



GPU

开发用户指南

文档版本 00B03
发布日期 2018-01-15

版权所有 © 深圳市海思半导体有限公司 2017-2018。保留一切权利。

非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。

商标声明



HISILICON、海思和其他海思商标均为深圳市海思半导体有限公司的商标。

本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受海思公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，海思公司对本文档内容不做任何明示或默示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

深圳市海思半导体有限公司

地址：深圳市龙岗区坂田华为基地华为电气生产中心 邮编：518129

网址：<http://www.hisilicon.com>

客户服务电话：+86-755-28788858

客户服务传真：+86-755-28357515

客户服务邮箱：support@hisilicon.com



前 言

概述

本文档主要介绍 GPU 模块基本功能与开发指引。

产品版本

与本文档相对应的产品版本如下。

产品名称	产品版本
Hi3559A	V100ES
Hi3559A	V100
Hi3559C	V100



说明

未有特殊说明，Hi3559CV100 与 Hi3559AV100 内容一致。

读者对象






本文档（本指南）主要适用于以下工程师：

- 技术支持工程师
- 软件开发工程师

符号约定

在本文中可能出现下列标志，它们所代表的含义如下。



符号	说明
 危险	表示有高度潜在危险，如果不能避免，会导致人员死亡或严重伤害。
 警告	表示有中度或低度潜在危险，如果不能避免，可能导致人员轻微或中等伤害。
 注意	表示有潜在风险，如果忽视这些文本，可能导致设备损坏、数据丢失、设备性能降低或不可预知的结果。
 窍门	表示能帮助您解决某个问题或节省您的时间。
 说明	表示是正文的附加信息，是对正文的强调和补充。

修订记录

修订记录累积了每次文档更新的说明。最新版本的文档包含以前所有文档版本的更新内容。

文档版本 00B03 (2018-01-15)

第 3 次临时版本发布。

添加 Hi3559AV100 和 Hi3559CV100 的相关内容

1.1、2.3、2.5 和 2.6 小节涉及修改，新增 2.1 小节

文档版本 00B02 (2017-05-27)

第 2 次临时版本发布。

删除 Vulkan 的相关内容

文档版本 00B01 (2017-03-10)

第 1 次临时版本发布。



目 录

前 言.....	i
1 概述.....	1
1.1 GPU 简介.....	1
1.1.1 硬件架构	1
1.1.2 软件架构	1
1.2 重要概念.....	2
1.2.1 Open GLES.....	2
1.2.2 OpenCL.....	2
1.2.3 EGL.....	2
1.3 功能描述.....	3
1.3.1 功能特点	3
1.3.2 模块原理	4
2 开发指引.....	7
2.1 驱动编译与加载.....	7
2.2 使用 hi_dbe 驱动.....	8
2.2.1 场景说明	8
2.2.2 工作流程	8
2.2.3 注意事项	9
2.3 WindowSurface EGL Sample 开发.....	9
2.3.1 场景说明	9
2.3.2 工作流程	9
2.3.3 注意事项	10
2.4 Pixmap Surface EGL Sample 开发.....	11
2.4.1 场景说明	11
2.4.2 工作流程	11
2.4.3 注意事项	12
2.5 OpenGL ES 开发	12
2.6 OpenCL 开发	13



插图目录

图 1-1 GPU 标准关系图	1
-----------------------	---



表格目录

表 1-1 Display、Window、Pixmap 三种概念在 FBDEV 窗口系统中的含义.....	4
表 1-2 结构体 <code>egl_linux_pixmap</code> 成员含义.....	5



1 概述

1.1 GPU 简介

GPU（Graphics Process Unit）是 3D 图形处理单元，海思提供完全符合业界标准的 EGL 1.4，OpenGL ES 1.1/2.0/3.0/3.1/3.2 及 OpenCL2.0 标准接口。接口具体的标准规范可参考这个链接：

