HiFB

Development Guide

Issue **05**

Date **2013-06-21**

HiSilicon Technologies Co., Ltd.

Address: Huawei Industrial Base
Bantian, Longgang
Shenzhen 518129
People's Republic of China

Website: http://www.hisilicon.com

Email: support@hisilicon.com

About This Document

Purpose

As a module of the HiSilicon digital media processing platform (HiMPP), the HiSilicon frame buffer (HiFB) is used to manage the graphics layers. The HiFB is developed based on the Linux frame buffer. Besides the basic functions provided by the Linux frame buffer, the HiFB also provides extended functions for controlling graphics layers such as the interlayer alpha and origin setting. This document describes how to load the HiFB, and how to develop products or solution by using the HiFB for the first time.

Related Version

The following table lists the product version related to this document.

Product Name	Version
Hi3531	V100
Hi3532	V100
Hi3521	V100
Hi3520A	V100
Hi3518	V100
Hi3520D	V100
Hi3515A	V100
Hi3515C	V100

Intended Audience

This document is intended for:

- Technical support personnel
- Board hardware development engineers

Change History

Changes between document issues are cumulative. Therefore, the latest document issue contains all changes made in previous issues.

Issue 05 (2013-06-21)

This issue is the fifth official release, which incorporates the following changes:

The descriptions related to the Hi3515C are added.

Issue 04 (2013-04-03)

This issue is the fourth official release, which incorporates the following changes:

Chapter 2 Loading Drivers

In section 2.2, the note "the hardware cursor is preferred" is added.

Issue 03 (2013-03-31)

This issue is the third official release, which incorporates the following changes:

The descriptions related to the Hi3520D/Hi3515A are added.

Issue 02 (2012-09-20)

This issue is the second official release, which incorporates the following changes:

Chapter 1 Overview

Below Figure 1-4, the sentences "When drawn contents are transferred from the user canvas buffer to the display buffer, scaling and anti-flicker are supported. Note that scaling is not supported during this process for the Hi3518." are added.

Issue 01(2012-08-30)

This issue is the first official release, which incorporates the following changes:

Chapter 1 Overview

In section 1.2, the descriptions of the Hi3520A and Hi3521 are combined, and the original Table 1-3 is deleted.

Chapter 2 Loading Drivers

In section 2.2, the descriptions of the Hi3520A and Hi3521 are combined.

Issue 00B50 (2012-08-09)

This issue is the sixth draft release, which incorporates the following changes:

Chapter 1 Overview

In section 1.2, the note is updated in the "Standard and Extended Functions" section.

Issue 00B40 (2012-07-30)

This issue is the fifth draft release, which incorporates the following changes:

Chapter 1 Overview

In section 1.2, the number of graphics layers managed by the Hi3520A is changed from 1 to, the SD device supported by the Hi3520A is changed from SD0 to SD1, and descriptions of the Hi3520A are updated.

Chapter 2 Loading Drivers

In section 2.2, the descriptions of u32VcmpBufNum are added.

Issue 00B30 (2012-06-08)

This issue is the fourth draft release, which incorporates the following changes:

The descriptions of the Hi3520A are added.

Issue 00B20 (2012-04-20)

This issue is the third draft release, which incorporates the following changes:

The descriptions of the Hi3521 are added.

Issue 00B10 (2012-01-15)

This issue is the second draft release, which incorporates the following changes:

Chapter 1 Overview

The note below Figure 1-4 is updated.

Issue 00B01 (2011-11-10)

This issue is the draft release.

