

DAYU 平台OpenHarmony 适配指导

2021 年8 月



目录

1 添加 DAYU product.....	1
2 编译.....	2
3 启动配置.....	3
4 TP 驱动.....	3
4.1 配置	4
4.2 配置对应的私有配置信息.....	5
4.3 驱动文件编写.....	7
4.4 所遇关键问题总结	8
5 显示驱动.....	10
5.1 display 驱动框架的.....	10
5.2 兼容 DRM 架构（显示架构适配层）	12
5.3 移植 panel 驱动的指导（以 ili9881c_boe.c 为例）	14
6 图形显示框架	18
7 HDC	19
8 版本烧录.....	20

1 添加 DAYU product

1. 在 productdefine/common/products 目录下创建 DAYU.json 产品 json 文件，如图：

```
wangshaofeng@wangshaofeng:~/extdisk/new_L2/productdefine/common/products$  
DAYU.json Hi3516DV300.json  
wangshaofeng@wangshaofeng:~/extdisk/new_L2/productdefine/common/products$
```

备注：DAYU.json 可以参考 Hi3516DV300 的内容进行创建，注意删除

"hisilicon_products:hisilicon_products":{}，其他部件根据产品情况自行删减

2. 在 productdefine/common/device 目录下创建 dayu.json 文件，如图

```
wangshaofeng@wangshaofeng:~/extdisk/new_L2/productdefine/common/device$ ls  
1  
{  
    "target_os": "ohos",  
    "target_cpu": "arm"  
}
```

根据实际情况，这里的 target_cpu 也可能是 arm64、riscv、x86 等。当前仅支持 arm 作为 target_cpu。

3. 在 build 目录下修改 subsystem_config.json，增加对编译的支持

```
"hihope_products": {  
    "project": "hmf/hihope_products",  
    "path": "device/hihope/build",  
    "name": "hihope_products",  
    "dir": "device/hihope"  
},
```

4. 新增 dayu 的编译三个仓

vendor/hihope ,device/hihope/build,device/hihope/dayu 这三个目录新增需要新建单独的 git 仓库 .repo\manifests\default.xml

```
<project name="vendor_hihope" path="vendor/hihope"
```

```
<project name="device_hihope_build" path="device/hihope/build"
```

```
<project name="device_hihope_dayu" path="device/hihope/dayu"
```

vendor/hihope : 主要为 hihope product 的产品特有相关配置

device/hihope/build: 主要为 hihope products 编译相关的路径配置

device/hihope/dayu: 主要为 dayu 相关的 rc 文件及 lib 库等文件

备注: 这个三个目录可以参考 Hi3516DV300 目录结构进行放置和编译

2 编译

1. 在 Linux 环境进行如下操作:

进入源码根目录, 执行如下命令进行版本编译。

```
./build.sh --product-name DAYU
```

检查编译结果。编译完成后, log 中显示如下:

```
build system image successful.
```

```
====build DAYU successful.
```

编译所生成的文件都归档在 out/ohos-arm-release/目录下,

结果镜像输出在 out/ohos-arm-release/packages/phone/images/ 目录下。

3 启动配置

1. 由于内核是 64 位，system 是 32 位，为了兼容 32 位需要修改 binder
2. 关闭 avb 等其他和平台强相关的安全配置，这个依各个平台情况所定
3. 关闭动态分区
4. data 分区挂载

device/hihope/dayu/build/vendor/etc 下新增：fstab.dayu 用于挂载 data 分区

并在 init.dayu.rc 中加入：

```
on fs
    mount_all /vendor/etc/fstab.dayu
    swapon_all /vendor/etc/fstab.dayu
```

5. rc 文件

rc 文件主要在 vendor 分区中，主要涉及

init.dayu.rc

init.dayu.usb.rc

fstab.dayu

init.dayu.common.rc

init.dayu_flash.rc

这些 rc 用于启动和配置平台私有的一些服务和配置

4 TP 驱动

HDF input 驱动模型包括内核驱动与用户态驱动两部分。用户态驱动提供了标准的鸿

蒙南向 HDI 接口以及实现代码，无需开发者改动即可使用，具体用法可参考开源资料相关部分。因此，HDF 触屏驱动移植主要工作为内核驱动移植。

触屏内核驱动依赖 platform 能力，主要包括 GPIO（拉高拉低）、中断（中断注册及处理）、IIC（与器件通信方式）能力；与用户态通信依赖 IOService 能力；驱动程序注册进内核依赖 HDF 框架；在确保上述能力功能正常后，即可开始本次 DAYU 开发板的移植工作。本文档基于已经开源的 hi3516dv300 开源板代码为基础版本，来进行此次移植。

