



MTK 平台 CAMERA 驱动浅析

Camera Driver analysis in the platform of MTK

Document Number:

Preliminary (Released) Information

Revision:0.1

Release Date:

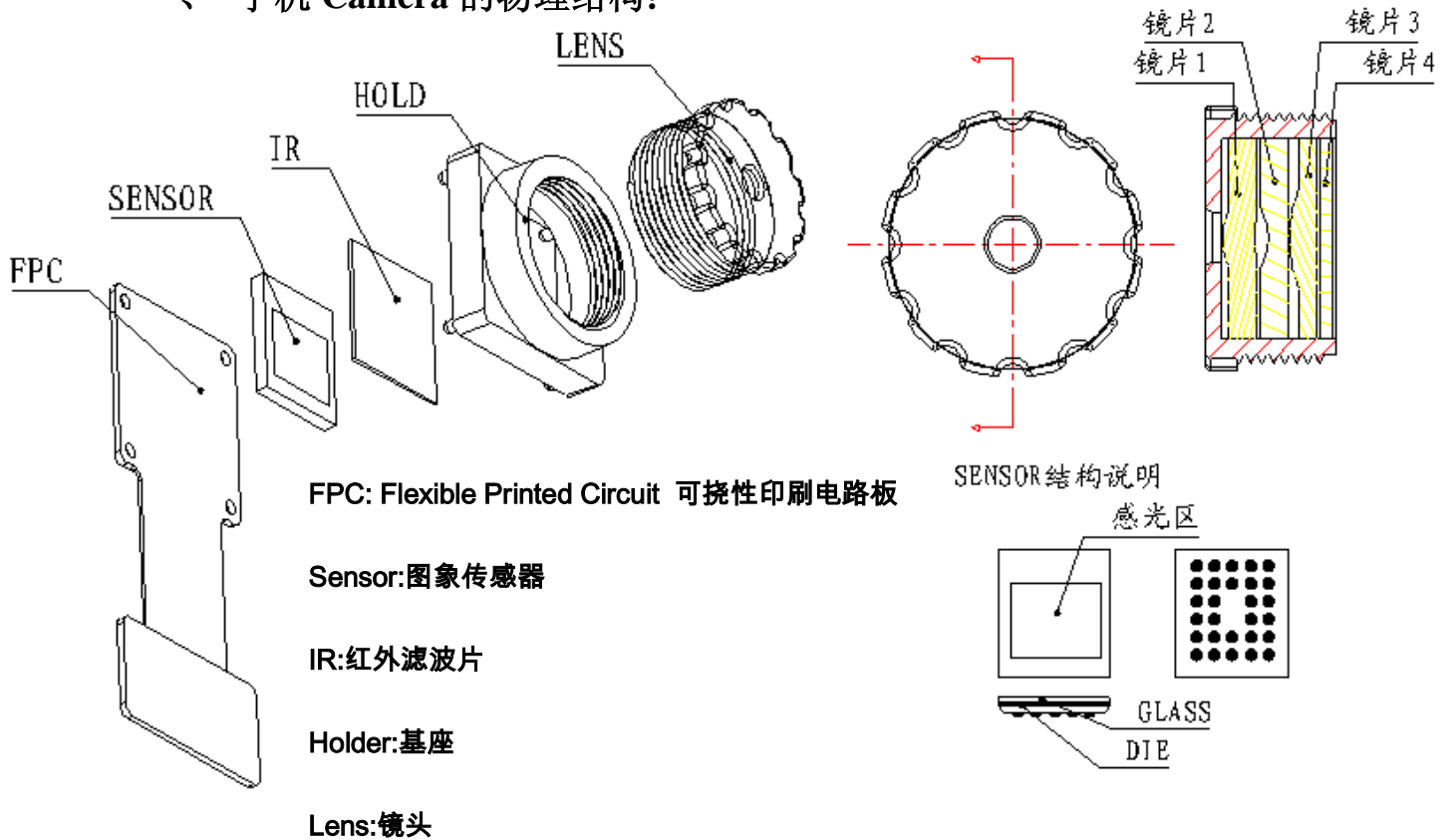
Revision History

Revision	Date (dd/mm/yyyy)	Author	Comments
0.1	14/02/2012	Guoqing Zhang	Draft Version

Contents

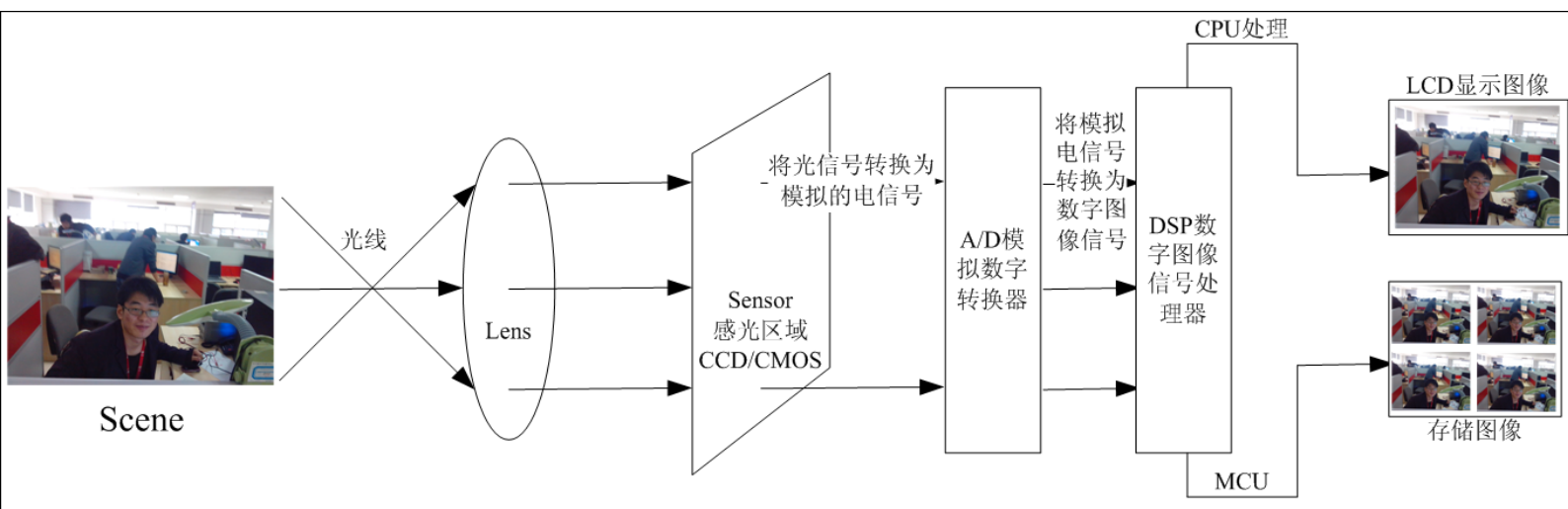
一、 手机 CAMERA 的物理结构:	- 4 -
二、 CAMERA 的成像原理:	- 4 -
三、 CAMERA 常见的数据输出格式:	- 5 -
四、 阅读 CAMERA 的规格书 (以 TRULY 模组 OV5647_RAW 为例):	- 6 -
五、 CAMERA 的硬件原理图及引脚	- 7 -
1、 电源部分:	- 7 -
2、 SENSOR INPUT 部分:	- 7 -
3、 SENSOR OUTPUT 部分:	- 7 -
4、 I2C 部分: SCL, I2C 时钟信号线和 SDA, I2C 数据信号线。	- 7 -
六、 MTK 平台 CAMERA 驱动架构:	- 8 -
七、 MTK 平台 CAMERA 相关代码文件 (以下代码均为 MTK6575 平台):	- 9 -
1、 CAMERASENSOR 驱动相关文件	- 9 -
2、 SENSOR ID 和一些枚举类型的定义	- 9 -
3、 SENSOR 供电	- 9 -
4、 KERNEL SPACE 的 SENSORLIST, IMGSENSOR 模块注册	- 9 -
5、 USER SPACE 的 SENSORLIST, 向用户空间提供支持的 SENSORLIST	- 10 -
6、 SENSOR 效果调整的接口	- 10 -
八、 CAMERA 模块驱动、设备与总线结构:	- 11 -
A) 驱动的注册:	- 11 -
B) 设备的注册:	- 11 -
C) 总线的匹配:	- 12 -
九、 CAMERA 驱动工作流程:	- 13 -
十、 CAMERA 驱动添加、调试流程:	- 17 -