<http://www.khronos.org/>

本文简单介绍 GPU 基本的软硬件架构、EGL 本地窗口相关的结构体以及 EGL 具体开发流程，还有一些调试手段。

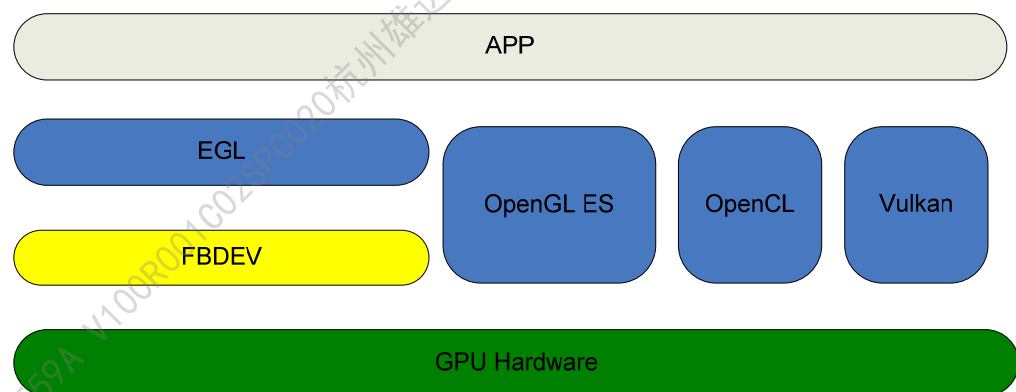
1.1.1 硬件架构

Hi3559AV100ES/Hi3559AV100 使用的 GPU 使用 ARM Bifrost 硬件架构的 Mali G71 GPU。

1.1.2 软件架构

GPU 相关标准在系统中的位置如图 1-1 所示。

图1-1 GPU 标准关系图





海思 EGL 目前支持与 FBDEV 窗口系统对接，用户可以将通过 OpenGL ES 渲染的 3D 内容送到 FBDEV 窗口系统对应的 FrameBuffer 或者 surface 上去显示。

1.2 重要概念

1.2.1 Open GLES

OpenGL ES (Embedded System) 是嵌入式领域的 3D 图形库，它是 khronos 组织基于桌面 OpenGL 标准裁减而来的：

- OpenGL ES 1.1：固定渲染管道，主要服务于低端的嵌入式设备上。
- OpenGL ES 2.0：可编程的渲染管道，主要服务于高端嵌入式设备上。
- OpenGL ES 3.0：向前兼容 OpenGL ES 2.0 的超集，具有更强大的着色功能，支持更多纹理格式。
- OpenGL ES 3.1：在 OpenGL ES 3.0 基础上添加了用于通用计算任务的运算着色器、独立着色顶点与片段的着色对象、间接从内存获取绘图命令等新特性。
- OpenGL ES 3.2：在 OpenGL ES 3.1 基础上增强了对 Android 的支持（集成 AEP 部分内容），增加了延迟渲染、基于物理着色、HDR 色调映射等功能。

ES 3.X 兼容 ES 2.0，基于 ES 3.X 开发的程序可以在 2.0 上运行，但是 ES2.0 和 ES1.1 是不兼容的。

1.2.2 OpenCL

OpenCL (Open Computing Language) 是异构计算的编程接口，它支持将运算任务针对性地分发到不同运算单元上，通过并行的方式提升运算效率，驱动中主要分发至 GPU 上进行运算处理。

1.2.3 EGL

Embedded-System Graphics Library (native platform graphics interface) 是介于 OpenGL ES 等绘制接口与底层窗口系统之间的接口。