Contents

About This Document i

1 Overview 1

1.1 HiFB Overview 1

1.1.1 System Architecture 1

1.1.2 Application Scenarios 1

1.2 Comparing the HiFB with the Linux FB 2

1.3 Related Document 7

2 Loading Drivers 8

2.1 Principle 8

2.2 Parameter Configuration 8

2.3 Configuration Examples 10

2.4 Exception 10

3 Initial Development Application 11

3.1 Development Process 11

3.2 Examples 12

Figures

Figure 1-1 System architecture of the HiFB 1
Figure 1-2 Non-buffer mode 6
Figure 1-3 Single-buffer mode 6
Figure 1-4 Dual-buffer mode 7
Figure 3-1 Development process of the HiFB 11

Tables

Table 1-1 Relationships among the FB device files, graphics layers, and output devices of the Hi3531 2

Table 1-2 Relationships among the FB device files, graphics layers, and output devices of the Hi3521, Hi3520A, Hi3520D, Hi3515A, or Hi3515C 3

Table 1-3 Relationship among the FB device file, graphics layer, and output device of the Hi3518 4

Table 1-4 Function differences among chips 4

Table 3-1 Tasks of the HiFB completed in each development phase 12

1 Overview

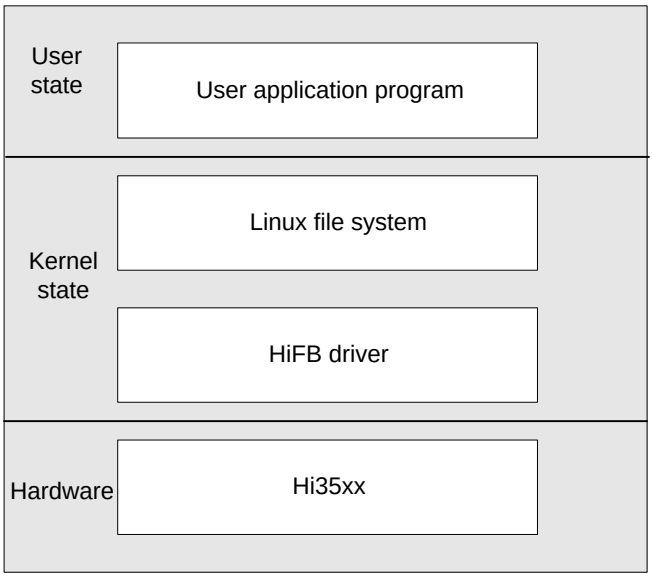
1.1 HiFB Overview

As a module of the HiMPP, the HiFB is used to manage the graphics layers. The HiFB provides not only the basic functions of the Linux FB, but also some extended functions such as the interlayer colorkey, interlayer colorkey mask, interlayer Alpha, and origin offset.

1.1.1 System Architecture

The application program uses the HiFB through the Linux file system. Figure 1-1 shows the system architecture of the HiFB.

Figure 1-1 System architecture of the HiFB



1.1.2 Application Scenarios

The HiFB applies to the following scenarios:

- MiniGUI window system

The MiniGUI window system supports the Linux FB. With a slight modification, the MiniGUI window system can be ported to the Hi35xx quickly.

- Other application programs based on the Linux FB

  Without modification or with a slight modification, the Linux-FB-based application programs can be ported to the Hi35xx quickly.

1.2 Comparing the HiFB with the Linux FB

Managing Graphics Layers

For the Linux FB, a sub device number corresponds to a video device. For the HiFB, a sub device number corresponds to a graphics layer. The HiFB manages multiple graphics layers and the number of the graphics layers is determined by the chip.