一、手机 Camera 的物理结构：



二、Camera 的成像原理：

景物通过镜头（LENS）生成的光学图像投射到图像传感器(Sensor)表面上，然后转为模拟的电信号，经过 A/D（模数转换）转换后变为数字图像信号，再送到数字信号处理芯片（DSP）中加工处理，再通过 IO 接口传输到 CPU 中处理，通过 LCD 就可以看到图像了。



图像传感器（SENSOR）是一种半导体芯片，其表面包含有几十万到几百万的光电二极管。光电二极管受到光照射时，就会产生电荷。目前的 SENSOR 类型有两种：

CCD（Charge Couple Device），电荷耦合器件，它是目前高像素类 sensor 中比较成熟的成像器件，是以一行为单位的电流信号。

CMOS（Complementary Metal Oxide Semiconductor），互补金属氧化物半导体。CMOS 的信号是以点为单位的电荷信号，更为敏感，速度也更快，更为省电。

ISP 的性能是决定影像流畅的关键，JPEG encoder 的性能也是关键指标之一。而 JPEG encoder 又分为硬件 JPEG 压缩方式，和软件 RGB 压缩方式。

DSP 控制芯片的作用是：将感光芯片获取的数据及时快速地传到 baseband 中并刷新感光芯片，因此控制芯片的好坏，直接决定画面品质（比如色彩饱和度、清晰度）与流畅度。

三、 Camera 常见的数据输出格式：

常见的数据输出格式有：Rawdata 格式、YUV 格式、RGB 格式。

RGB 格式：采用这种编码方法，每种颜色都可用三个变量来表示红色、绿色以及蓝色的强度。每一个像素有三原色 R 红色、G 绿色、B 蓝色组成。

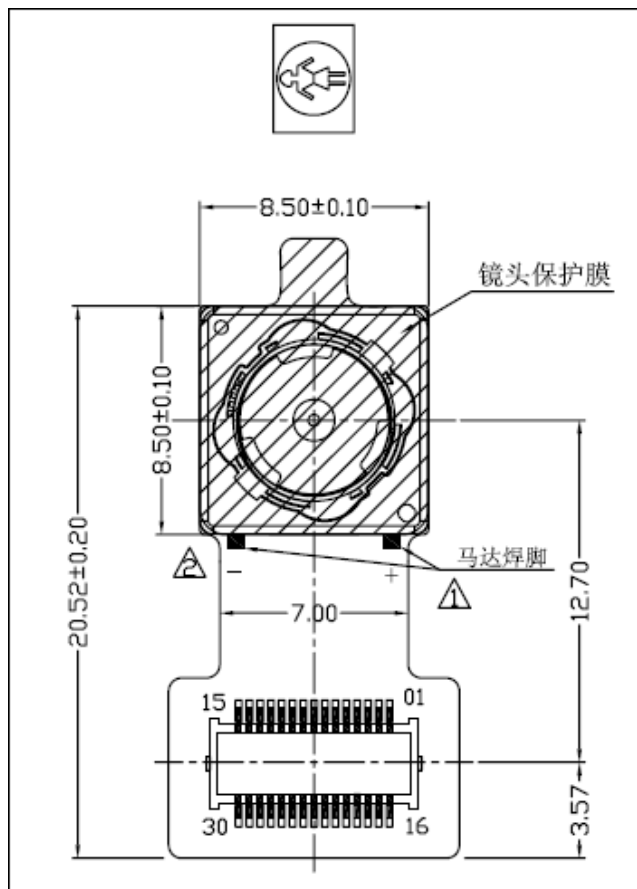
YUV 格式：其中“Y”表示明亮度(Luminance 或 Luma),就是灰阶值；而“U”和“V”表示色度(Chrominance 或 Chroma),是描述影像色彩及饱和度，用于指定像素的颜色。

RAW DATA 格式：是 CCD 或 CMOS 在将光信号转换为电信号时的电平高低的原始记录，单纯地将没有进行任何处理的图像数据，即摄像元件直接得到的电信号进行数字化处理而得到的。

支持 YUV/RGB 格式的模组，一般会在模组上集成 ISP（Image Single Processor），经过 A/D 转换过的原始数据经过 ISP 处理生成 YUV 标准格式传到 BB。一般来说，这种设计适用于低像素 Camera 的要求，会在主板上省去一个 DSP，可降低成本。在调试过程中，YUV/RGB 格式的摄像头，其所有参数都可在 kernel 层通过寄存器来控制。调试一般由 sensor 的原厂支持。

支持 RawData 格式的模组，由于感光区域的需求，不会再模组内集成 ISP 以最大程度的增大感光区域的面积，提高照片质量。模组把原始的数字信号传给 BB 上的 DSP 进行处理，MTK 自带的 DSP 一般包含 ISP、JPEG encoder、和 DSP 控制芯片。在调试的时候图像的效果需要 MTK 在 HAL 层的参数进行支持。

四、 阅读 Camera 的规格书（以 Truly 模组 OV5647_Raw 为例）：



PIN NO.	SIGNAL
1	D5
2	D6
3	D7
4	D8
5	D9
6	NC
7	RESET
8	NC
9	DOVDD_1.8V
10	AVDD_2.8V
11	AF_VCC_2.8V
12	PWDN
13	DGND
14	STROBE
15	D1

16	DGND
17	XCLK
18	PCLK
19	D2
20	D3
21	D4
22	NC
23	HREF
24	VSYNC
25	SIOD
26	SIOC
27	AGND
28	DGND
29	D0
30	DNGD

主要参数 (Module Specification)	
焦距 (EFL)	3.37 mm
光圈 (F.NO)	2.8±5%
视场角 (View Angle)	67.4° ±2°
畸变 (Distortion)	< 1 %
景深 (Focusing Range)	10 cm~Infinity
感光芯片 (Chip Type)	OV5647
像素 (Array Size)	5.0M
镜头类型 (lens Size)	1/4 INCH 4P+IR

备注：

1. 带*尺寸为关键尺寸；

2. DVDD内部供电；

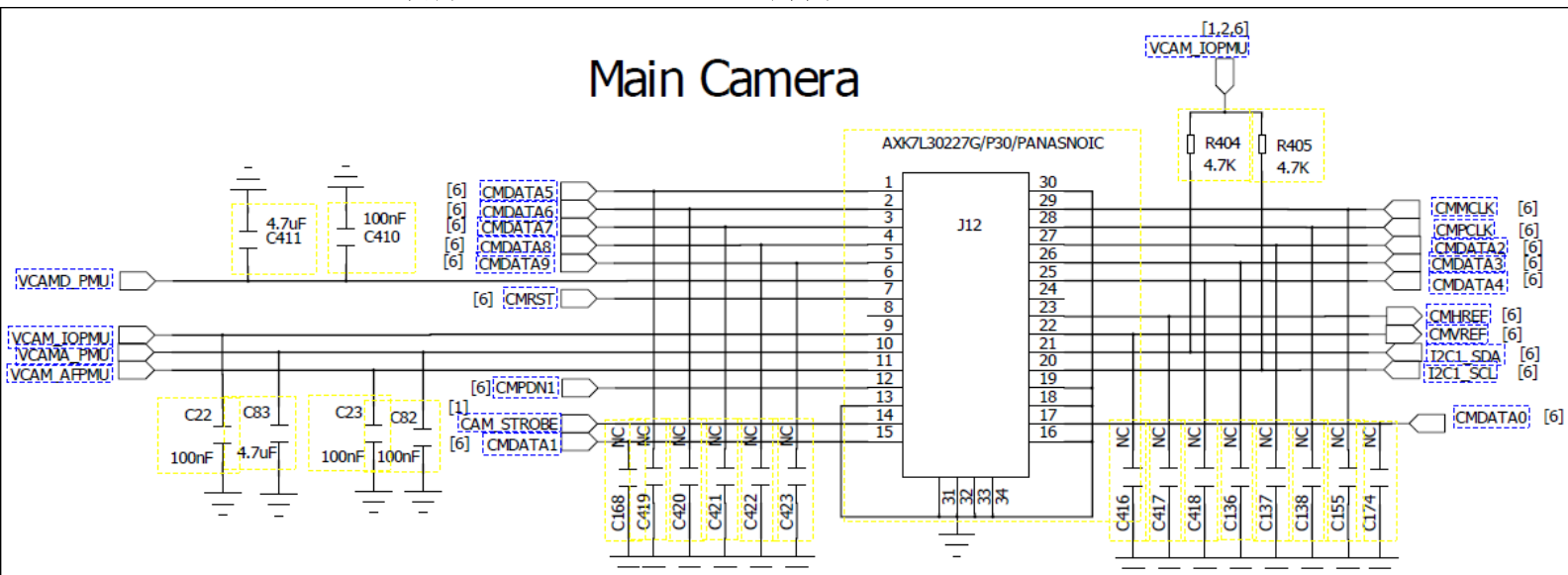
3. Driver IC: DW9712-WCMNGTL

由于RAWDATA模式模组
不带ISP，所以不用DVDD

Actuator Specification	
Type	VCM Parameter
Control Mode	IIC
Supply Voltage	2.8V~3.3 V
Coil Resistance	16 ± 2 ohm
Rated Current	100mA or less
Lens Movement	0 ~ 0.25 mm