OpenGL ES 实际上是一个图形渲染管道的状态机，而 EGL 则是负责维护这些状态以及把渲染的结果送到对应的窗口或者 surface 上去。

【Window Surface】

Window surface 是 EGL 中可以在屏幕中显示的 surface，通过 EGLNativeWindowType 来创建。

【Pixmap Surface】

Pixmap surface 是 EGL 中离屏的 surface，通过 EGLNativePixmapType 封装而来。

【Pbuffer Surface】

Pbuffer 是 EGL 中与平台无关的离屏 surface，主要用于渲染成 texture，或者用于绘制比 window 更大的 surface。

【EGLNativeDisplayType】



EGLNativeDisplayType 是对本地显示设备的抽象，EGL 需要参考本地显示设备创建对应的 EGLDisplay。

【EGLNativeWindowType】

EGLNativeWindowType 是对本地窗口的抽象，EGL 需要参考本地窗口创建对应的 Window Surface，用于屏上显示。

【EGLNativePixmapType】

EGLNativePixmapType 是对本地离屏 buffer 的抽象，EGL 需要参考本地离屏 buffer 创建对应的 Pixmap Surface，用于离屏绘制。



说明

本文档只介绍 EGL 的使用，Open GLES, OpenCL 属于标准绘制接口，请参考 khronos 组织与 ARM 提供的开发指导。

1.3 功能描述

1.3.1 功能特点

EGL 模块主要提供了以下功能：

- 初始化、去初始化功能
 - eglInitialize: 初始化 EGL。
 - eglTerminate: 终止 EGL。
 - eglGetDisplay: 获取显示设备。
- 配置管理功能
 - eglGetConfigs: 获取 display 支持的 framebuffer 配置列表。
 - eglGetConfigAttrib: 获取 display framebuffer 配置中的相关信息。
 - eglChooseConfig: 选取一个符合指定配置的 framebuffer 配置列表。
- 绘制 Surface 管理
 - eglCreateWindowSurface: 创建一个 window surface。
 - eglCreatePbufferSurface: 创建一个离屏的 pbuffer surface。
 - eglCreatePixmapSurface: 创建一个离屏的 pixmap surface。
 - eglDestroySurface: 销毁一个 EGL surface。
 - eglQuerySurface: 查询 surface 属性信息。
- 绘制上下文管理
 - eglCreateContext: 创建一个 context。
 - eglDestroyContext: 销毁一个 context。
 - eglMakeCurrent: 绑定当前 context 与 surface。
 - eglGetCurrentContext: 获取当前 context。
 - eglGetCurrentSurface: 获取当前 read 或者 write surface。



- eglGetCurrentDisplay: 获取当前 display。
- eglQueryContext: 查询 context 相关信息。
- buffer 投递显示
 - eglSwapBuffers: 将 surface 内容送到本地窗口上显示。
 - eglCopyBuffers: 拷贝 surface 内容到本地 pixmap 上。
- EGL 扩展功能
 - eglCreateImageKHR: 创建 EGL image。
 - eglDestroyImageKHR: 销毁 EGL image。

1.3.2 模块原理

Display、Window、Pixmap 是 EGL 编程基本的结构体，1.2.3 EGL 章节中已经做了简介，表 1-1 和表 1-2 对这些基本结构体成员做简单的介绍。

表1-1 Display、Window、Pixmap 三种概念在 FBDEV 窗口系统中的含义

对象	FBDEV	备注
EGLNativeDisplayType	创建到/dev/fb0	默认使用/dev/fb0 作为显示设备
EGLNativeWindowType	<pre>typedef struct fbdev_window { unsigned short width; unsigned short height; } fbdev_window;</pre>	width: 窗口宽度 height: 窗口高度
EGLNativePixmapType	<pre>typedef struct egl_linux_pixmap { int width, height; struct { khronos_usize_t stride; khronos_usize_t size; khronos_usize_t offset; } planes[3]; uint64_t pixmap_format; mem_handle handles[3]; } egl_linux_pixmap;</pre>	请见表 1-2。

