            break;
    }
    SetInfo(pCmd->u32Arg1, pCmd->u32Arg2, (UINT32)g_Temp);
    return 0;
}
```

HID_CmdImageWrite sets to the member "u32Signature" of pCmd and g_u32BytesInJPEGBuf to 0 to start the image writing flow. It will receive and write data to display buffer in HID_GetOutReport.

```
INT32 HID_CmdImageWrite(CMD_T *pCmd)
{
    UINT32 u32Size;

    u32Size     = pCmd->u32Arg1;

    sysprintf("\rImage Write command - u32Size: %d\n", u32Size);

    g_u32BytesInJPEGBuf = 0;

    /* The signature is used to data counter */
    pCmd->u32Signature = 0;
    return 0;
}
```

2.4 Report Descriptor

HID_ DeviceReportDescriptor and HID_ DeviceReportDescriptor_FS arrays include the HID Report Descriptor for HID function.

```
#if defined (__GNUC__)
UINT8  HID_DeviceReportDescriptor[] __attribute__((aligned(4))) =
#else
__align(4) UINT8  HID_DeviceReportDescriptor[] =
#endif
{
    0x06, 0x06, 0xFF,       /* USAGE_PAGE (Vendor Defined)*/
    0x09, 0x01,             /* USAGE (0x01)*/
    0xA1, 0x01,             /* COLLECTION (Application)*/
    0x15, 0x00,             /* LOGICAL_MINIMUM (0)*/
    0x26, 0xFF, 0x00,       /* LOGICAL_MAXIMUM (255)*/
    0x75, 0x08,             /* REPORT_SIZE (8)*/
```

```
    0x96, 0x00, 0x02,       /* REPORT_COUNT*/
    0x09, 0x01,
    0x81, 0x02,             /* INPUT (Data,Var,Abs)*/
    0x96, 0x00, 0x02,       /* REPORT_COUNT*/
    0x09, 0x01,
    0x91, 0x02,             /* OUTPUT (Data,Var,Abs)*/
    0x95, 0x08,             /* REPORT_COUNT (8) */
    0x09, 0x01,
    0xB1, 0x02,             /* FEATURE */
    0xC0,                   /* END_COLLECTION*/
};

#if defined (__GNUC__)
UINT8  HID_DeviceReportDescriptor_FS[] __attribute__((aligned(4))) =
#else
__align(4) UINT8  HID_DeviceReportDescriptor_FS[] =
#endif
{
    0x06, 0x00, 0xFF,   /* Usage Page = 0xFF00 (Vendor Defined Page 1) */
    0x09, 0x01,         /* Usage (Vendor Usage 1) */
    0xA1, 0x01,         /* Collection (Application) */
    0x19, 0x01,         /* Usage Minimum */
    0x29, 0x40,         /* 64 input usages total (0x01 to 0x40) */
    0x15, 0x00,         /* Logical Minimum (data bytes in the report may have minimum
                           value = 0x00) */
    0x26, 0xFF, 0x00,   /* Logical Maximum (data bytes in the report may have maximum
                           value = 0x00FF = unsigned 255) */
    0x75, 0x08,         /* Report Size: 8-bit field size */
    0x95, 0x40,         /* Report Count: Make sixty-four 8-bit fields (the next time the
                           parser hits an "Input", "Output", or "Feature" item) */
    0x81, 0x00,         /* Input (Data, Array, Abs): Instantiates input packet fields
                           based on the above report size, count, logical min/max, and
                           usage./*
    0x19, 0x01,         /* Usage Minimum */
    0x29, 0x40,         /* 64 output usages total (0x01 to 0x40) */
    0x91, 0x00,         /* Output (Data, Array, Abs): Instantiates output packet fields.
                           Uses same report size and count as "Input" fields, since
                           nothing new/different was specified to the parser since the
                           "Input" item. */
    0xC0                /* End Collection */
};
```

2.5 Configuration Descriptor

USB host requests configuration descriptor for device enumeration. HID_ConfigurationBlock HID_ConfigurationBlock and HID_ConfigurationBlock_FS are the configuration descriptor which contains description of one interface. Field “bNumInterfaces” be set as 1 to inform that there are one interface in the HID device. Interface descriptor, HID descriptor and endpoint descriptor are listed sequentially in following.