五、 Camera 的硬件原理图及引脚

(以 W19S 项目 MainCameraOV5647 为例):



从上面可看出，连接 Camera 的 30 根 Pin 脚可大致分为以下几类：

1、电源部分：

- VCAMD 就是 DVDD 数字供电，主要给 ISP 供电，由于 RAWDATA 格式的 sensor 其 ISP 是在 BB 端，所以将其引脚将其 NC。从上面的规格书上可以看出 DVDD 是内部 BB 端供电。模组已将其 NC 掉了；
- VCAM_IO 就是 VDDIO 数字 IO 电源主要给 I2C 部分供电；
- VCAMA 就是 AVDD 模拟供电，主要给感光区和 ADC 部分供电；
- VCAM_AF 是对 Camera 自动对焦马达的供电。

2、Sensor Input 部分：

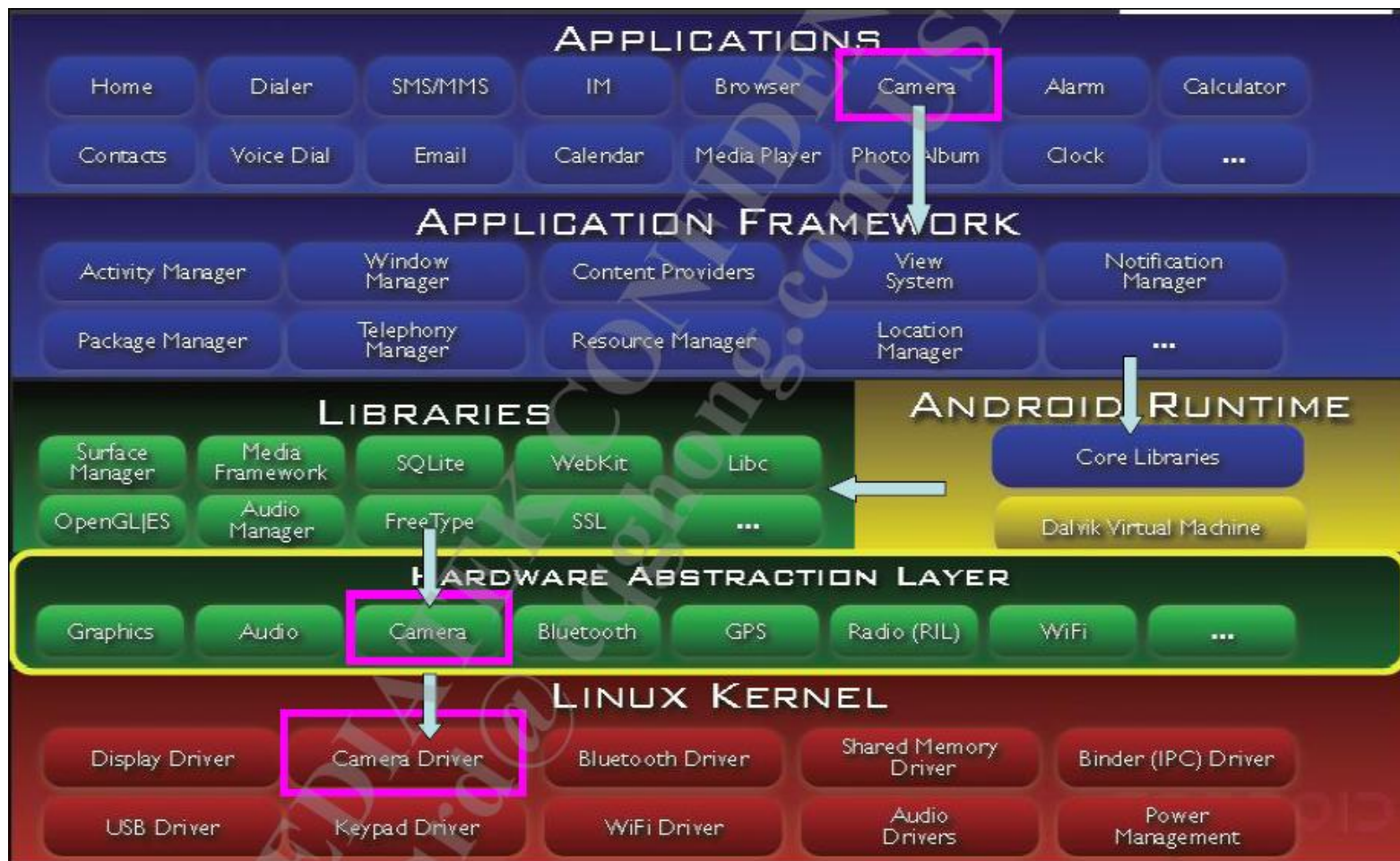
- Reset 信号，用于复位、初始化。
- Standby/PowerDown 信号，用于进入待机模式，降低功耗。
- Mclk，即 MasterClock 信号，是由 BB 端提供。

3、Sensor OutPut 部分：

- Pclk，即 PixelClock 信号，由 MCLK 分频得到，作为外部时钟控制图像传输帧率
- HSYNC，行同步信号，其上升沿表示新一列行图像数据的开始。
- VSYNC，帧同步信号，其下降沿表示新的一帧图片的开始。
- D0-D9 一共 10 根数据线（8/10 根等）；