NOTE

- For the Hi3531, the HiFB manages at most seven graphics layers: G0 to G6. G5 and G6 are cursor layers. The corresponding device files are **/dev/fb0**–**/dev/fb6**. The output devices supported by the Hi3531 allow the graphics layers to be overlaid on the following output devices: HD output device 0 (DHD0), HD output device 1 (DHD1), standard definition (SD) output device 0 (DSD0), and SD device 1 (DSD1). Table 1-1 describes the relationship among the FB device files, graphics layers, and output devices of the Hi3531.
- For the Hi3532, the HiFB manages at most two graphics layers: G0 and G5 (G5 is the cursor layer). The device files corresponding to G0 and G5 are **/dev/fb0** and **/dev/fb5** respectively. The Hi3532 supports only DHD0. G0 and G5 can be overlaid on DHD0 simultaneously without binding. The other functions of G0 and G5 are the same as those of the Hi3531.
- For the Hi3521, Hi3520A, or Hi3520D/Hi3515A/Hi3515C, the HiFB manages at most four graphics layers: G0 to G3 (G3 is the cursor layer). The corresponding device files are **/dev/fb0**–**/dev/fb3**. The output devices supported by the Hi3521, Hi3520A, or Hi3520D/Hi3515A/Hi3515C allow the graphics layers to be overlaid on the following output devices: DHD0, DSD0, and DSD1. Table 1-2 describes the relationship among the FB device files, graphics layers, and output devices of the Hi3521.
- For the Hi3518, the HiFB manages only one graphics layer. The corresponding device file is **/dev/fb0**. The output devices supported by the Hi3518 allow the graphics layers to be overlaid on DSD1. Table 1-3 describes the relationship among the FB device file, graphics layer, and output device of the Hi3518.

Table 1-1 Relationships among the FB device files, graphics layers, and output devices of the Hi3531

FB Device File	Graphics Layer	Corresponding Display Device
/dev/fb0	G0	G0 is displayed only on DHD0.
/dev/fb1	G1	G1 is displayed only on DHD1.
/dev/fb2	G2	G2 is displayed only on DSD0.
/dev/fb3	G3	G3 is displayed only on DSD1.

FB Device File	Graphics Layer	Corresponding Display Device
/dev/fb4, /dev/fb5, /dev/fb6	G4, cursor layer0 (G5), cursor layer1 (G6)	G4–G6 are displayed on DHD0, DHD1, DSD0, or DSD1. You can specify the display device by calling the related binding interfaces. G5 and G6 are the cursor layers and are always the uppermost layers when multiple layers are overlaid on a display device. G4 is the layer under G5. If the video layer, G0, G4, and G5 are overlaid on DHD0, the overlaid sequence from the bottom up is: video layer, G0, G4, and G5. G5 and G6 can act as the hardware cursor layers or software cursor layers, which is determined by the module loading parameter **softcursor**. When G5 and G6 are the hardware cursor layers, they are used in the same way as that of other graphics layers are used. When G5 and G6 are the software cursor layers, they must be operated by using the software cursor dedicated interfaces that are provided by the HiFB.

Table 1-2 Relationships among the FB device files, graphics layers, and output devices of the Hi3521, Hi3520A, Hi3520D, Hi3515A, or Hi3515C

FB Device File	Graphics Layer	Corresponding Display Device
/dev/fb0	G0	G0 is displayed only on DHD0.
/dev/fb1	G1	G1 is displayed only on DSD0.
/dev/fb2	G2	G2 is displayed only on DSD1.
/dev/fb3	cursor layer 0 (G3)	G3 is displayed on DHD0, DHD1, DSD0, or DSD1. You can specify the display device by calling the related binding interfaces. G3 are the cursor layers and are always the uppermost layers when multiple layers are overlaid on a display device. If the video layer, G0, and G3 are overlaid on DHD0, the overlaid sequence from the bottom up is: video layer, G0, and G3. G3 can act as the hardware cursor layers or software cursor layers, which is determined by the module loading parameter **softcursor**. For details, see section 2.2.2. When G3 is the hardware cursor layers, they are used in the same way as that of other graphics layers are used. When G3 is the software cursor layers, they must be operated by using the software cursor dedicated interfaces that are provided by the HiFB.

Table 1-3 Relationship among the FB device file, graphics layer, and output device of the Hi3518

FB Device File	Graphics Layer	Corresponding Display Device
/dev/fb0	G3	G2 is displayed only on DSD1.

By setting the module loading parameter, you can configure the HiFB to manage one or multiple graphics layers and operate graphics layers as easily as files.

Differences Among Chips

Table 1-4 describes the function differences among chips.