基础环境配置可参考开源资料部分相关章节。关于 HDF_INPUT 模型部分可参考开源资料中相关部分。

DAYU 开发板移植 touch 驱动涉及的文件及目录：

1. Makefile 文件：drivers\adapter\khdf\linux\model\input\Makefile
2. drivers\adapter\khdf\linux\hcs\device_info\device_info.hcs
3. drivers\adapter\khdf\linux\hcs\input\input_config.hcs
4. drivers\framework\model\input\driver\touchscreen

4.1 配置

DAYU 开发板所使用的触摸屏 IC 为 GT9271，分辨率为 800 * 1280，使用 I2C3 与主控通信（通信地址为 0x5d），中断管脚为 272 号管脚，复位管脚为 273 号管脚。HDF 将驱动加载信息以及硬件资源统一配置为 hcs 文件来进行统一管理，原驱动代码对配置文件进行了统一的解析。因此，首先第一步应将相应硬件资源进行 hcs 配置，具体需要配置的文件一共有两处 drivers\adapter\khdf\linux\hcs\device_info\device_info.hcs 和 drivers\adapter\khdf\linux\hcs\input\input_config.hcs。前者供 HDF 框架加载驱动时使用，后者包含了触屏的板级硬件信息和器件信息（接下来统称为私有配置信息）：

1. 首先需要配置 device_info.hcs 文件，配置完成后如下图所示

```
device_hdf_touch :: device {
    device0 :: deviceNode {
        policy = 2;
        priority = 120;
        preload = 1;
        permission = 0660;
        moduleName = "HDF_TOUCH";
        serviceName = "hdf_input_event1";
        deviceMatchAttr = "touch_device1";
    }
    device1 :: deviceNode {
        policy = 2;
        priority = 120;
        preload = 0;
        permission = 0660;
        moduleName = "HDF_TOUCH";
        serviceName = "hdf_input_event1";
        deviceMatchAttr = "touch_device2";
    }
}

device_touch_chip :: device {
    device0 :: deviceNode {
        policy = 0;
        priority = 130;
        preload = 1;
        permission = 0660;
        moduleName = "HDF_TOUCH_GT911";
        serviceName = "hdf_touch_gt911_service";
        deviceMatchAttr = "zsj_gt911_5p5";
    }
    device1 :: deviceNode {
        policy = 0;
        priority = 130;
        preload = 0;
        permission = 0660;
        moduleName = "HDF_TOUCH_GT9271";
        serviceName = "hdf_touch_gt9271_service";
        deviceMatchAttr = "zsj_gt91271";
    }
}
```

图中“device0”为原生 hi3516dv300 的配置，由于原生驱动不需要加载因此相关字段“preload”置为 1。

“device1”为新增 DAYU 配置，需要注意的几个关键字段，device_hdf_touch 中的 moduleName 字段需要与驱动注册入口处的保持一致，deviceMatchAttr 字段需要与私有配置信息中的保持一致。