4、I2C 部分：SCL，I2C 时钟信号线和 SDA，I2C 数据信号线。

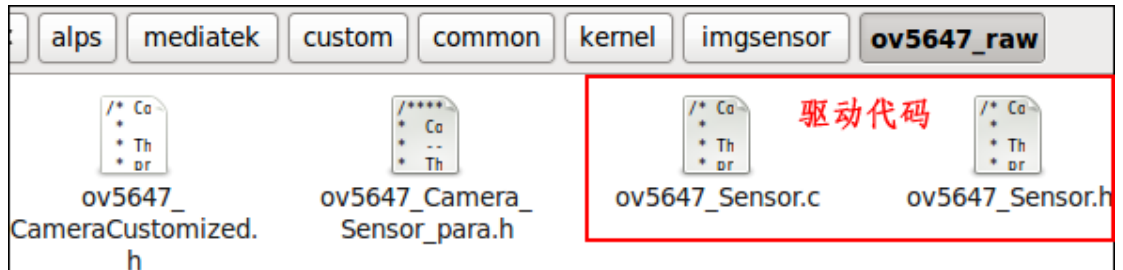
六、 MTK 平台 Camera 驱动架构：



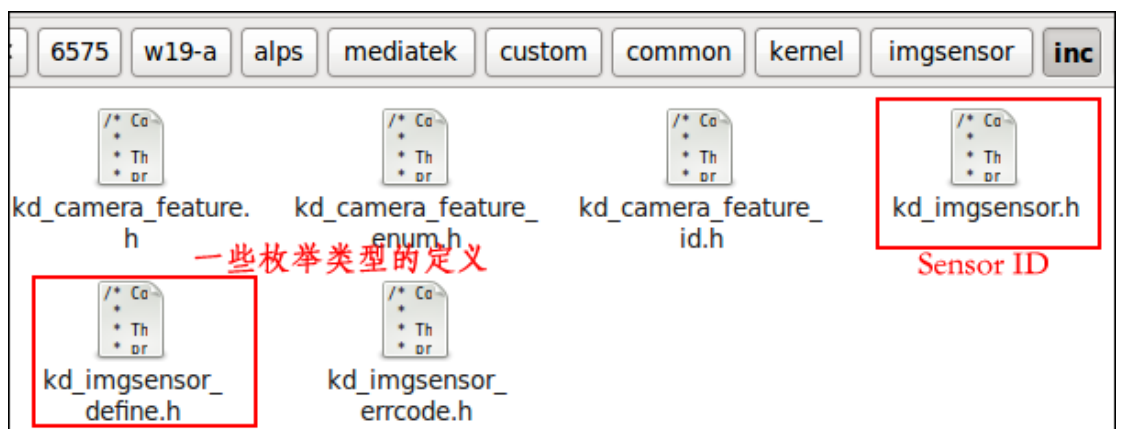
上图的架构相信大家都有了一定的了解，android 将系统大致分为应用层、库文件和硬件抽象层、Linux 内核三层。在底层的内核空间，Camera 的 driver 将其驱动起来以后，将硬件驱动的接口交给硬件抽象层，android 上层的 Camera 应用程序在 android 实时系统中的虚拟机中，加载 android 留给 Camera 公用的一些库文件，调用硬件抽象层的接口来控制 Camera 硬件来实现功能。当然，如果是 Raw 模式的 Camera，还需要在硬件抽象层调用一些参数来控制 Camera 的效果。

七、 MTK 平台 Camera 相关代码文件(以下代码均为 MTK6575 平台):

1、 CameraSensor 驱动相关文件



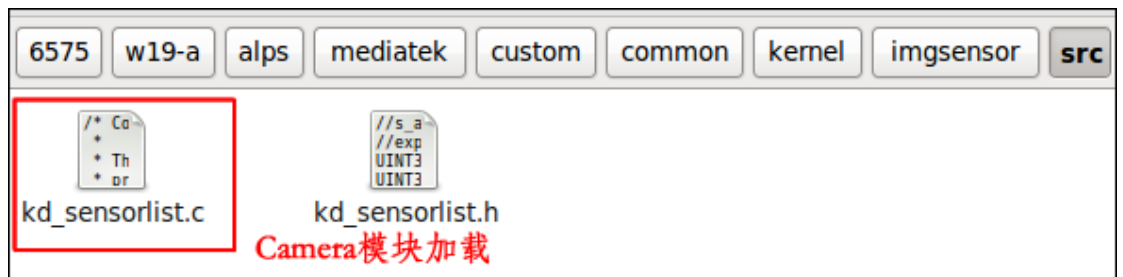
2、 Sensor ID 和一些枚举类型的定义



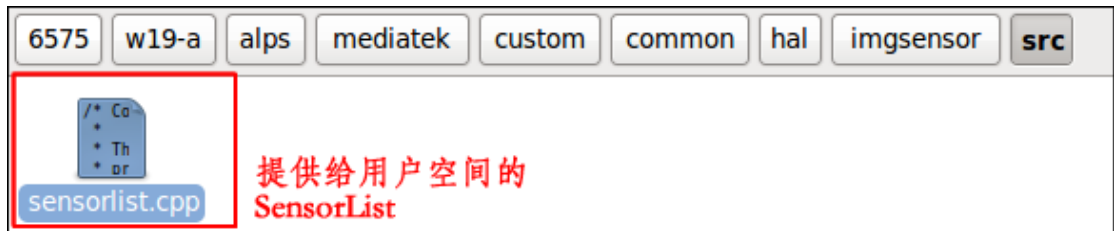
3、 Sensor 供电



4、 Kernel Space 的 SensorList, imgsensormodule 模块注册



5、User Space 的 SensorList，向用户空间提供支持的 SensorList



6、Sensor 效果调整的接口

w19-a alps mediatek custom common hal imgsensor ov5647 raw			
Name	Size	Type	RAW格式
camera_AE_PLineTable_ov5647.h	193.6 KB	C header	调整自动曝光值
camera_info_ov5647_mt6575.h	5.4 KB	C header	
camera_isp_pca_ov5647_mt6573.h	28.8 KB	C header	调整ISP参数
camera_isp_pca_ov5647_mt6575.h	28.8 KB	C header	
camera_isp_regs_ov5647_mt6573.h	14.9 KB	C header	
camera_isp_regs_ov5647_mt6575.h	15.9 KB	C header	
camera_tuning_para_ov5647_mt6573.cpp	106.8 KB	C++ source code	
camera_tuning_para_ov5647_mt6575.cpp	261.0 KB	C++ source code	

w19-a alps mediatek custom common hal imgsensor ov5642 yuv			
Name	Size	Type	YUV的调整参数 几乎所有的Sensor都一致
camera_info_ov5642_yuv.cpp	18.3 KB	C++ source code	
camera_info_ov5642_yuv.h	4.1 KB	C header	
camera_sensor_para_ov5642_yuv.h	5.7 KB	C header	
camera_tuning_para_ov5642_yuv.cpp	24.2 KB	C++ source code	
cfg_ftbl_ov5642_yuv.h	14.0 KB	C header	
feature_ov5642_yuv.cpp	7.6 KB	C++ source code	

八、 Camera 模块驱动、设备与总线结构：

一般在 Linux 设备驱动模型中，我们只需要关心总线、设备、驱动这三个实体。总线会充当红娘对加载于其上的设备与驱动进行配对，对于 Camera 模块也不例外，下面从总线、设备、驱动的角度来分析 Camera 模块驱动的注册、匹配与加载过程。

a) 驱动的注册：

在 (\custom\common\kernel\imgsensor\src\Kd_sensorlist.c) CAMERA_HW_i2C_init 这个函数里通过 **Platform_driver_register**(&g_stCAMERA_HW_Driver)把 Camera 模块驱动注册到 Platform 总线上。而 g_stCAMERA_HW_Driver 是对结构体 Platform_driver 这个结构体的填充。

```
static struct platform_driver g_stCAMERA_HW_Driver = {
    .probe      = CAMERA_HW_probe,
    .remove     = CAMERA_HW_remove,
    .suspend    = CAMERA_HW_suspend,
    .resume     = CAMERA_HW_resume,
    .driver     = {
        .name    = "image_sensor",
        .owner   = THIS_MODULE,
    },
};
```

(Kernel\include\linux\Platform_device.h)

```
struct platform_driver {
    int (*probe)(struct platform_device *);
    int (*remove)(struct platform_device *);
    void (*shutdown)(struct platform_device *);
    int (*suspend)(struct platform_device *, pm_message_t state);
    int (*resume)(struct platform_device *);
    struct device_driver driver;
    const struct platform_device_id *id_table;
};
```

Platform_driver 这个结构体包含 Probe ()、Remove () 等函数来完成驱动的填充。

b) 设备的注册：

对 platform_device 的定义通常在 BSP 的板级文件：

(kernel\arch\sh\boards\mach-ap325rxa\Setup.c) 中实现，在板级文件中，将 platform_device 归纳为一个数组，最终通过 platform_add_device()函数统一注册：

```
static struct platform_device camera_device = {
    .name      = "soc_camera_platform",
    .dev       = {
        .platform_data = &camera_info,
        .release       = dummy_release,
    },
};
```

```
static int ap325rxa_camera_add(struct soc_camera_link *icl,
                               struct device *dev)
{
    if (icl != &camera_link || camera_probe() <= 0)
        return -ENODEV;

    camera_info.dev = dev;

    return platform_device_register(&camera_device);
}
```

```
static struct soc_camera_link camera_link = {
    .bus_id      = 0,
    .add_device  = ap325rxa_camera_add,
    .del_device  = ap325rxa_camera_del,
    .module_name = "soc_camera_platform",
    .priv        = &camera_info,
};
```

c) 总线的匹配:

既然是驱动 Platform_device 那对应的设备必然是挂载 Platform 总线上的 Platform_device, Platform 总线是 Linux 系统提供的一种机制, 不同于 I2C、I2S 等总线, 它是一种虚拟的总线。

Linux 系统为 Platform 总线定义了一个 bus_type 的实例 Platform_bus_type:

(Kernel\drivers\base\platform.c)

```
struct bus_type platform_bus_type = {
    .name      = "platform",
    .dev_attrs = platform_dev_attrs,
    .match     = platform_match,
    .uevent    = platform_uevent,
    .pm        = &platform_dev_pm_ops,
};
```