表1-2 结构体 `egl_linux_pixmap` 成员含义

数据成员	含义说明
<code>width</code>	Surface 宽度
<code>height</code>	Surface 高度
<code>planes[3]</code>	<p>Pixmap 内存排布:</p> <ul style="list-style-type: none"> 对于 ARGB 颜色格式, 只有一个 plane, 所以 <code>planes[0]</code>有效, <code>handles[0]</code>描述了内存对象 对于 YUV semi-planar 格式, 有 2 个 plane, <code>planes[0]</code>描述了 Y 分量内存排布, <code>handles[0]</code>是其内存对象指针; <code>planes[1]</code>描述了 UV 分量内存排布, <code>handles[1]</code>是其内存对象指针 对于 YUV planar 格式, 有 3 个 plane, <code>planes[0]</code>描述了 Y 分量内存排布, <code>handles[0]</code>是其内存对象指针; <code>planes[1]</code>描述了 U 分量内存排布, <code>handles[1]</code>是其内存对象指针; <code>planes[2]</code>描述了 V 分量内存排布, <code>handles[2]</code>是其内存对象指针 对于每个 plane: <ul style="list-style-type: none"> ➢ <code>stride</code> 是每个分量行间距 ➢ <code>size</code> 是分量内存大小 ➢ <code>offset</code> 是分量在内存中偏移量
<code>pixmap_format</code>	<p>支持如下像素格式:</p> <ul style="list-style-type: none"> 16bit ARGB 像素格式 <ul style="list-style-type: none"> ➢ <code>EGL_COLOR_BUFFER_FORMAT_BGR565</code> ➢ <code>EGL_COLOR_BUFFER_FORMAT_RGB565</code> ➢ <code>EGL_COLOR_BUFFER_FORMAT_ABGR4444</code> ➢ <code>EGL_COLOR_BUFFER_FORMAT_ARGB4444</code> ➢ <code>EGL_COLOR_BUFFER_FORMAT_BGRA4444</code> ➢ <code>EGL_COLOR_BUFFER_FORMAT_RGBA4444</code> ➢ <code>EGL_COLOR_BUFFER_FORMAT_ABGR1555</code> ➢ <code>EGL_COLOR_BUFFER_FORMAT_ARGB1555</code> ➢ <code>EGL_COLOR_BUFFER_FORMAT_BGRA5551</code> ➢ <code>EGL_COLOR_BUFFER_FORMAT_RGBA5551</code> 24bit ARGB 像素格式 <ul style="list-style-type: none"> ➢ <code>EGL_COLOR_BUFFER_FORMAT_BGR888</code> ➢ <code>EGL_COLOR_BUFFER_FORMAT_RGB888</code> 32bit ARGB 像素格式 <ul style="list-style-type: none"> ➢ <code>EGL_COLOR_BUFFER_FORMAT_ABGR8888</code> ➢ <code>EGL_COLOR_BUFFER_FORMAT_sABGR8888</code> ➢ <code>EGL_COLOR_BUFFER_FORMAT_ARGB8888</code> ➢ <code>EGL_COLOR_BUFFER_FORMAT_BGRA8888</code> ➢ <code>EGL_COLOR_BUFFER_FORMAT_RGBA8888</code>