Table 1-4 Function differences among chips

Chip	Supported Graphics Layer	Compression	Colorkey	Binding Relationship
Hi3531	G0–G6	Only G0 to G4 support compression.	Only the cursor layers G5 and G6 support colorkey.	G0 to G3 are fixed bound to DHD0, DHD1, DSD0, and DSD1 respectively. G4 to G6 can be dynamically bound, but G5 and G6 cannot be bound to a device at the same time.
Hi3532	G0 and G5	Compression is not supported.	Only G5 supports colorkey.	G0 and G5 are fixed bound to a device.
Hi3521/Hi3520A/Hi3520D/Hi3515A/Hi3515C	G0–G3	Only G0 supports compression.	All layers support colorkey.	G0 to G2 are fixed bound to DHD0, DSD0, and DSD1 respectively. G3 can be dynamically bound.
Hi3518	G3	Compression is not supported.	Supported	G3 is fixed bound to DSD1

Controlling the Timing

The Linux FB provides the controlling modes (hardware support required) such as the synchronous timing, scanning mode, and synchronous signal mechanism. The contents of the physical display buffer are displayed through different output devices such as the PC monitor, TV, and LCD. At present, the HiFB does not support the controlling modes such as synchronous timing, scanning mode, and synchronous signal mechanism.

Standard and Extended Functions

The HiFB supports the following standard functions of the Linux FB:

- Create/Destroy a map between the physical display buffer and the virtual memory.
- Operate the physical display buffer like a common file.
- Set the hardware display resolution and the pixel format. The maximum resolution and the pixel format supported by each graphics layer can be obtained through the support capability interface.
- Perform the read, write and display operations from any position of the physical display buffer.
- Set and obtain 256-color palette when the graphics layer supports the index format.

NOTE

For the Hi3531, Hi3532, Hi3521, or the Hi3520A, each graphics layer supports only the ARGB1555 and ARGB8888 formats. The Hi3518 supports only the ARGB1555 and ARGB4444 formats.

The HiFB has the following extended functions:

- Set and obtain the Alpha value of the graphics layer.
- Set and obtain the colorkey values of the graphics layer.
- Set the start position of the current graphics layer (namely, the offset from the screen origin).
- Set and obtain the display state of the current graphics layer (display/hide).
- Set the size of HiFB physical display buffer and manage the number of the graphics layers through the module loading parameters.
- Set and obtain the premultiply mode.
- Set and obtain the status of the compression mode.
- Set and obtain the refresh mode of graphics layers (non-buffer mode, single–buffer mode, and dual-buffer mode).
- Support operations related to the software cursor.

The HiFB does not support the following standard functions of the Linux FB:

- Set and obtain the Linux FB of corresponding console.
- Obtain the real-time information about hardware scanning.
- Obtain the hardware-related information.
- Obtain the hardware synchronous timing.
- Obtain the hardware synchronous signal mechanism.

Refresh Mode of Graphics Layers – Extended FB Mode

The HiFB provides a comprehensive refresh scheme for upper-layer users that is called extended FB mode. You can select an appropriate refresh type based on the system performance, memory size, and graphics display effect. The supported refresh modes include:

- Non-buffer mode (that is HIFB_LAYER_BUF_NONE)

  The canvas buffer for upper-layer users is the display buffer. In this mode, the required memory is reduced, and the refresh speed is the fastest, but users can view the graphics drawing process. The diagram is shown in 0.

- Single-buffer mode (that is HIFB_LAYER_BUF_ONE)

  The display buffer is provided by the HiFB. Therefore, a certain memory is required. In this mode, the display effect and required memory are balanced. However, the picture alias occurs. See Figure 1-3.

- Dual-buffer mode

  The display buffer is provided by the HiFB. Compared with the preceding two modes, the dual-buffer mode requires the most memory, but provides the best display effect. Figure 1-4 shows the dual-buffer mode. The refresh modes include:

  - HIFB_LAYER_BUF_DOUBLE
  - HIFB_LAYER_BUF_DOUBLE_IMMEDIATE

  The difference between these two refresh modes is that the corresponding functions are returned only after the drawn contents are displayed when the HIFB_LAYER_BUF_DOUBLE_IMMEDIATE refresh mode is used.