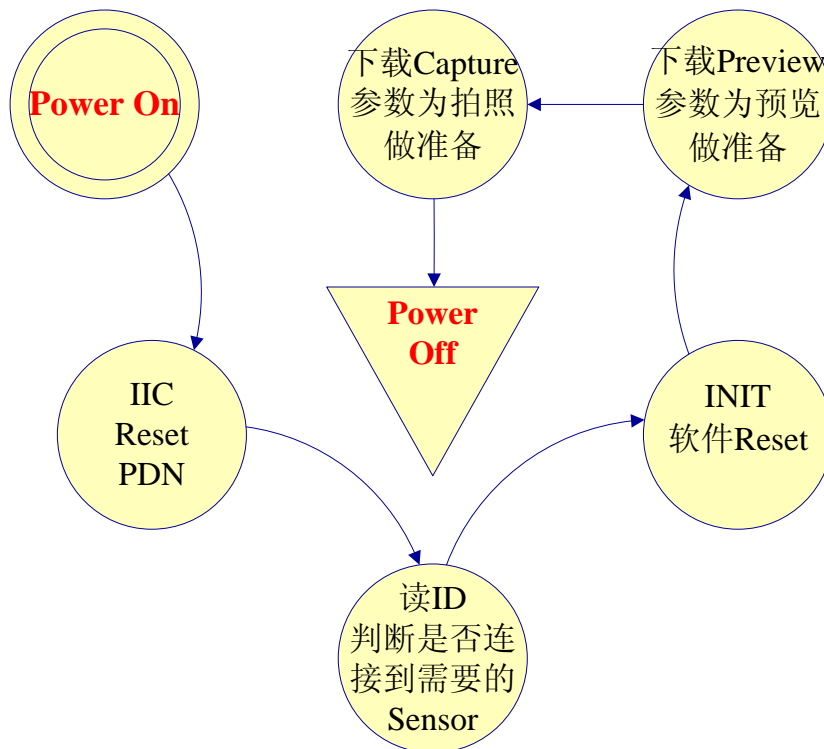
Platform 总线通过 platform_match 这个成员函数来确定 platform_device 与 platform_driver 如何进行匹配:

```
static int platform_match(struct device *dev, struct device_driver *drv)
{
    struct platform_device *pdev = to_platform_device(dev);
    struct platform_driver *pdrv = to_platform_driver(drv);

    /* match against the id table first */
    if (pdrv->id_table)
        return platform_match_id(pdrv->id_table, pdev) != NULL;

    /* fall-back to driver name match */
    return (strcmp(pdev->name, drv->name) == 0);
}
```

九、 Camera 驱动工作流程:



从上图可以清晰的了解到 Camera 的一个工作流程主要分为这么七步:

1. 打开 Camera Power LDO, 让 Camera 有能量保证。
2. 打开 IIC, 设置 PDN 引脚, 使 Camera 退出 Standby 模式, 按照要求让 Reset 脚做一个复位动作。
3. 读一下 sensor 的版本 ID, 这样可以让你确认是否连接上你想要的 sensor。
4. 对 Sensor 进行初始化下载最基本的参数让 Sensor 工作起来, 可能包括软复位。
5. 下载 preview 的参数, 为预览动作准备。
6. 下载 Capture 的参数, 为拍照动作准备。
7. 设置 PDN 引脚, 使 Sensor 进入 Standby 模式, 或者关掉 LDO 等动作, 退出 Camera。

我们都知道, Linux 内核是通过模块的机制来加载设备驱动的, 那么接下来我们就从设备模块加载的角度来看下 Camera 工作流程的驱动代码是如何工作的。

在 `-alps\mediatek\custom\common\kernel\imgsensor\src\kd_sensorlist.c` 中可以看到:

```
module_init(CAMERA_HW_i2C_init);
module_exit(CAMERA_HW_i2C_exit);
```

在这里 Linux 内核加载和卸载 Camera 模块。

```
static struct platform_driver g_stCAMERA_HW_Driver = {
    .probe      = CAMERA_HW_probe,
    .remove     = CAMERA_HW_remove,
    .suspend    = CAMERA_HW_suspend,
    .resume     = CAMERA_HW_resume,
    .driver     = {
        .name    = "image_sensor",
        .owner   = THIS_MODULE,
    }
};
```

Camera 模块初始化开始向总线注册驱动，在 Platform_driver 的成员函数.probe()中，通过 i2c_add_driver(&CAMERA_HW_i2c_driver)向 I2C 申请，而 CAMERA_HW_i2c_driver 这个结构体里填充的是将 Camera 作为一个字符设备在 I2C 上进行注册：

```
struct i2c_driver CAMERA_HW_i2c_driver = {
    .probe = CAMERA_HW_i2c_probe,
    .remove = CAMERA_HW_i2c_remove,
    .detect = CAMERA_HW_i2c_detect,
    .driver.name = CAMERA_HW_DRVNAME,
    .id_table = CAMERA_HW_i2c_id,
    .address_data = &addr_data,
};
```

```
static int CAMERA_HW_i2c_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
    int i4RetValue = 0;
    PK_DBG("[CAMERA_HW] Attach I2C \n");

    //get sensor i2c client
    g_pstI2Cclient = client;
    //set I2C clock rate
    g_pstI2Cclient->timing = 200;//200k

    //Register char driver
    i4RetValue = RegisterCAMERA_HWCharDrv();

    if(i4RetValue){
        PK_ERR("[CAMERA_HW] register char device failed! \n");
        return i4RetValue;
    }

    //spin_lock_init(&g_CamHWLock);

    PK_DBG("[CAMERA_HW] Attached!! \n");
    return 0;
} ? end CAMERA_HW_i2c_probe ?
```

在 RegisterCAMERA_HWCharDrv () 中

cdev_init(g_pCAMERA_HW_CharDrv, &g_stCAMERA_HW_fops);对设备进行初始化，并将 g_stCAMERA_HW_fops 这个文件操作函数作为上层对 Camera 设备操作的接口留给上层进行调用：


```
static const struct file_operations g_stCAMERA_HW_fops =
{
    .owner = THIS_MODULE,
    .open = CAMERA_HW_Open,
    .release = CAMERA_HW_Release,
    #ifdef USE_NEW_IOCTL
    .unlocked_ioctl = CAMERA_HW_Ioctl
    #else
    .ioctl = CAMERA_HW_Ioctl
    #endif
};
```

其中成员函数 open（）只是初始化一个原子变量留给系统调用。ioctl（）才是整个 Camera 驱动的入口：

```
static const struct file_operations g_stCAMERA_HW_fops =
{
    .owner = THIS_MODULE,
    .open = CAMERA_HW_Open,
    .release = CAMERA_HW_Release,
    #ifdef USE_NEW_IOCTL
    .unlocked_ioctl = CAMERA_HW_Ioctl
    #else
    .ioctl = CAMERA_HW_Ioctl
    #endif
};
```

CAMERA_HW_Ioctl（）是上层文件操作系统操作底层硬件的方法，它先对 Camera 需要的 Buffer 做一个初始化，然后建立对 Cameraopen、getinfo 等操作的接口：

```
case KDIMGSENSORIOC_X_SET_DRIVER:
    i4RetValue = kdSetDriver((unsigned int*)pBuff);
    break;
case KDIMGSENSORIOC_T_OPEN:
    i4RetValue = adopt_CAMERA_HW_Open();
    break;
case KDIMGSENSORIOC_X_GETINFO:
    i4RetValue = adopt_CAMERA_HW_GetInfo(pBuff);
    break;
case KDIMGSENSORIOC_X_GETRESOLUTION:
    i4RetValue = adopt_CAMERA_HW_GetResolution(pBuff);
    break;
case KDIMGSENSORIOC_X_FEATURECONCTROL:
    i4RetValue = adopt_CAMERA_HW_FeatureControl(pBuff);
    break;
case KDIMGSENSORIOC_X_CONTROL:
    i4RetValue = adopt_CAMERA_HW_Control(pBuff);
    break;
case KDIMGSENSORIOC_T_CLOSE:
    i4RetValue = adopt_CAMERA_HW_Close();
    break;
case KDIMGSENSORIOC_T_CHECK_IS_ALIVE:
    i4RetValue = adopt_CAMERA_HW_CheckIsAlive();
    break;
default :
    PK_DBG("No such command \n");
    i4RetValue = - EPERM;
    break;
```