数据成员	含义说明
	<ul style="list-style-type: none">➢ EGL_COLOR_BUFFER_FORMAT_XBGR8888➢ EGL_COLOR_BUFFER_FORMAT_sXBGR8888➢ EGL_COLOR_BUFFER_FORMAT_XRGB8888➢ EGL_COLOR_BUFFER_FORMAT_BGRX8888➢ EGL_COLOR_BUFFER_FORMAT_RGBX8888• 8bit 亮度分量<ul style="list-style-type: none">➢ EGL_COLOR_BUFFER_FORMAT_L8• YV12 格式，CSC 转换时支持 BT601 或者 BT709，NARROW 或者 WIDE 模式<ul style="list-style-type: none">➢ EGL_COLOR_BUFFER_FORMAT_YV12_BT601_NARROW➢ EGL_COLOR_BUFFER_FORMAT_YV12_BT601_WIDE➢ EGL_COLOR_BUFFER_FORMAT_YV12_BT709_NARROW➢ EGL_COLOR_BUFFER_FORMAT_YV12_BT709_WIDE• NV12 格式，CSC 转换时支持 BT601 或者 BT709，NARROW 或者 WIDE 模式<ul style="list-style-type: none">➢ EGL_COLOR_BUFFER_FORMAT_NV12_BT601_NARROW➢ EGL_COLOR_BUFFER_FORMAT_NV12_BT601_WIDE➢ EGL_COLOR_BUFFER_FORMAT_NV12_BT709_NARROW➢ EGL_COLOR_BUFFER_FORMAT_NV12_BT709_WIDE• YUYV 格式，CSC 转换时支持 BT601 或者 BT709，NARROW 或者 WIDE 模式<ul style="list-style-type: none">➢ EGL_COLOR_BUFFER_FORMAT_YUYV_BT601_NARROW➢ EGL_COLOR_BUFFER_FORMAT_YUYV_BT601_WIDE➢ EGL_COLOR_BUFFER_FORMAT_YUYV_BT709_NARROW➢ EGL_COLOR_BUFFER_FORMAT_YUYV_BT709_WIDE• NV21 格式，CSC 转换时支持 BT601 或者 BT709，NARROW 或者 WIDE 模式<ul style="list-style-type: none">➢ EGL_COLOR_BUFFER_FORMAT_NV21_BT601_NARROW➢ EGL_COLOR_BUFFER_FORMAT_NV21_BT601_WIDE➢ EGL_COLOR_BUFFER_FORMAT_NV21_BT709_NARROW➢ EGL_COLOR_BUFFER_FORMAT_NV21_BT709_WIDE
handles[3]	Pixmap 内存句柄



2 开发指引

GPU 模块的几个常用应用场景如下：

- 加载 GPU 驱动
- 配置 hi_dbe 驱动
- Bifrost EGL 基于 FBDEV 平台的开发
- OpenGL ES 开发
- OpenCL 开发
- 3D UI 引擎开发

2.1 驱动编译与加载

GPU 代码路径：SDK 目录下 mpp/component/gpu 文件夹。

GPU 驱动分内核态驱动与用户态驱动两部分，其中内核态驱动提供源码，需用户自行编译 .ko 文件并加载；用户态驱动提供编译完成的 .so 库文件，不需另外编译。

步骤 1. 编译内核态库文件

进入 SDK 目录 mpp/component/gpu/kernel 文件夹下，使用 make 命令编译。

步骤 2. 加载驱动依赖项

GPU 驱动依赖以下驱动，需预先加载：

- 加载 hi_osal.ko
- 加载 hifb.ko

步骤 3. 加载内核态 ko

内核态 ko 包含两个文件：

hi_dbe.ko	将指定物理地址内存封装成 dma_buf，供 GPU 直接访问
mali_kbase.ko	实现 GPU 内核态控制

进入 SDK 目录下 mpp/component/gpu/release/ko 文件夹

- insmod hi_dbe.ko
- insmod mali_kbase.ko

步骤 4. 加载用户态库文件



发布包中包含了编译完成的动态库文件：

libEGL.so	EGL 1.4 标准程序入口
libmali.so	DDK 用户态功能程序入口
libGLESv1_CM.so	OpenGL ES 1.x 标准程序入口
libGLESv2.so	OpenGL ES 2.0/3.x 标准程序入口
libOpenCL.so	OpenCL 1.2/2.0 标准程序入口

用户只需将库文件所在路径添加到 LD_LIBRARY_PATH 下即可。

- 用户执行 OpenGL ES 1.x 标准程序，需加载 libEGL.so、libmali.so、libGLESv1_CM.so 三个库文件；
- 用户执行 OpenGL ES 2.0/3.x 标准程序，需加载 libEGL.so、libmali.so、libGLESv2.so 三个库文件。
- 用户执行 OpenCL 1.2/2.0 标准程序，需加载 libOpenCL.so 和 libmali.so 两个库文件。

----结束

2.2 使用 hi_dbe 驱动

2.2.1 场景说明

由海思开发的 hi_dbe 驱动，用于将带物理首地址的物理内存 wrap 成 dma_buf 的 fd，EGL 可抽象这个 fd 成为 Pixmap Surface，GPU 绘制后的 Pixmap Surface 内容将同步到用户提供的物理内存上，中间不存在拷贝。