Figure 1-2 Non-buffer mode

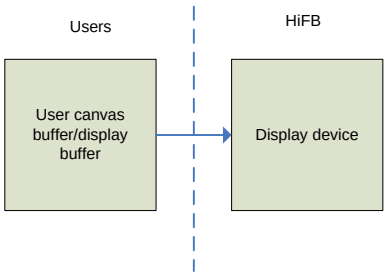


Figure 1-3 Single-buffer mode

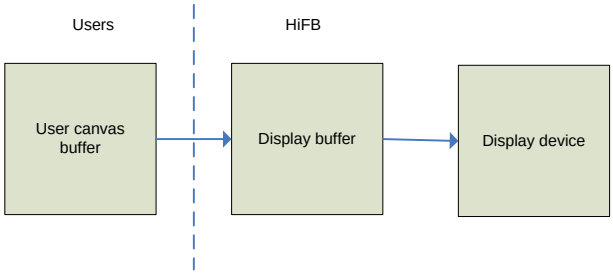
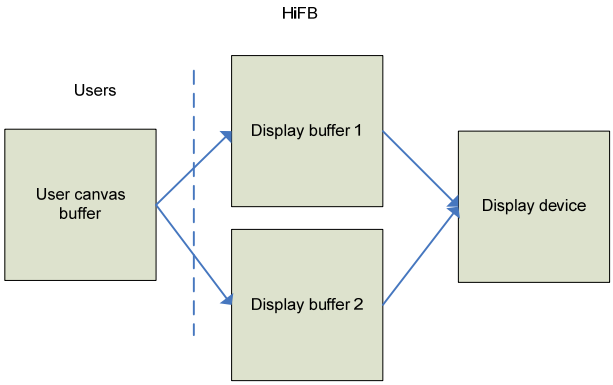


Figure 1-4 Dual-buffer mode



NOTE

The preceding resolutions are canvas resolution (resolution of user canvas buffers), display buffer resolution, and screen display resolution. When drawn contents are transferred from the user canvas buffer to the display buffer, scaling and anti-flicker are supported. Note that scaling is not supported during this process for the Hi3518. When drawn contents are transferred from the display buffer to the display device, scaling and anti-flicker are not supported. Therefore, the display buffer resolution and the screen display resolution are the same.

1.3 Related Document

See the *HiFB API Reference*.

2 Loading Drivers

2.1 Principle

The display attributes of some Linux FB drivers (for example versa), such as resolution, color depth, and timing cannot be changed during the operation. The Linux provides a mechanism that allows the system to transfer options to the Linux FB through the parameters in the case of the kernel booting or module loading. The kernel booting parameters can be set in the kernel loader. For the HiFB driver, only physical video display size can be set in the case of module loading.

When the HiFB driver **hifb.ko** is loaded, ensure that the standard FB driver **fb.ko** has been loaded. If **fb.ko** is not loaded, run **modprobe fb** to load **fb.ko** and **hifb.ko** in sequence.

2.2 Parameter Configuration

The HiFB can be used to set the size of the physical display buffer for the managed graphics layers. The size of the physical display buffer determines the maximum capacity of the physical display buffer used in the HiFB and the virtual resolution. When loading the HiFB module, the size of the physical display buffer is set through the parameter. The size of the physical display buffer cannot be changed after it is set.

video Parameter

```
video="hifb:vram0_size:xxx, vram1_size:xxx,…"
```

NOTE

- Items are separated with commas (,).
- An item and an item value are separated with a colon (:).
- If the size of the physical display buffer corresponding to a graphics layer is not set, the buffer is 0 by default.
- **vram0_size**–**vram6_size** correspond to G0–G6.

Where, **vramn_size: xxx** indicates the size of the physical display buffer configured for the graphics layer n. The buffer size is in the unit of KB.