4.2 配置对应的私有配置信息

在文件 input_config.hcs 文件中修改，原配置文件 touch0 中已包含 hi3516dv300 开

发板与触屏相关的板级硬件信息 (“ boardConfig” 节点) 以及触屏 IC 器件信息 (“ chipConfig” 节点)。在与 touch0 相同次级位置处添加 DAYU 的相关配置即可, 添加节点 touch1。配置完成后如下图所示:

```

touch1 {
    boardConfig {
        match_attr = "touch_device2";
        inputAttr {
            /* 0:touch 1:key 2:keyboard 3:mouse 4:button 5:crown 6:encoder */
            inputType = 0;
            solutionX = 1280;
            solutionY = 800;
            devName = "main_touch";
        }
        busConfig {
            /* 0:i2c 1:spi */
            busType = 0;
            busNum = 3;
            clkGpio = 146;
            dataGpio = 147;
        }
        pinConfig {
            rstGpio = 273;
            intGpio = 272;
        }
    }
    chipConfig {
        template touchChip {
            match_attr = "";
            chipName = "gt911";
            vendorName = "zsj";
            chipInfo = "AAAA11222"; // 4-ProjectName, 2-TP IC, 3-TP Module
            /* 0:i2c 1:spi */
            busType = 0;
            deviceAddr = 0x5D;
            /* 0:None 1:Rising 2:Falling 4:High-level 8:Low-level */
            irqFlag = 2;
            maxSpeed = 400;
            chipVersion = 0; //parse Coord TypeA
            powerSequence {
                powerOnSeq = [4, 0, 1, 0,
                    3, 0, 1, 10,
                    3, 1, 2, 60,
                    4, 2, 0, 0];
                suspendSeq = [3, 0, 2, 10];
                resumeSeq = [3, 1, 2, 10];
                powerOffSeq = [3, 0, 2, 10,
                    1, 0, 2, 20];
            }
        }
        chip0 :: touchChip {
            match_attr = "zsj_gt911_5p5";
            chipInfo = "ZIDN45100"; // 4-ProjectName, 2-TP IC, 3-TP Module
            chipVersion = 0; //parse point by TypeA
        }
    }
}

```

图中我们添加了硬件信息, 如 busconfig 中配置为 IIC 方式, 且配置 IIC 号为 3。中断管脚 intGpio 配置为 272, 复位管脚配置为 273。配置了分辨率相关字段 solutionX 与 solutionY 等。

4.3 驱动文件编写

HDF_Input 驱动模型一共包含三部分：驱动管理层、公共驱动层以及器件驱动层。开发者只需要适配少量器件驱动层代码即可。

首先在 `drivers\framework\model\input\driver\touchscreen` 目录下建立 C 文件以及对应的头文件，这里我们创建文件 `touch_gt9271.c` 和 `touch_gt9271.h`。其中依赖的头文件与 `touch_gt911.c` 保持一致即可。添加驱动注册入口如下：

```
struct HdfDriverEntry g_touchGoodixChipEntry = {  
    .moduleVersion = 1,  
    .moduleName = "HDF_TOUCH_GT9271",    //与 device_info.hcs 中的  
moduleName 字段保持一致  
    .Init = HdfGoodixChipInit,  
};  
HDF_INIT(g_touchGoodixChipEntry);
```

在 `touch_gt9271.c` 文件中，驱动开发者需要重点实现器件检测函数、器件数据解析函数等。

```
static struct TouchChipOps g_gt911ChipOps = {  
    .Init = ChipInit,  
    .Detect = ChipDetect,  
    .Resume = ChipResume,  
    .Suspend = ChipSuspend,
```

```
.DataHandle = ChipDataHandle,

.UpdateFirmware = UpdateFirmware,

};
```

由于DAYU 开发板所使用触屏器件IC 与Hi3516dv300 开发板触屏器件IC 为同一系列产品，因此器件驱动代码部分可直接复用。除驱动注册入口处 moduleName 值不一样以外，其他保持一致即可。

编译文件配置：(drivers\adapter\khdf\linux\model\input\Makefile)

添加如下内容：

```
obj-$(CONFIG_DRIVERS_HDF_TP_5P5_GT911) += \
    $(INPUT_ROOT_DIR)/touchscreen/touch_gt911.o
obj-y += \
    $(INPUT_ROOT_DIR)/touchscreen/touch_gt9271.o
obj-$(CONFIG_DRIVERS_HDF_TP_2P35_FT6236) += \
    $(INPUT_ROOT_DIR)/touchscreen/touch_ft6336.o
```

至此，DAYU 开发板 touch 驱动移植完毕。

4.4 所遇关键问题总结

1、GPIO 及中断相关

问题 1：request_threaded_irq 会导致系统休眠，无法在自旋锁中使用

在处理线程中断注册时，原先方案是通过 linux 的 request_threaded_irq 直接使用 linux

内核的线程中断机制。但是鸿蒙 GPIO 框架，在调用每一个 GpioCntlr 钩子方法之前，都会故，当用户以 GPIO_IRQ_USING_THREAD 标记调用 GpioSetIrq 时，会引起内核告警、注持有 spinlock，之后不允许休眠。