通过判断 Sensor 状态的逻辑值来进行具体的操作，对于这个值的定义在：

Mediatek\custom\common\kernel\imgsensor\inc\Kd_imgsensor.h 中

```
//This command will TBD
#define KDIMSENSORIOC_T_OPEN      _IO(IMGSENSORMAGIC,0)
//sensorGetInfo
//This command will TBD
#define KDIMSENSORIOC_X_GETINFO    _IOWR(IMGSENSORMAGIC,5,ACDK
//sensorGetResolution
//This command will TBD
#define KDIMSENSORIOC_X_GETRESOLUTION _IOWR(IMGSENSORMAGIC,10,
//sensorFeatureControl
//This command will TBD
#define KDIMSENSORIOC_X_FEATURECONCTROL _IOWR(IMGSENSORMAGIC,15
//sensorControl
//This command will TBD
#define KDIMSENSORIOC_X_CONTROL    _IOWR(IMGSENSORMAGIC,20,ACD
//sensorClose
//This command will TBD
#define KDIMSENSORIOC_T_CLOSE      _IO(IMGSENSORMAGIC,25)
//set sensor driver
#define KDIMSENSORIOC_X_SET_DRIVER _IOWR(IMGSENSORMAGIC,40,u32
//sensorSearch
#define KDIMSENSORIOC_T_CHECK_IS_ALIVE _IO(IMGSENSORMAGIC, 30)
```

在 KdSetDriver（）中通过判断 name 和 ID 匹配具体型号的 sensor 的驱动，判断它是主摄还是次摄，并对它进行初始化：

```
if (drvIdx < MAX_NUM_OF_SUPPORT_SENSOR)
{
    if (NULL == pSensorList[drvIdx].SensorInit)
    {
        PK_ERR("ERROR:kdSetDriver()\n");
        return -EIO;
    }
}
```

```
typedef struct
{
    MUINT32 SensorId;
    MUINT8 drvname[32];
    MUINT32 (* SensorInit)(PSENSOR_FUNCTION_STRUCT *pfFunc);
} ACDK_KD_SENSOR_INIT_FUNCTION_STRUCT, *PACDK_KD_SENSOR_INIT_FUNCTION_STRUCT;
```

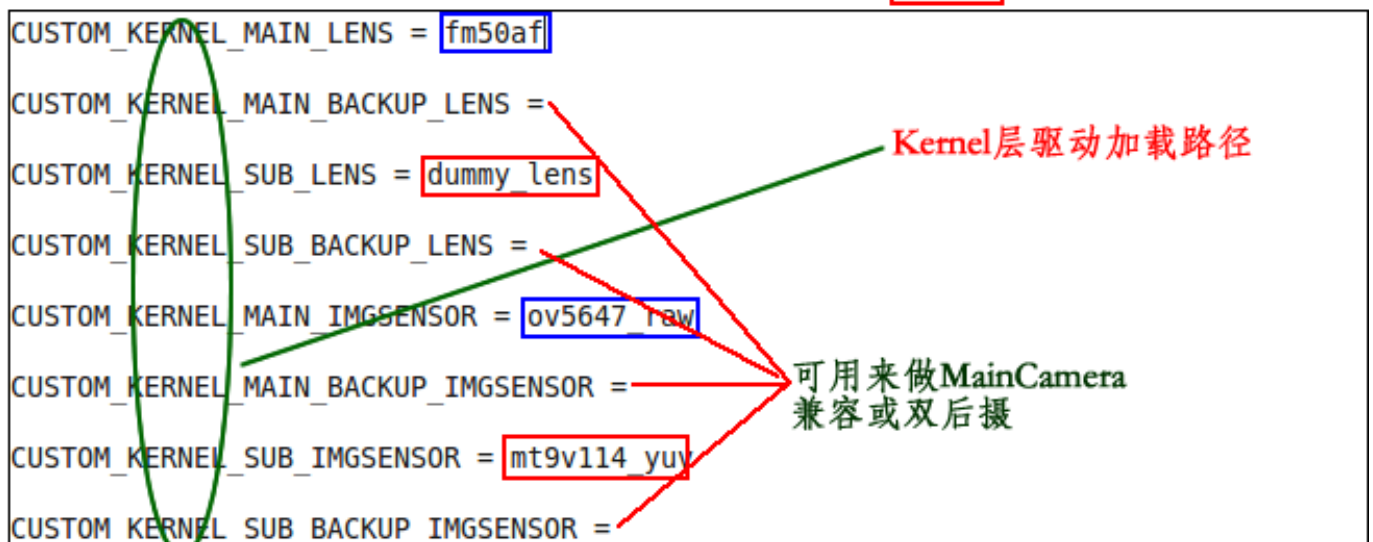
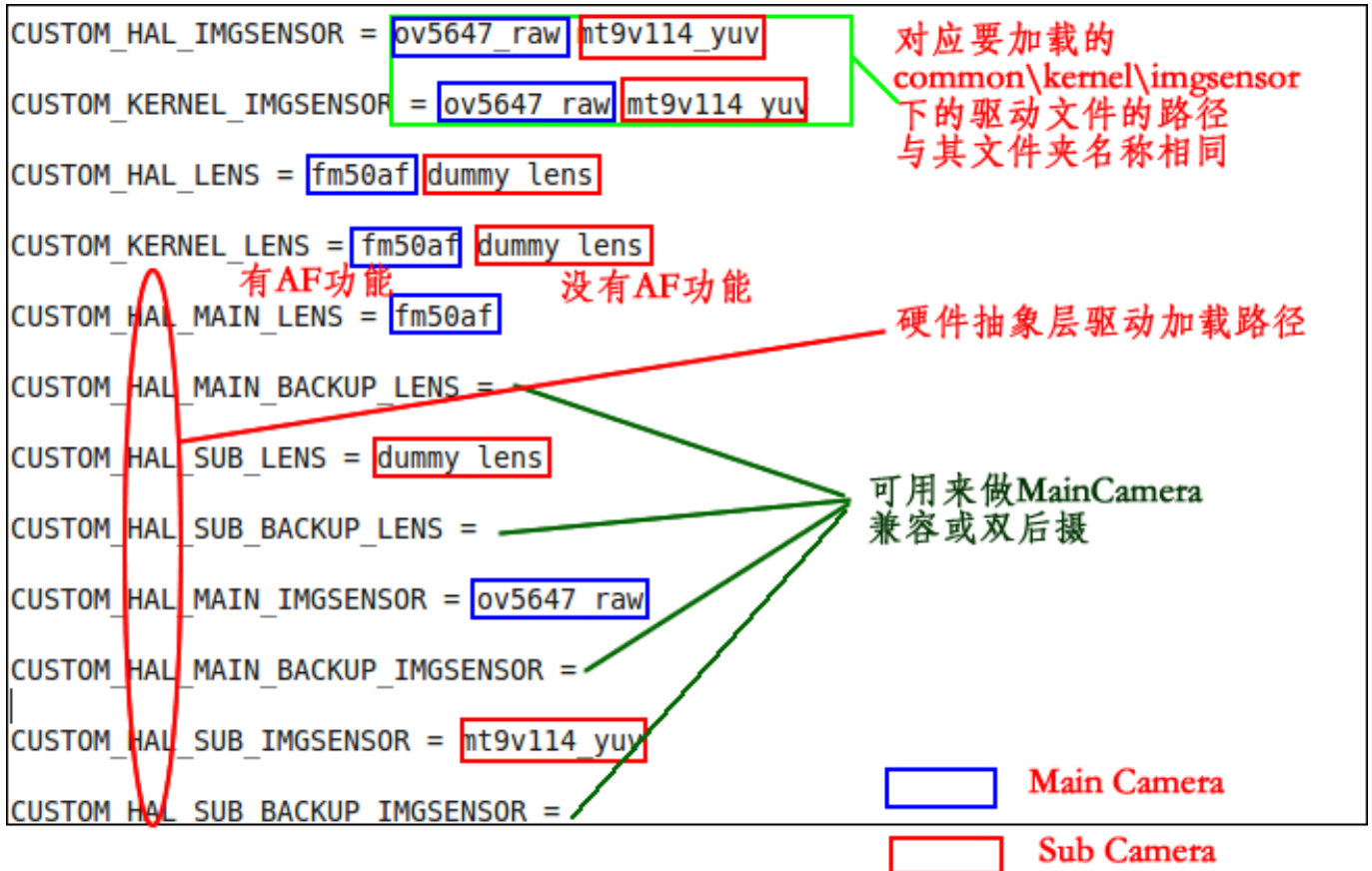
通过 NAME 和 ID 匹配完成后会将 PSENSOR_FUNCTION_STRUCT *pfFunc 这个结构体匹配到具体型号的驱动代码中：

```
SENSOR_FUNCTION_STRUCT SensorFuncOV5647=
{
    OV5647Open,
    OV5647GetInfo,
    OV5647GetResolution,
    OV5647FeatureControl,
    OV5647Control,
    OV5647Close
};
```