For the standard FB mode, the relationship between vramn_size and virtual resolution is as follows:

```
Vramn_size * 1024 >= xres_virtual * yres_virtual * bpp;
```

where **xres_virtual * yres_virtual** indicates the virtual resolution, and **bpp** indicates the number of bytes occupied by each pixel.

(2) For the extended FB mode, the required memory depends on the value of **displaysize**, pixel format of the graphics layer, and refresh mode. The relationship is as follows:

```
vramn_size * 1024 >= displaywidth * displayHeight * bpp * BufferMode;
```

Assume that the refresh mode is dual-buffer mode, the resolution is 1280x720, and the pixel format is ARGB8888 format. The required memory of G0 is calculated as follows: vram0_size = 1280 x 720 x 4 x 2 = 7200 KB.

NOTE

The value of **vramn_size** must be a multiple of **PAGE_SIZE** (4 KB). Otherwise, the HiFB rounds up the value to a multiple of **PAGE_SIZE**.

softcursor Parameter

The **softcursor** parameter determines whether the software cursor function is enabled. When **softcursor** is **off**, the software cursor function is disabled. That is, the hardware cursor function is available. After the HiFB driver is loaded, you can confirm whether the software cursor function is enabled.

NOTE

The hardware cursor is preferred.

apszLayerMmzNames Parameter

This parameter determines that the memory used by each graphics layer is allocated from which media memory zone (MMZ). This parameter is a string array and has at most seven values that correspond to fb0–fb6. After the HiFB driver is loaded, the MMZ whose memory is used by each graphics layer can be determined. If this parameter is not set, the anonymous MMZ is used.

u32VcmpBufNum

This parameter defines the number of video compress (VCMP) buffers used by each graphics layer in compression mode. A larger value indicates that more memory is used. When **u32VcmpBufNum** is less than 3, artifacts may occur at the bottom of the displayed picture. The maximum value of **u32VcmpBufNum** is 3, and the default value of **u32VcmpBufNum** is 2.

Default Parameter Values

If the program has no parameter when the HiFB driver is loaded, the default parameter values for the Hi3531, Hi3532 and the Hi3521 are as follows:

- Hi3531

  video="hifb:vram0_size:7200,vram1_size:7200,vram2_size:3240,vram3_size:3240,vram4_size:7200,vram5_size:128,vram6_size:128" softcursor="off"
- Hi3532

  video = "hifb:vram0_size:7200,vram5_size:128" softcursor="off"
- Hi3521/Hi3520A/Hi3520D/Hi3515A/Hi3515C

video="hifb:vram0_size:8100, vram1_size:1620,vram2_size:1620,vram3_size:32" softcursor="off"

- Hi3518

  video = "hifb:vram0_size:1620"

You must configure the graphics layers managed by the HiFB, specify the MMZ whose memory is allocated, and allocate appropriate display buffer for each graphics layer from a global aspect.

2.3 Configuration Examples

The examples of configuring the graphics layers managed by the HiFB are as follows:

NOTE

hifb.ko is the HiFB driver.

- Configure the HiFB to manage one graphics layer

  If the HiFB manages only G0 with the maximum virtual resolution 720 x 576 and with the pixel format ARGB1555, the minimum display buffer of G0 is 720 x 576 x 2 = 829440 = 810 K. The configuration parameter is as follows:

  insmod hifb.ko video="hifb:vram0_size:810, vram2_size:0".

  If the dual-buffer mode is used, the value of vram0_size must be multiplied by 2. That is, the parameters are set as follows:

  insmod hifb.ko video="hifb:vram0_size:1620, vram2_size:0"

- Configure the HiFB to manage multiple graphics layers

  If the HiFB manages G0 and graphics layer 1 with the maximum virtual resolution 720 x 576 and with the pixel format ARGB1555, the minimum display buffer of the two graphics layers is 720 x 576 x 2 = 829440 = 810 K. The configuration parameter is as follows:

  insmod hifb.ko video="hifb:vram0_size:810, vram1_size: 810".