册失败。

解决方法：不使用 linux 内核的中断机制，采用 core 层主动创建中断线程的方式（同 liteos）。

问题 2：在调试 TP 时，GPIO 操作始终无法生效

在 linux 内核下，一个 gpio 管脚的编号不能简单的通过 soc 手册确定。必须通过 of_get_named_gpio 从 dts 节点中读取。

2、内核态数据上报用户态失败

问题：HDF IoService listener 机制无法获取内核态上报事件

IoService listener 通过 ioctl 与内核设备交互，ioctl 设计的用于交互的 HdfWriteReadBuf 数据结构中存在 uintptr_t、size_t 等可移植数据类型。

struct HdfWriteReadBuf

```
{ int cmdCode;

  size_t writeSize;    // bytes to write

  size_t writeConsumed; // bytes consumed by driver (for ERESTARTSYS)

  uintptr_t writeBuffer;

  size_t readSize; // bytes to read

  size_t readConsumed; // bytes consumed by driver (for ERESTARTSYS)

  uintptr_t readBuffer;
};
```

但是 DAYU 使用 64bit 内核+32bit 用户态，导致用户态和内核计算的 HdfWriteReadBuf 大小不一致，在拷贝该对象和访问用户态指针时出现错位。

解决方法：使用确定长度的可移植数据类型，保证用户态和内核态在任何场景下计算得到的 数据长度均一致。

3.内核态数据上报用户态失败

DAYU 开发板上 init 时序有问题，无法保证 udevd 先于 uinput_inject 和 weston 前启动，通过将 udevd 、 uinput_inject、 weston 等的启动放到一个 rc 中可以规避 init 时序问题导致的问题。

5 显示驱动

5.1 display 驱动框架的

基于 HDF 框架之上的，主要如下几层：

1. 显示架构适配层（标准显示系统选用）：用于桥接标准 DRM 现实框架，对应的实现文件是“ hdf_drm_panel.c” ；

2. 显示公共驱动层：用于实现所有 panel 驱动层通用的一些功能，比如 esd 检查，产线检测方案，这样可以简化驱动的编写，同时避免 panel 驱动重复实现一些通用的功能，而且方便统一管理和维护，对应的实现文件是“ hdf_disp.c” ；

3. 芯片平台适配层（轻量级 soc 设备选用）：用于初始化轻量级的 soc 与显示相关的接口，比如 Mipi、PWM 等配置，具体可以参考“ adapter_soc/ hi35xx_disp.c” ；

4. Panel 驱动层：用于具体模组的驱动实现，使用HDF 框架提供的各种接口进行驱动的实现。具体可以参考“ panel/ mipi_icn9700.c” 和“ panel/ili9881c_boe.c” ；

panel 驱动是如何注册

通过 “int32_t RegisterPanel(struct PanelData *data)” 就可以把具体模组的驱动注册到 display 框架中，其中“ struct PanelData ” 是对模组的抽象，包含了所有模组共用

的一些信息以及一些需要驱动实现的函数接口，具体如下：

```
struct PanelData {
    struct HdfDeviceObject *object;
    int32_t (*init)(struct PanelData *panel);
    int32_t (*on)(struct PanelData *panel);
    int32_t (*off)(struct PanelData *panel);
    int32_t (*setBacklight)(struct PanelData *panel, uint32_t level);
    struct PanelInfo *info;
    struct PanelStatus status;
    struct PanelEsd *esd;
    void *priv;
};
```

object: 用于保存 HDF 框架中的“ struct HdfDeviceObject”，主要用 object 的 priv 成员保存描述具体驱动信息的结构体指针，然后在具体驱动实现的方法中找到对应的结构体指针，避免驱动中各种全局变量的问题；