说明

hi_dbe 驱动要求使用的内存物理首地址满足 4K 对齐。

2.2.2 工作流程

配置 hi_dbe 驱动步骤如下：

步骤 1. 加载 hi_dbe.ko

步骤 2. 打开 hi_dbe 设备，调用 hi_dbe 接口，将内存抽象成 dma_buf 的 fd。

```
int HI_COMMON_LOCAL_WrapDmaBufFD(unsigned int physical_address, unsigned
int size, int flag)
{
    struct hidbe_ioctl_wrap phywrap;
    int dmabuf_fd;
    int device_fd = open("/dev/hi_dbe", O_RDWR);    #打开hi_dbe设备
```




```
phywrap.dbe_phyaddr = physical_address;          #配置内存地址
phywrap.dbe_size = size;                          #配置内存大小
dmabuf_fd = ioctl(device_fd, DBE_COMMAND_WRAP_ADDRESS, &phywrap); #调用命令，抽象内存
close(device_fd);                                #关闭hi_dbe设备
return dmabuf_fd ;                               #返回wrap的dma buf fd
}
```

----结束

2.2.3 注意事项

无。

2.3 WindowSurface EGL Sample 开发

2.3.1 场景说明

用 EGL 创建 Window Surface，GPU 绘制后的 Window Surface 的内容可直接显示在屏幕上。

2.3.2 工作流程

用 EGL 创建 Window Surface 的过程如下：

- 步骤 1. 选取本地显示设备，0 对应/dev/fb0fb 支持的设备号请参考 fb 驱动相关说明。本 sample 中选取 EGL_DEFAULT_DISPLAY。

```
int nativeDisplay = 0;
```

- 步骤 2. 创建一个本地窗口，假定分辨率为 1280 * 720。

```
fbdev_window * nativeWindow = malloc( sizeof( fbdev_window ) );
if ( NULL == fbwin )
{
    return 0;
}
nativeWindow->width = 1280;
nativeWindow->height = 720;
```

- 步骤 3. 根据本地显示设备句柄获取 EGL 显示对应的句柄。

```
EGLDisplay eglDisplay = eglGetDisplay(nativeDisplay);
```

- 步骤 4. 初始化 EGL。

```
eglInitialize(eglDisplay, NULL, NULL);
```




步骤 5. 选择对应的配置。

```
eglChooseConfig(eglDisplay, configAttribs, &configs[0], 10,  
&matchingConfigs);
```

步骤 6. 创建 EGL Surface。

```
EGLSurface eglSurface = eglCreateWindowSurface(eglDisplay, configs[0],  
nativeWindow, configAttribs);
```

步骤 7. 创建 EGL context。

```
EGLContext eglContext = eglCreateContext(eglDisplay, configs[0], NULL,  
ctxAttribs);
```

步骤 8. 设置当前上下文。

```
eglMakeCurrent(eglDisplay, eglSurface, eglSurface, eglContext);
```

至此 EGL 的整个初始化过程已经完成了，用户可以开始自己的绘制动作了。

----结束

EGL 销毁的过程如下：

步骤 1. 当前上下文置 NULL。

```
eglMakeCurrent(eglDisplay, NULL, NULL, NULL);
```

步骤 2. 销毁 context

```
eglDestroyContext(eglDisplay, eglContext);
```

步骤 3. 销毁 eglSurface

```
eglDestroySurface(eglDisplay, eglSurface);
```

步骤 4. 终止 EGL

```
eglTerminate(g_EglDisplay);
```

步骤 5. 销毁本地窗口

```
free(nativeWindow);
```

----结束

2.3.3 注意事项

绘制结束后用户需要调用 `eglSwapBuffers` 把绘制的内存刷新到屏幕上去。上述过程中只有创建过程的步骤 2 和步骤 3 是和本地窗口系统相关的，其他接口都是标准的 EGL 接口，用户可以从 khronos 组织的官方网站上找到对应的接口说明。