2.4 Exception

If the physical display buffer for a graphics layer is incorrectly configured, the HiFB does not manage the graphics layer.

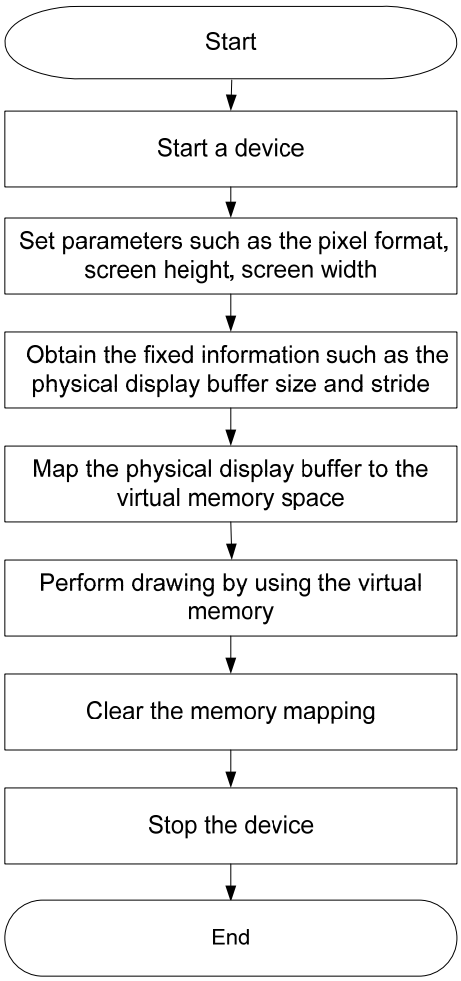
3 Initial Development Application

3.1 Development Process

The HiFB displays two-dimensional images (in the mode of operating on the physical display buffer directly).

Figure 3-1 shows the development process of the HiFB.

Figure 3-1 Development process of the HiFB



To develop the HiFB, perform the following steps:

Step 1 Call the open function to start the HiFB device.

Step 2 Call the ioctl function to set parameters of the HiFB, such as the pixel format, screen height, and screen width. For details, see the *HiFB API Reference*.

Step 3 Call the ioctl function to obtain the fixed information about the HiFB, such as the physical display buffer size and the stride. You can call the ioctl function to use the interlayer colorkey, interlayer colorkey mask, interlayer Alpha, and origin offset provided by the HiFB.

Step 4 Call the mmap function to map the physical display buffer to the virtual memory space.

Step 5 Operate the virtual memory to perform the specific drawing tasks. In this step, you can use the dual-buffer page up/down function provided by the HiFB to implement drawing effects.

Step 6 Call munmap to clear the display buffer mapping.

Step 7 Call the close function to stop the device.

----End

NOTE

The modification of the virtual resolution may change the HiFB fixed information fb_fix_screeninfo::line_length (stride). To ensure that the drawing program runs properly, it is recommended to set the HiFB variable information fb_var_screeninfo and then obtain the HiFB fixed information fb_fix_screeninfo::line_length.

Table 3-1 lists tasks of the HiFB completed in each development phase.

Table 3-1 Tasks of the HiFB completed in each development phase

Phase	Task
Initialization	Set the display attributes and map the physical display buffer.
Drawing	Perform the specific drawing operations.
Termination	Clean up resources.

3.2 Examples

In this example, PAN_DISPLAY is used to consecutively display 15 pictures with the 640 x 352 resolution for the dynamic display effect.

The 15 pictures are stored under hifb/res of the SDK demonstration sample directory. Each file stores the pure data (excluding additional information) in the pixel format of ARGB1555.