init: 用于对具体模组驱动的初始化，一般来说，具体的驱动中“ struct HdfDriverEntry”中的“ Init”用于对驱动的初始化，类似于 Linux 驱动模型中的“ probe”函数，此处的“ init”主要用于一些特殊平台需要先初始化的情况，比如“ mipi_icn9700”驱动用于 Hi35xx 平台时，一些需要 Hi35xx 先进行初始化，然后“ mipi_icn9700”才能初始化，此时把这些资源的初始化就放在此处的“ init”函数实现中，具体可以参考“ mipi_icn9700.c”中的实现；

on: 用于实现亮屏的操作，包括上电以及下发屏幕的初始化序列等；

off: 用于实现灭屏的操作，包含下发灭屏 Code 以及下电的操作等；

setBacklight: 用于设置模组的背光；

info: 用于保存模组的一些信息，比如分辨率、porch、时钟等；

status: 用于记录模组电源状态以及当前的背光值；

esd: 用于实现模组 esd 检测；

Priv: 用于保存一些在“ hdf_disp”层需要的信息。

5.2 兼容 DRM 架构（显示架构适配层）

对于一些处理功能强大的芯片，比如海思、高通、联发科以及展锐手机芯片，在 Linux 中相关的 display 驱动都是基于 DRM 框架的，所以我们专门对 DRM 框架进行了兼容，“drivers/framework/model/display/driver”下的“hdf_drm_panel.c”用于兼容 DRM 框架。它的实现方式类似于一个驱动的实现，首先注册到 HDF 框架中。

注册部分的代码如下：

```
int32_t HdfDrmPanelEntryInit(struct HdfDeviceObject *object)
{
    uint32_t i;
    uint32_t j;
    uint32_t ret;
    uint32_t panelNum;
    struct HdfDrmPanel *hdfDrmPanel = NULL;
    struct mipi_dsi_device *dsiDev = NULL;
    struct DispManager *manager = NULL;

    manager = GetDispManager();
    if (manager == NULL) {
        HDF_LOGE("%s manager is null", _func_);
        return HDF_FAILURE;
    }
    panelNum = manager->panelManager->panelNum;
    for (i = 0; i < panelNum; i++) {
        hdfDrmPanel = (struct HdfDrmPanel *)OsalMemCalloc(sizeof(struct
HdfDrmPanel));
        if (hdfDrmPanel == NULL) {
            HDF_LOGE("%s hdfDrmPanel malloc fail", _func_);
            return HDF_FAILURE;
        }
        ...
    }
    HDF_LOGI("%s success", _func_);
    return HDF_SUCCESS;
}
```

```
struct HdfDriverEntry g_hdfDrmPanelEntry = {
    .moduleVersion = 1,
    .moduleName = "HDF_DRMPANEL",
    .Init = HdfDrmPanelEntryInit,
};

HDF_INIT(g_hdfDrmPanelEntry);
```

然后，在"drivers/adapter/khdf/linux/hcs/device_info/device_info.hcs"文件进行具体的配置，注意：hcs 中配置的"moduleName"一定要与实现文件中的"moduleName"一致，因为 HDF 框架就是比较"moduleName"是来决定是否运行对应的".Init"，配置部分如下：

```
display :: host {
    hostName = "display host";
    device_hdf_drm_panel :: device {
        device0 :: deviceNode {
            policy = 0;
            priority = 197;
            preload = 0;
            moduleName = "HDF_DRMPANEL";
        }
    }
    device_hdf_disp :: device {
        device0 :: deviceNode {
            policy = 2;
            priority = 196;
            permission = 0660;
            moduleName = "HDF_DISP";
            serviceName = "hdf_disp";
        }
    }
    device_hi35xx_disp :: device {
        device0 :: deviceNode {
            policy = 0;
            priority = 195;
            moduleName = "HI351XX_DISP";
        }
    }
    device_lcd :: device {
        device0 :: deviceNode {
            policy = 0;
            priority = 100;
            preload = 2;
            moduleName = "LITE_LCDKIT";
            deviceMatchAttr = "hdf_lcdkit_driver";
        }
    }
}
```


5.3 移植 panel 驱动的指导 (以 ili9881c_boe.c 为例)

1. 新增 panel 驱动对应的文件

在“drivers/framework/model/display/driver/panel”目录下新增具体驱动对应的实现文件“ili9881c_boe.c”和“ili9881c_boe.h”。

