HiFB

Development Guide

Issue	**05**
Date	**2016-05-15**

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.

NOTE

Unless otherwise specified, this document applies to the Hi3516A, Hi3516D, Hi3518E V200/V201, and Hi3516C V200.

Unless otherwise stated, the contents of Hi3516C V200 are consistent with those of Hi3518E V200/Hi3518E V201.

Related Version

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

Product Name	Version
Hi3516A	V100
Hi3516D	V100
Hi3518E	V200
Hi3518E	V201
Hi3516C	V200
Hi3519	V100
Hi3519	V101

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 (2016-05-15)

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

The contents related to the Hi3519 V101 are added.

Issue 04 (2015-12-28)

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

Section 2.2, the contents related to the softcursor parameter is deleted.

Issue 03 (2015-08-20)

The contents related to the Hi3519 V100 are added.

Issue 02 (2015-06-23)

The contents related to the Hi3518E V200/V201 and Hi3516C V200 are added.

Sections 1.2 and 2.2 are modified.

Issue 01 (2014-12-20)

The contents related to the Hi3516D are added.

Issue 00B01 (2014-09-14)

This issue is the first draft release.

Contents

Figures

Tables

1 Overview

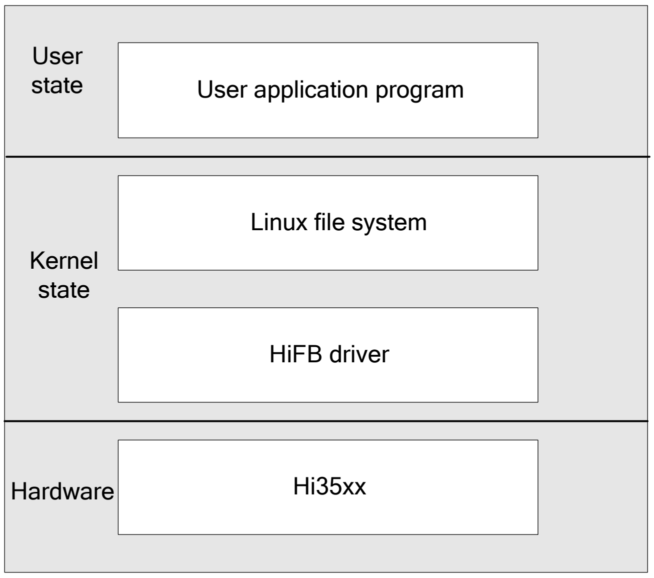
1.1 HiFB Overview

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

The Hi3516A/Hi3518E V200/Hi3519 V100/Hi3519 V101 HiFB manages at most one graphics layer (G0) that corresponds to devices files **/dev/fb0.** The Hi3516A/Hi3518E V200/Hi3519 V100/Hi3519 V101 supports one SD VO device (DSD0) that can be overlaid with graphics layers. Table 1-1 describes the mapping between the graphics layer and VO device of the Hi3516A/Hi3518 EV200/Hi3519 V100/Hi3519 V101.

Table 1-1 Mapping between the graphics layers and VO devices of the Hi3516A/Hi3518E V200/Hi3519 V100/Hi3519 V101

FB Device File	Graphics Layer	Corresponding Display Device
/dev/fb0	G0	G0 is displayed only on DSD0.

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 1-2 describes the function differences among chips.

Table 1-2 Function differences among chips

Chip	Supported Graphics Layer	Compression	Colorkey	Binding Relationship
Hi3516A/ Hi3518E V200/Hi3519 V100/Hi3519 V101	G0	G0 does not support compression.	Support	G0 is always bound to DSD0

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.

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 DDR detection status.
- 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 (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 Figure 1-2.
- Single-buffer mode (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, jaggies appear. 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

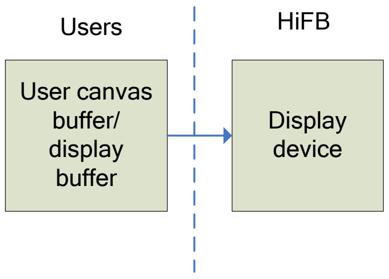


Figure 1-3 Single-buffer mode

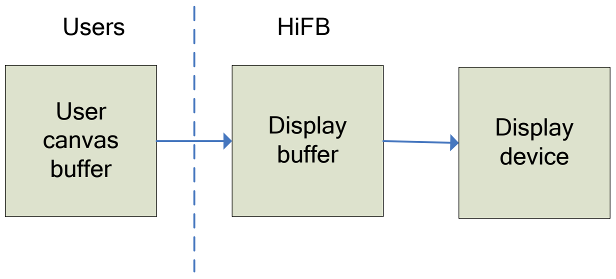
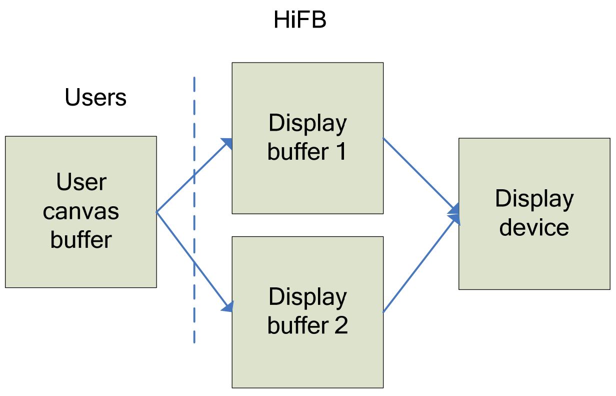