```
#if defined (__GNUC__)
static UINT8 HID_ConfigurationBlock[] __attribute__((aligned(4))) =
#else
__align(4) static UINT8 HID_ConfigurationBlock[] =
#endif
{
    LEN_CONFIG,     /* bLength */
    DESC_CONFIG,    /* bDescriptorType */
    /* wTotalLength */
    (LEN_CONFIG + LEN_INTERFACE + LEN_HID + LEN_ENDPOINT * 2) & 0x00FF,
    (((LEN_CONFIG + LEN_INTERFACE + LEN_HID + LEN_ENDPOINT * 2) & 0xFF00) >> 8),
    0x01,           /* bNumInterfaces */
    0x01,           /* bConfigurationValue */
    0x00,           /* iConfiguration */
    0x80 | (USBD_SELF_POWERED << 6) | (USBD_REMOTE_WAKEUP << 5),/* bmAttributes */
    USBD_MAX_POWER,         /* MaxPower */

    /* I/F descr: HID */
    LEN_INTERFACE,  /* bLength */
    DESC_INTERFACE, /* bDescriptorType */
    0x00,           /* bInterfaceNumber */
    0x00,           /* bAlternateSetting */
    0x02,           /* bNumEndpoints */
    0x03,           /* bInterfaceClass */
    0x00,           /* bInterfaceSubClass */
    0x00,           /* bInterfaceProtocol */
    0x00,           /* iInterface */

    /* HID Descriptor */
    LEN_HID,        /* Size of this descriptor in UINT8s. */
    DESC_HID,       /* HID descriptor type. */
    0x10, 0x01,     /* HID Class Spec. release number. */
    0x00,           /* H/W target country. */
```

```
    0x01,           /* Number of HID class descriptors to follow. */
    DESC_HID_RPT,   /* Descriptor type. */
    /* Total length of report descriptor. */
    sizeof(HID_DeviceReportDescriptor) & 0x00FF,
    ((sizeof(HID_DeviceReportDescriptor) & 0xFF00) >> 8),

    /* EP Descriptor: interrupt in. */
    LEN_ENDPOINT,                       /* bLength */
    DESC_ENDPOINT,                      /* bDescriptorType */
    (INT_IN_EP_NUM | EP_INPUT),         /* bEndpointAddress */
    EP_INT,                             /* bmAttributes */
    /* wMaxPacketSize */
    EPA_MAX_PKT_SIZE & 0x00FF,
    ((EPA_MAX_PKT_SIZE & 0xFF00) >> 8),
    HID_DEFAULT_INT_IN_INTERVAL,        /* bInterval */

    /* EP Descriptor: interrupt out. */
    LEN_ENDPOINT,                       /* bLength */
    DESC_ENDPOINT,                      /* bDescriptorType */
    (INT_OUT_EP_NUM | EP_OUTPUT),       /* bEndpointAddress */
    EP_INT,                             /* bmAttributes */
    /* wMaxPacketSize */
    EPB_MAX_PKT_SIZE & 0x00FF,
    ((EPB_MAX_PKT_SIZE & 0xFF00) >> 8),
    HID_DEFAULT_INT_IN_INTERVAL         /* bInterval */
};

#if defined (__GNUC__)
static UINT8 HID_ConfigurationBlock_FS[] __attribute__((aligned(4))) =
#else
__align(4) static UINT8 HID_ConfigurationBlock_FS[] =
#endif
{
    LEN_CONFIG,     /* bLength */
    DESC_CONFIG,    /* bDescriptorType */
    /* wTotalLength */
    (LEN_CONFIG + LEN_INTERFACE + LEN_HID + LEN_ENDPOINT * 2) & 0x00FF,
    (((LEN_CONFIG + LEN_INTERFACE + LEN_HID + LEN_ENDPOINT * 2) & 0xFF00) >> 8),
    0x01,           /* bNumInterfaces */
    0x01,           /* bConfigurationValue */
    0x00,           /* iConfiguration */
```

```
    0x80 | (USBD_SELF_POWERED << 6) | (USBD_REMOTE_WAKEUP << 5),  /* bmAttributes */
    USBD_MAX_POWER, /* MaxPower */