2. 修改 Makefile 文件

修改“drivers/adapter/khdf/linux/model/display”目录下的 Makefile 文件，主要是把新增的驱动编译进去，具体如下：

```
obj-$(CONFIG_ARCH_SPRD) += \
    $(DISPLAY_ROOT_DIR)/hdf_drm_panel.o \
    $(DISPLAY_ROOT_DIR)/panel/ili9881c_boe.o
```

3. 配置 HCS 文件

修改“drivers/adapter/khdf/linux/hcs/device_info”目录下的 hcs 文件，新增对“ili9881c_boe”的描述，具体如下：

```
display :: host {
    hostName = "display_host";
    device_hdf_drm_panel :: device {
        device0 :: deviceNode {
            policy = 0;
            priority = 197;
            preload = 0;
            moduleName = "HDF_DRMPANEL";
        }
    }
    ...
    device_lcd :: device {
        ...
        device3 :: deviceNode {
            policy = 0;
            priority = 100;
            preload = 0;
            moduleName = "LCD_ILI9881CBOE";
        }
    }
}
```

注意此处的“moduleName”要与“ili9881c_boe.c”文件中的一样，这样驱动才能跑起来。

4. 修改 patch 文件

因为在原生的linux 系统中没有HDF 框架，而且一般来说SOC 厂家也实现了一些panel 的驱动，我们要实现的"ili9881c_boe.c"正好与厂家提供的" panel.c"是相对应的，所以要屏蔽"panel.c" 以及其他文件中调用此文件中实现的一些函数，同时需要打开"CONFIG_DRIVERS_HDF_DISP"宏，使基于 HDF 框架的 display 驱动模型参与编译。

5. 驱动模板

如下是实现具体驱动的一个模板，具体可以参考" ili9881c_boe.c" 和" mipi_icn9700.c" 实现的相关细节。

```
struct xxxDev {
    struct PanelData panel; //PanelData 是所有模组共用的部分
    /* 包含具体模组特有的信息 */
    ...
};

static struct xxxDev *ToxxxDev(struct PanelData *panel)
{
    return (struct xxxDev *)panel->object->priv;
}

static int32_t xxxOn(struct PanelData *panel)
{
    struct xxxDev *xxxDev;
    xxxDev = ToxxxDev(panel);
    ...
    return 0;
}

static int32_t xxxOff(struct PanelData *panel)
{
    struct xxxDev *xxxDev;
    xxxDev = ToxxxDev(panel);
    ...
}
```

```

    return 0;
}

static int32_t xxxInit(struct PanelData *panel)
{
    struct xxxDev *xxxDev;
    xxxDev = ToxxxDev(panel);
    ...
    return 0;
}

static int32_t xxxSetBl(struct PanelData *panel, uint32_t level)
{
    struct xxxDev *xxxDev;
    xxxDev = ToxxxDev(panel);
    ...
    return 0;
}

static struct PanelInfo g_panelInfo = {
    .width = xx,          /* width */
    .height = xx,        /* height */
    .hbp = xx,            /* horizontal back porch */
    .hfp = xx,            /* horizontal front porch */
    .hsw = xx,            /* horizontal sync width */
    .vbp = xx,            /* vertical back porch */
    .vfp = xx,            /* vertical front porch */
    .vsw = xx,            /* vertical sync width */
    .clockFreq = xx,
    .pWidth = xx,
    .pHeight = xx,
};

static void xxxResInit(struct Ili9881cBoeDev *ili9881cBoeDev)
{
    ...
    xxxDev->panel.info = &g_panelInfo;
    xxxDev->panel.init = xxxInit;
}