The code file in the example is stored under hifb/api_sample_hifb.c of the SDK demonstration sample directory.

```
[Reference Codes]
#include <stdio.h>
```

```
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <linux/fb.h>
#include "hifb.h"

#define IMAGE_WIDTH    640
#define IMAGE_HEIGHT   352
#define IMAGE_SIZE     (640*352*2)
#define IMAGE_NUM      14
#define IMAGE_PATH     "./res/%d.bits"

static struct fb_bitfield g_r16 = {10, 5, 0};
static struct fb_bitfield g_g16 = {5, 5, 0};
static struct fb_bitfield g_b16 = {0, 5, 0};
static struct fb_bitfield g_a16 = {15, 1, 0};

int main()
{
    int fd;
    int i;
    struct fb_fix_screeninfo fix;
    struct fb_var_screeninfo var;
    unsigned char *pShowScreen;
    unsigned char *pHideScreen;
    HIFB_POINT_S stPoint = {40, 112};
    FILE *fp;
    char image_name[128];

    /*1. open FB device overlay 0*/
    fd = open("/dev/fb/0", O_RDWR);
    if(fd < 0)
    {
        printf("open fb0 failed!\n");
        return -1;
    }

    /*2. set the screen original position*/
    if (ioctl(fd, FBIOPUT_SCREEN_ORIGIN_HIFB, &stPoint) < 0)
    {
        printf("set screen original show position failed!\n");
        return -1;
    }
```

```
    /*3. obtain the variable screen info*/
    if (ioctl(fd, FBIOGET_VSCREENINFO, &var) < 0)
    {
        printf("Get variable screen info failed!\n");
       close(fd);
       return -1;
    }

    /*4. modify the variable screen info
         the screen size: IMAGE_WIDTH*IMAGE_HEIGHT
         the virtual screen size: IMAGE_WIDTH*(IMAGE_HEIGHT*2)
         the pixel format: ARGB1555
    */
    var.xres = var.xres_virtual = IMAGE_WIDTH;
    var.yres = IMAGE_HEIGHT;
    var.yres_virtual = IMAGE_HEIGHT*2;

    var.transp= g_a16;
    var.red = g_r16;
    var.green = g_g16;
    var.blue = g_b16;
    var.bits_per_pixel = 16;

    /*5. set the variable screeninfo*/
    if (ioctl(fd, FBIOPUT_VSCREENINFO, &var) < 0)
    {
        printf("Put variable screen info failed!\n");
       close(fd);
       return -1;
    }

    /*6. obtain the fix screen info*/
    if (ioctl(fd, FBIOGET_FSCREENINFO, &fix) < 0)
    {
       printf("Get fix screen info failed!\n");
       close(fd);
       return -1;
    }

    /*7. map the physical display buffer for user use*/
    pShowScreen = mmap(NULL, fix.smem_len, PROT_READ|PROT_WRITE, MAP_SHARED,
fd, 0);
    pHideScreen = pShowScreen + IMAGE_SIZE;
    memset(pShowScreen, 0, IMAGE_SIZE);
```

```
    /*8. load the bitmaps from file to hide screen and set pan display the hide
screen*/
    for(i = 0; i < IMAGE_NUM; i++)
    {
        sprintf(image_name, IMAGE_PATH, i);
        fp = fopen(image_name, "rb");
        if(NULL == fp)
        {
            printf("Load %s failed!\n", image_name);
            close(fd);
            return -1;
        }

        fread(pHideScreen, 1, IMAGE_SIZE, fp);
        fclose(fp);
        usleep(10);
        if(i%2)
        {
            var.yoffset = 0;
            pHideScreen = pShowScreen + IMAGE_SIZE;
        }
        else
        {
            var.yoffset = IMAGE_HEIGHT;
            pHideScreen = pShowScreen;
        }

        if (ioctl(fd, FBIOPAN_DISPLAY, &var) < 0)
        {
            printf("FBIOPAN_DISPLAY failed!\n");
            close(fd);
            return -1;
        }
    }

    printf("Enter to quit!\n");
getchar();

    /*9. close the FB device*/
    close(fd);

    return 0;
}
```