到这里，整个 Camera 驱动从总线注册到完成具体 sensor 的初始化的流程就完成了，CAMERA_HW_Ioclt（）中其他的 ioctl 操作函数最后都会都在 \$sensor\$_sensor.c 中实现。

十、Camera 驱动添加、调试流程：

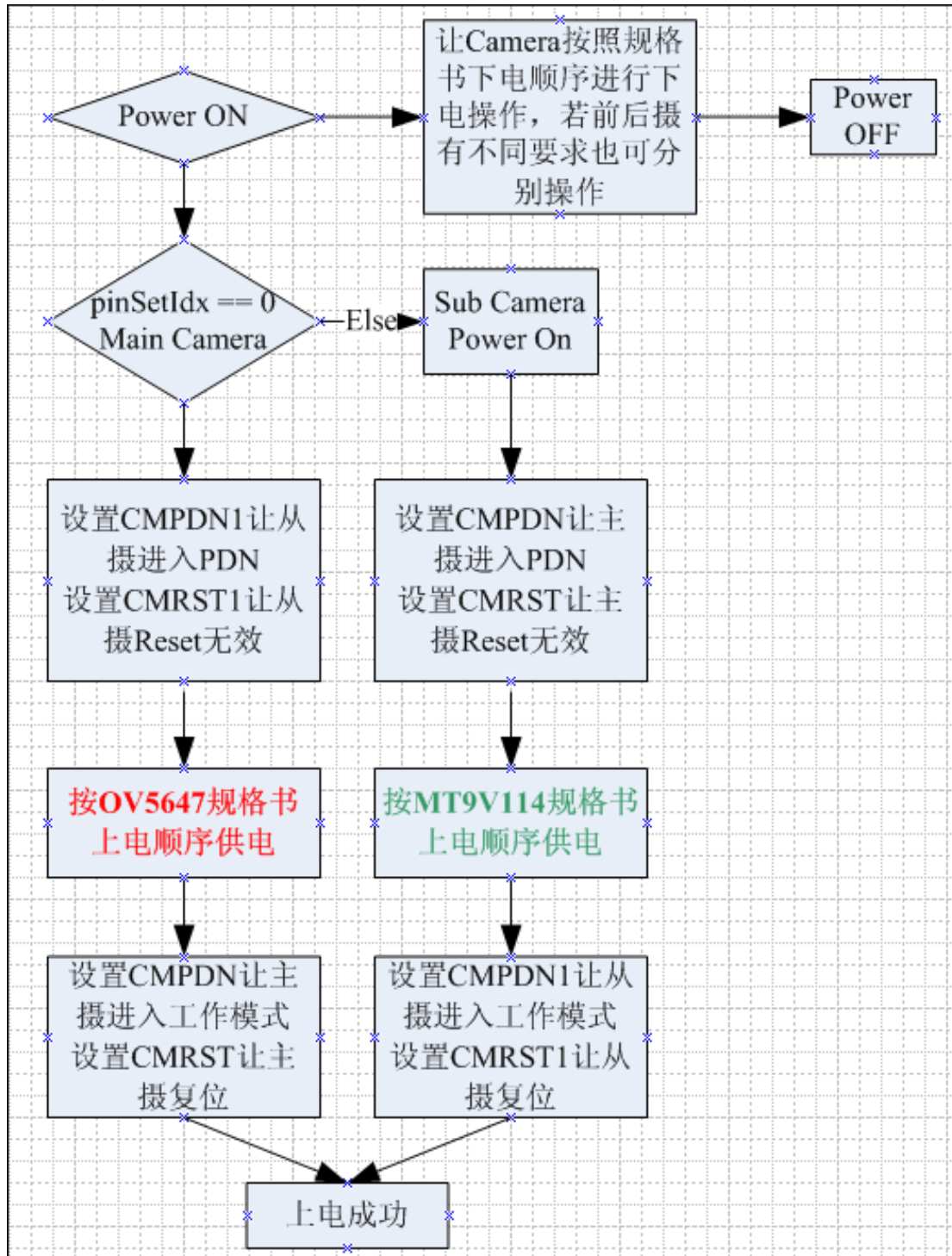
1、修改系统配置文件 ProjectConfig.mk:

-alps\mediatek\config\\${project}\ProjectConfig.mk

2、检查、配置供电文件：

-alps\mediatek\custom\\${project}\Kernel\Camera\Camera\kd_camera_hw.c

Camera 供电流程（以 3M 前摄 MT9V114+5M 后摄 OV5647 为例）：



其实在 kd_camera_hw.c 中只有一个函数 kdCISModulePowerOn ()，在这个函数中需要注意的是通过 GPIO 口控制 PDN 和 RST 引脚的时候，对于其相关的定义，由其切换平台的时候。例如 MT6573 和 MT6575 的定义顺序就不同。

```

u32 pinSetIdx = 0; //default main sensor 默认从MainSensor开始上电

#define IDX_PS_CMRST 0
#define IDX_PS_CMPDN ④

#define IDX_PS_MODE 1
#define IDX_PS_ON 2
#define IDX_PS_OFF ③

u32 pinSet[3][8] = {
    //for main sensor
    {GPIO_CAMERA_CMRST_PIN,
      GPIO_CAMERA_CMRST_PIN_M_GPIO, /* mode */
      GPIO_OUT_ONE, /* ON state */
      GPIO_OUT_ZERO, /* OFF state */
      GPIO_CAMERA_CMPDN_PIN,
      GPIO_CAMERA_CMPDN_PIN_M_GPIO,
      GPIO_OUT_ZERO,
      GPIO_OUT_ONE,
    },
    //for sub sensor
    {GPIO_CAMERA_CMRST1_PIN,
      GPIO_CAMERA_CMRST1_PIN_M_GPIO,
      GPIO_OUT_ONE,
      GPIO_OUT_ZERO,
      GPIO_CAMERA_CMPDN1_PIN,
      GPIO_CAMERA_CMPDN1_PIN_M_GPIO,
      GPIO_OUT_ONE,
      GPIO_OUT_ZERO,
    },
    //for main 2 sensor
    {GPIO_CAMERA_2_CMRST_PIN,
      GPIO_CAMERA_2_CMRST_PIN_M_GPIO,
      GPIO_OUT_ZERO,
      GPIO_OUT_ONE,
      GPIO_CAMERA_2_CMPDN_PIN_M_GPIO,
      GPIO_OUT_ZERO,
      GPIO_OUT_ONE,
      GPIO_OUT_ZERO,
    },
};

if(mt_set_gpio_mode(pinSet[1-pinSetIdx][IDX_PS_CMPDN],pinSet[1-pinSetIdx][IDX_PS_CMPDN+IDX_PS_MODE]))
if(mt_set_gpio_dir(pinSet[1-pinSetIdx][IDX_PS_CMPDN],GPIO_DIR_OUT))
if(mt_set_gpio_out(pinSet[1-pinSetIdx][IDX_PS_CMPDN],pinSet[1-pinSetIdx][IDX_PS_CMPDN+IDX_PS_OFF]))

```

若想打开相机默认为前摄
交换0、1即可

用作后摄兼容或双后摄供电

数组是从0开始数，所以第七位是置1

所以在主Sensor的PDN脚置1，根据OV5647的数据手册，PDN置1是进入standby模式休眠

根据它前面的if条件判断，这里pinSetIdx=1，所以1-pinSetIdx就是对MainSensor PDN的操作

4+3=7
所以看数组中对MainSensor定义的第七个元素

3、添加 Camera 驱动（以 ov5647 为例）：

创建 **SensorFuncOV5647** 这样一个数据结构

SENSOR_FUNCTION_STRUCT **SensorFuncOV5647**=

```

{
    OV5647Open,
    OV5647GetInfo,
    OV5647GetResolution,
    OV5647FeatureControl,
    OV5647Control,
    OV5647Close};