```

```

xxxDev->panel.on = xxxOn;
xxxDev->panel.off = xxxOff;
xxxDev->panel.setBacklight = xxxSetBl;
}

int32_t xxxEntryInit(struct HdfDeviceObject *object)
{
    struct xxxDev *xxxDev;

    /* 动态分配具体模组对应的结构体，使驱动可以支持多个 device */
    xxxDev = (struct xxxDev *)OsalMemCalloc(sizeof(struct xxxDev));
    if (xxxDev == NULL) {
        HDF_LOGE("%s xxxDev malloc fail", _func_);
        return HDF_FAILURE;
    }
    xxxResInit(xxxDev);
    xxxDev->panel.object = object;
    object->priv = xxxDev; //保存具体摸对应的结构体指针，用于在调用 on、off 接口
    时获取具体

    //模组对应的结构体
    if (RegisterPanel(&xxxDev->panel) == HDF_SUCCESS)
        { return HDF_SUCCESS;
        }

    FAIL:
    OsalMemFree(xxxDev);
    return HDF_FAILURE;
}

struct HdfDriverEntry xxxDevEntry = {
    .moduleVersion = 1,
    .moduleName = "LCD_xxx",
    .Init = xxxEntryInit,
};

HDF_INIT(xxxDevEntry);

```

6 图形显示框架

鸿蒙标准系统采用 Wayland 图形框架，适配过程中主要涉及

1. 增加当前平台的 drm 驱动的 DRIVER_NAME

foundation/graphic/standard/interfaces/innerkits/vsync/vsync_type.h

2. 驱动 drm 驱动需要 weston 调用的 ioctl 接口

主要一下几个 ioctl:

gem_prime_export
gem_prime_get_sg_table
gem_prime_vmap
gem_prime_vunmap
gem_prime_mmap

3. Weston drm 修改

```
libweston/backend-drm/drm.c
@@ -821,7 +821,7 @@ drm_plane_create(struct drm_backend *b, const drmModePlan
21 821 }
22 822 else {
23 823     plane->possible_crtcs = (1 << output->pipe);
24 - plane->plane_id = 0;
24 + plane->plane_id = 1;
25 825     plane->count_formats = 1;
26 826     plane->formats[0].format = format;
27 827     plane->type = type;
```

4. hisi tde 相关的接口

```
libweston/tde-render-part.c
@@ -344,16 +344,19 @@ int tde_render_attach_hook(struct weston_surface *es, struct weston_buffer *buff
344 344 ps->tde->image.fd[0] = fd;
345 345
346 346 uint32_t* ptr = (uint32_t*)mmap(NULL, stride * buffer->height, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
347 - ps->tde->image.phyaddr = drm_fd_phyaddr(es->compositor, fd);
348 - drm_close_handle(es->compositor, fd);
349 - if (ps->tde->image.phyaddr == 0) {
350 -     if (ptr) {
351 -         munmap(ptr, stride * buffer->height);
352 -     }
353 -     return 0;
354 -
355 + struct pixman_renderer *pr = get_renderer(es->compositor);
356 + /*
357 + if (pr->tde->use_tde) {
358 +     ps->tde->image.phyaddr = drm_fd_phyaddr(es->compositor, fd);
359 +     drm_close_handle(es->compositor, fd);
360 +     if (ps->tde->image.phyaddr == 0) {
361 +         if (ptr) {
362 +             munmap(ptr, stride * buffer->height);
363 +         }
364 +         return 0;
365 +     }
366 + }
367 + */
```

5. 内核中将 format 更改为 DRM_FORMAT_XBGR8888, 颜色显示正常。

7 HDC

因为在原生 HDC 主要依赖 usb 的 device 功能, 需要开启 usb 的 device 功能, 需要适配 usb/gadget/function 层驱动。