Figure 1-4 Dual-buffer mode



NOTE

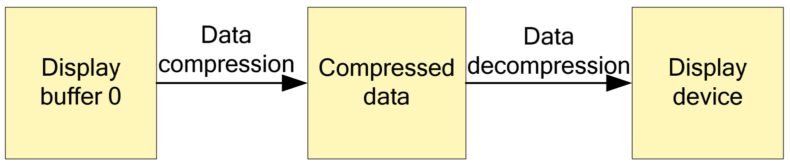
The preceding resolutions are canvas resolution (resolution of user canvas buffers), display buffer resolution, and screen display resolution. When drawn contents are transferred from the user canvas buffer to the display buffer, scaling and anti-flicker are supported. When drawn contents are transferred from the display buffer to the display device, scaling and anti-flicker are not supported. Therefore, the display buffer resolution is always the same as the screen display resolution.

Graphics Layer Compression

Graphics layer compression indicates that graphics layers compress the display buffer to generate compressed data, and then decompress the data for display. When the data in the display buffer does not change, graphics layers load and decompress the compressed data for display. When the compression function of a graphics layer is enabled, the bandwidth loaded by the bus is reduced, but an extra memory for storing a frame of compressed data is required.

Figure 1-5 shows the workflow of a graphics layer compression buffer.

Figure 1-5 Workflow of a graphics layer compression buffer



The compression function is irrelevant to the refresh mode. To be specific, the compression function can be enabled or disabled in both the standard mode of the Linux FB and extended FB refresh mode.

DDR Detection

DDR detection indicates that graphics layers detect whether the data in the display buffer changes. DDR detection takes effect only when the refresh mode is non-buffer mode and the compression function is enabled. When the DDR change is detected, the compression function is enabled to update the compressed data, which prevents refresh operations for display.

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**–**vram3_size** correspond to G0–G3.

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**.

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 four values that correspond to **fb0**–**fb3**. 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.

Default Parameter Values

If the program has no parameter when the HiFB driver is loaded, the default parameter values for the Hi3516A/Hi3518E V200/Hi3519 V100/Hi3519 V101 are as follows:

video="hifb:vram0_size:8100"

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, the maximum virtual resolution is 720 x 576, and the pixel format is 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 G1, the maximum virtual resolution is 720 x 576, and 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.

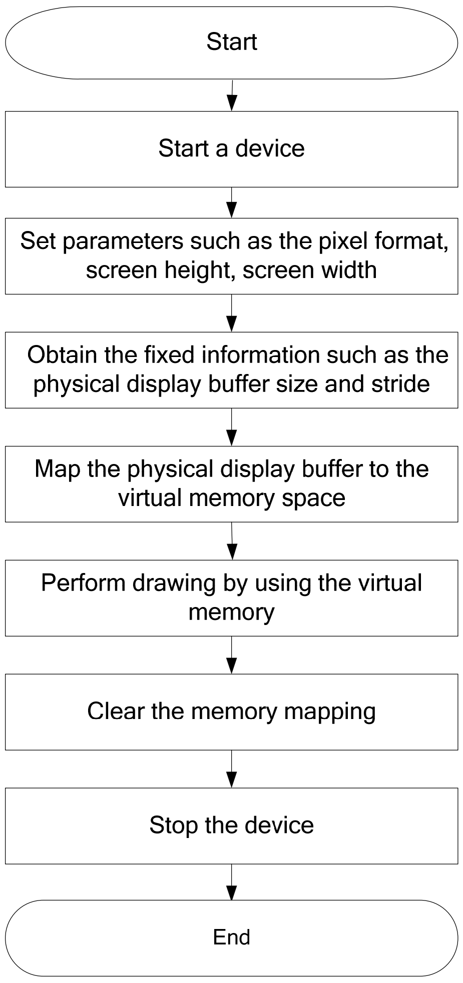
3 Initial Development Application

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


Start
Start a device
Set parameters such as the pixel format, screen height, screen width
Obtain the fixed information such as the physical display buffer size and stride
Map the physical display buffer to the virtual memory space
Perform drawing by using the virtual memory
Clear the memory mapping
Stop the device
End

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 display 15 consecutive pictures with the 640 x 352 resolution for the dynamic display effect.

Each file stores only pure ARGB1555 data (picture data without attached information).

```
[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;
VO_PUB_ATTR_S stPubAttr = {0};
   char image_name[128];

    /*0. open VO device 0 */
/* …… initialize the attributes for stPubAttr */
    HI_MPI_VO_SetPubAttr(0, &stPubAttr);
HI_MPI_VO_Enable(0);

   /*1. open Framebuffer device overlay 0*/
   fd = open("/dev/fb0", 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. get 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. get 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 video memory 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 devices*/
    close(fd);
```

```
HI_MPI_VO_Disable(0);

    return 0;
}
```