2.4 Pixmap Surface EGL Sample 开发

2.4.1 场景说明

Pixmap 的物理内存由用户自己分配，通过 hi_dbe 驱动将内存封装成 fd，并调用 EGL 使用 fd 封装成 Pixmap Surface，GPU 可绘制这块内存。

2.4.2 工作流程

用 EGL 创建 Pixmap Surface 的过程如下：

步骤 1. 选取本地显示设备。

0 对应/dev/fb0

fb 支持的设备号请参考 fb 驱动相关说明。本 sample 中选取/dev/fb0。

```
int nativeDisplay = 0;
```

步骤 2. 创建一个本地离屏内存，假定分辨率为 1280*720，格式为 RGBA8888。

```
struct egl_linux_pixmap * nativePixmap =  
(struct egl_linux_pixmap*)malloc(sizeof(struct egl_linux_pixmap));  
memset(nativePixmap, 0, sizeof(*nativePixmap));  
  
nativePixmap->planes[0].stride = 1280 * 4;  
nativePixmap->planes[0].size = nativePixmap->planes[0].stride * 720;  
nativePixmap->planes[0].offset = 0;  
nativePixmap->pixmap_format = EGL_PIXMAP_FORMAT_ARGB8888;
```

需要加载 hi_dbe.ko 将内存 wrap 成 dma buf fd，具体可参考“2.2 使用 hi_dbe 驱动”。

```
nativePixmap->handles[0].fd = HI_COMMON_LOCAL_WrapDmaBufFD(buffer_phyaddr,  
buffer_size, 0);  
return (NativePixmapType)egl_create_pixmap_ID_mapping(nativePixmap);
```

步骤 3. 根据本地显示设备句柄获取 EGL 显示对应的句柄。

```
EGLDisplay eglDisplay = eglGetDisplay(nativeDisplay);
```

步骤 4. 初始化 EGL。

```
eglInitialize(eglDisplay, NULL, NULL);
```

步骤 5. 选择对应的配置。

```
eglChooseConfig(eglDisplay, configAttribs, &configs[0], 10,  
&matchingConfigs);
```

步骤 6. 创建 EGL Surface。

```
EGLSurface eglSurface = eglCreatePixmapSurface(eglDisplay, configs[0],  
nativePixmap, configAttribs);
```



步骤 7. 创建 EGL context。

```
EGLContext eglContext = eglCreateContext(eglDisplay, configs[0], NULL,  
ctxAttribs);
```

步骤 8. 设置当前上下文。

```
eglMakeCurrent(eglDisplay, eglSurface, eglSurface, eglContext);
```

----结束

EGL 销毁的过程如下：

步骤 1. 当前上下文置 NULL。

```
eglMakeCurrent(eglDisplay, NULL, NULL, NULL);
```

步骤 2. 销毁 context

```
eglDestroyContext(eglDisplay, eglContext);
```

步骤 3. 销毁 eglSurface

```
eglDestroySurface(eglDisplay, eglSurface);
```

步骤 4. 终止 EGL

```
eglTerminate(g_EglDisplay);
```

步骤 5. 销毁本地离屏 pixmap

```
struct egl_linux_pixmap *pixmap_dma = (struct egl_linux_pixmap  
(*)egl_lookup_pixmap_ID_mapping((int)nativePixmap);  
egl_destroy_pixmap_ID_mapping((int)nativePixmap);  
  
close(pixmap_dma->handles[0].fd);  
free(pixmap_dma);
```

----结束

2.4.3 注意事项

无。

2.5 OpenGL ES 开发

OpenGL ES 开发过程以及优化，请参考 ARM 提供的指导文档路径：

<https://developer.arm.com/graphics/tutorials/android-opengl-es-3-0-and-3-1-tutorials>

在文档中详细介绍了利用 OpenGL ES 进行开发工作的流程。



2.6 OpenCL 开发

OpenCL 开发流程以及优化，请参考 ARM 提供的 OpenCL 开发指导路径：

<https://developer.arm.com/graphics/tutorials/opencl-tutorials>

在文档中详细介绍了利用 OpenCL 进行开发工作的流程。