接着配置 usb device 模式的文件, 主要在

device/hihope/dayu/build/rootfs/init.dayu.usb.rc 中

```
on boot
  mkdir /dev/usb-ffs 0770 shell shell
  mkdir /dev/usb-ffs/hdc 0770 shell shell
  mount configfs none /config
  mkdir /config/usb_gadget/g1 0770 shell shell
  write /config/usb_gadget/g1/idVendor 0x12d1
  write /config/usb_gadget/g1/idProduct 0x5000
  write /config/usb_gadget/g1/os_desc/use 1
  write /config/usb_gadget/g1/bcdDevice 0x0404
  write /config/usb_gadget/g1/bcdUSB 0x0200
  mkdir /config/usb_gadget/g1/strings/0x409 0770
  copy /sys/block/mmcblk0/device/cid /config/usb_gadget/g1/strings/0x409/serialnumber
  write /config/usb_gadget/g1/strings/0x409/manufacturer "HiHope"
  write /config/usb_gadget/g1/strings/0x409/product "HiHope Phone"
  mkdir /config/usb_gadget/g1/functions/accessory.gs2
  mkdir /config/usb_gadget/g1/functions/audio_source.gs3
  mkdir /config/usb_gadget/g1/functions/ffs.hdc
  mkdir /config/usb_gadget/g1/functions/mtp.gs0
  mkdir /config/usb_gadget/g1/functions/ptp.gs1
  mkdir /config/usb_gadget/g1/functions/midi.gs5
  mkdir /config/usb_gadget/g1/configs/b.1 0770 shell shell
  mkdir /config/usb_gadget/g1/configs/b.1/strings/0x409 0770 shell shell
  write /config/usb_gadget/g1/os_desc/b vendor code 0x1
  write /config/usb_gadget/g1/os_desc/qw sign "MSFT100"
  write /config/usb_gadget/g1/configs/b.1/MaxPower 500
  symlink /config/usb_gadget/g1/configs/b.1 /config/usb_gadget/g1/os_desc/b.1
  mount functionfs hdc /dev/usb-ffs/hdc uid=2000,gid=2000
  setprop sys.usb.configfs 1
  setprop sys.usb.controller "musb-hdrc.0.auto"
```

正常执行完后在 dev 下生成/dev/usb-ffs/hdc/ep0 ep1 ep2 节点

且 log 中会看到明显的枚举信息, 查看设备管理器可以看到如图信息, 表示设备端准备完毕

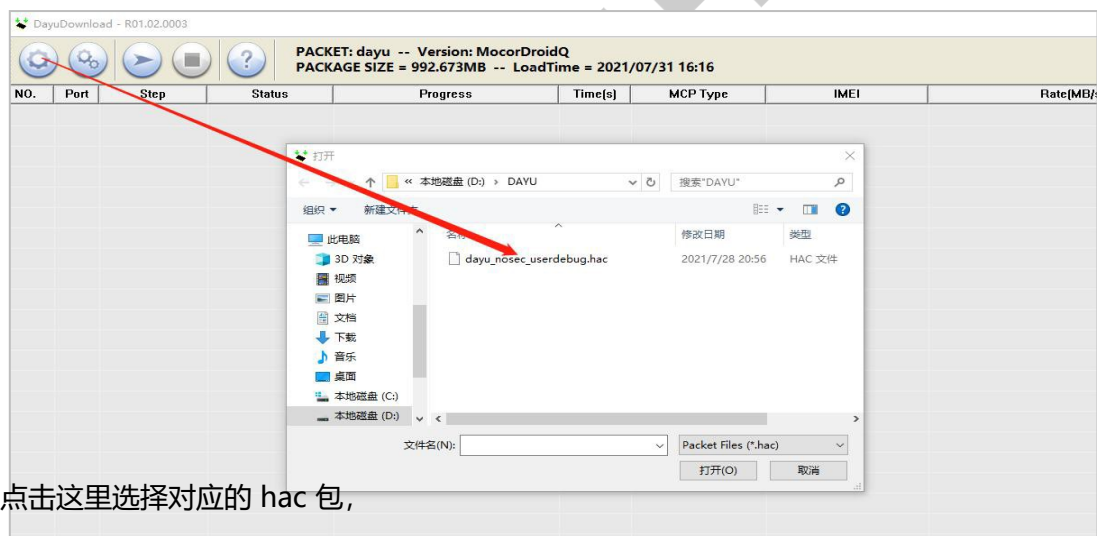


使用 `hdc list targets` 可以拿到设备 id

使用 `hdc shell` 可以正常进入 shell 终端，则表示 hcd 调试成功

8 版本烧录

首先要烧写基础 base 版本,烧写方法如下:



点击这里选择对应的 hac 包,



拔掉 usb 线，点击开始，同时按住 power+ vol-键,插入 usb ，自动开始下载，待开始下

载后，手可以松开了。

接下来就可以替换编译出来的 system.img ,userdata.img,vendor.img,如下图所示，选对
应的镜像