    /* I/F descr: HID */
    LEN_INTERFACE,  /* bLength */
    DESC_INTERFACE, /* bDescriptorType */
    0x00,           /* bInterfaceNumber */
    0x00,           /* bAlternateSetting */
    0x02,           /* bNumEndpoints */
    0x03,           /* bInterfaceClass */
    0x00,           /* bInterfaceSubClass */
    0x00,           /* bInterfaceProtocol */
    0x00,           /* iInterface */

    /* HID Descriptor */
    LEN_HID,        /* Size of this descriptor in UINT8s. */
    DESC_HID,       /* HID descriptor type. */
    0x10, 0x01,     /* HID Class Spec. release number. */
    0x00,           /* H/W target country. */
    0x01,           /* Number of HID class descriptors to follow. */
    DESC_HID_RPT,   /* Descriptor type. */
    /* Total length of report descriptor. */
    sizeof(HID_DeviceReportDescriptor_FS) & 0x00FF,
    ((sizeof(HID_DeviceReportDescriptor_FS) & 0xFF00) >> 8),

    /* EP Descriptor: interrupt in. */
    LEN_ENDPOINT,                       /* bLength */
    DESC_ENDPOINT,                      /* bDescriptorType */
    (INT_IN_EP_NUM | EP_INPUT),         /* bEndpointAddress */
    EP_INT,                             /* bmAttributes */
    /* wMaxPacketSize */
    EPA_OTHER_MAX_PKT_SIZE & 0x00FF,
    ((EPA_OTHER_MAX_PKT_SIZE & 0xFF00) >> 8),
    HID_DEFAULT_INT_IN_INTERVAL,        /* bInterval */

    /* EP Descriptor: interrupt out. */
    LEN_ENDPOINT,                       /* bLength */
    DESC_ENDPOINT,                      /* bDescriptorType */
    (INT_OUT_EP_NUM | EP_OUTPUT),       /* bEndpointAddress */
    EP_INT,                             /* bmAttributes */
    /* wMaxPacketSize */
```

```
    EPB_OTHER_MAX_PKT_SIZE & 0x00FF,
    ((EPB_OTHER_MAX_PKT_SIZE & 0xFF00) >> 8),
    HID_DEFAULT_INT_IN_INTERVAL         /* bInterval */
};
```

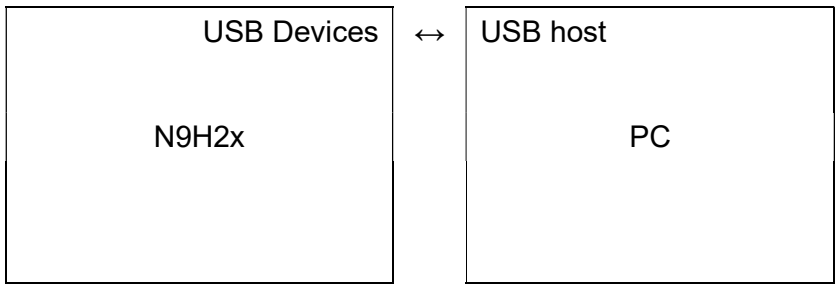
3 Software and Hardware Environment

Software Environment

- BSP version
  - N9H2x Series BSP
- IDE version
  - Keil uVersion4.54

Hardware Environment

- Circuit components
  - N9H2x Development Board
  - USB micro USB cable
- Diagram



4 Directory Information

- N9H2x BSP
  - USBD
    - HID_Transfer_FirmwareUpdate — Source file of example code
      - Doc — Document
      - HID_Transfer — Source file of HID Transfer code
      - SpiLoader_VerCtrl — Source file of SpiLoader with Version Control
      - WindowsTool — PC Tool source code

5 How to Execute Example Code

1. This project supports Keil uVersion 4.54 or above.
2. Browsing into sample code folder (SpiLoader_VerCtrl & HID_Transfer) by Directory Information (section 4) and double click project file.
3. Enter Keil compile mode and build
4. Burn SpiLoader_VerCtrl & HID_Transfer.bin to SPI flash
5. Connect N9H2x to PC and power on N9H2x.
6. Run Command Line PC Tool

6 Revision History

Date	Revision	Description
Feb.26, 2021	1.00	1. Initially issued.