```


a) OV5647Open

```

UINT32 OV5647Open(void)
{
    kal_uint16 sensor_id=0;
    sensor_id=((OV5647_read_cmos_sensor(0x300A) << 8) | OV5647_read_cmos_sensor(0x300B));
    if (sensor_id != OV5647_SENSOR_ID) {
        return ERROR_SENSOR_CONNECT_FAIL;
    }
    if (ID匹配, 则调用初始化函数) {
        OV5647_Sensor_Init();
    }
    return ERROR_NONE;
}
/* OV5647Open */

```

读取ID
根据OV5647DataSheet

在kd_imgsensor.h中预设好的

如果ID匹配, 则调用初始化函数

0x300A, 0x300B, 0x5634, 0x7673, 0x5647

初始化操作就是对 SensorIC 中寄存器的操作，调试主要由 IC 原厂支持。Open 函数结束后返回 ERROR_NONE 表示初始化成功，可以正常使用。

b) OV5647GetInfo

```

UINT32 OV5647GetInfo(MSDK_SCENARIO_ID_ENUM ScenarioId,
MSDK_SENSOR_INFO_STRUCT *pSensorInfo,
MSDK_SENSOR_CONFIG_STRUCT *pSensorConfigData)

```

第一个参数 **ScenarioId** 来自于 MSDK_SCENARIO_ID_ENUM 这个数组，在 kd_imgsensor_define.h 中是这样定义的：

```

#define MSDK_SCENARIO_ID_ENUM          ACDK_SCENARIO_ID_ENUM
typedef enum
{
    ACDK_SCENARIO_ID_CAMERA_PREVIEW=0,
    ACDK_SCENARIO_ID_VIDEO_PREVIEW,
    ACDK_SCENARIO_ID_VIDEO_CAPTURE_MPEG4,
    ACDK_SCENARIO_ID_CAMERA_CAPTURE_JPEG,
    ACDK_SCENARIO_ID_CAMERA_CAPTURE_MEM,
    ACDK_SCENARIO_ID_CAMERA_BURST_CAPTURE_JPEG,
    ACDK_SCENARIO_ID_VIDEO_DECODE_MPEG4,
    ACDK_SCENARIO_ID_VIDEO_DECODE_H263,
    ACDK_SCENARIO_ID_VIDEO_DECODE_H264,
    ACDK_SCENARIO_ID_VIDEO_DECODE_WMV78,
    ACDK_SCENARIO_ID_VIDEO_DECODE_WMV9,
    ACDK_SCENARIO_ID_VIDEO_DECODE_MPEG2,
    ACDK_SCENARIO_ID_IMAGE_YUV2RGB,
    ACDK_SCENARIO_ID_IMAGE_RESIZE,
    ACDK_SCENARIO_ID_IMAGE_ROTATE,

```



```

ACDK_SCENARIO_ID_IMAGE_POST_PROCESS,
ACDK_SCENARIO_ID_JPEG_RESIZE,
ACDK_SCENARIO_ID_JPEG_DECODE,
ACDK_SCENARIO_ID_JPEG_PARSE,
ACDK_SCENARIO_ID_JPEG_ENCODE,
ACDK_SCENARIO_ID_JPEG_ENCODE_THUMBNAIL,
ACDK_SCENARIO_ID_DRIVER_IO_CONTROL,
ACDK_SCENARIO_ID_DO_NOT_CARE,
ACDK_SCENARIO_ID_IMAGE_DSPL_BUFFER_ALLOC,
ACDK_SCENARIO_ID_TV_OUT,
ACDK_SCENARIO_ID_MAX,
ACDK_SCENARIO_ID_VIDOE_ENCODE_WITHOUT_PREVIEW,
ACDK_SCENARIO_ID_CAMERA_CAPTURE_JPEG_BACK_PREVIEW,
ACDK_SCENARIO_ID_VIDEO_DECODE_RV8,
ACDK_SCENARIO_ID_VIDEO_DECODE_RV9,
ACDK_SCENARIO_ID_CAMERA_ZSD,
} ACDK_SCENARIO_ID_ENUM;

```

通过这个数组定义 Camera 的各种模式，并且给他们从 0 开始给一个模拟的 ID，通过这个 **ScenarioID** 来控制 Camera 的工作模式是在拍照、摄像等等。

想要了解 *pSensorInfo 这个指针的内容就得看 MSDK_SENSOR_INFO_STRUCT 的定义

```

#define MSDK_SENSOR_INFO_STRUCT ACDK_SENSOR_INFO_STRUCT
typedef struct
{
    MUINT16 SensorPreviewResolutionX;
    MUINT16 SensorPreviewResolutionY;
    MUINT16 SensorFullResolutionX;
    MUINT16 SensorFullResolutionY;
    MUINT8 SensorClockFreq; /* MHz */
    MUINT8 SensorCameraPreviewFrameRate;
    MUINT8 SensorVideoFrameRate;
    MUINT8 SensorStillCaptureFrameRate;
    MUINT8 SensorWebCamCaptureFrameRate;
    MUINT8 SensorClockPolarity; /*
SENSOR_CLOCK_POLARITY_HIGH/SENSOR_CLOCK_POLARITY_Low */

```

```

MUINT8 SensorClockFallingPolarity;
MUINT8 SensorClockRisingCount;          /* 0..15 */
MUINT8 SensorClockFallingCount;         /* 0..15 */
MUINT8 SensorClockDividCount;           /* 0..15 */
MUINT8 SensorPixelClockCount;           /* 0..15 */
MUINT8 SensorDataLatchCount;            /* 0..15 */
MUINT8 SensorHsyncPolarity;
MUINT8 SensorVsyncPolarity;
MUINT8 SensorInterruptDelayLines;
MINT32  SensorResetActiveHigh;
MUINT32 SensorResetDelayCount;
ACDK_SENSOR_INTERFACE_TYPE_ENUM SensorInterfaceType;
ACDK_SENSOR_OUTPUT_DATA_FORMAT_ENUM SensorOutputDataFormat;
ACDK_SENSOR_MIPI_LANE_NUMBER_ENUM SensorMIPILaneNumber;
CAMERA_ISO_BINNING_INFO_STRUCT  SensorISOBinningInfo;
MUINT32 CaptureDelayFrame;
MUINT32 PreviewDelayFrame;
MUINT32 VideoDelayFrame;
MUINT16 SensorGrabStartX;
MUINT16 SensorGrabStartY;
MUINT16 SensorDrivingCurrent;
MUINT8  SensorMasterClockSwitch;
MUINT8  AEShutDelayFrame;                /* The frame of setting shutter default 0 for TG
int */
MUINT8  AESensorGainDelayFrame;         /* The frame of setting sensor gain */
MUINT8  AEISPGainDelayFrame;
MUINT8  MIPIDataLowPwr2HighSpeedTermDelayCount;
MUINT8  MIPIDataLowPwr2HighSpeedSettleDelayCount;
MUINT8  MIPICLKLowPwr2HighSpeedTermDelayCount;
MUINT8  SensorWidthSampling;
MUINT8  SensorHightSampling;
MUINT8  SensorPacketECCOrder;
MUINT8  SensorDriver3D;
} ACDK_SENSOR_INFO_STRUCT, *PACDK_SENSOR_INFO_STRUCT;

```

这个结构体列取了 Sensor 的时钟频率、预览时的帧率、行同步/帧同步频率等参数。

第三个参数*pSensorConfigData 同样根据 MSDK_SENSOR_CONFIG_STRUCT 结构体

```
#define      MSDK_SENSOR_CONFIG_STRUCT    ACDK_SENSOR_CONFIG_STRUCT

typedef struct
{
    ACDK_SENSOR_IMAGE_MIRROR_ENUM    SensorImageMirror;
    MINT32  EnableShutterTansfer;      /* Capture 时的快门设置 */
    MINT32  EnableFlashlightTansfer;   /*有闪光灯的 SensorCapture 时的快门设置*/
    ACDK_SENSOR_OPERATION_MODE_ENUM    SensorOperationMode;
    MUINT16  ImageTargetWidth;         /* Capture 的图像宽度 */
    MUINT16  ImageTargetHeight;        /* Capture 的图像高度*/
    MUINT16  CaptureShutter;           /* Capture 时的快门设置 */
    MUINT16  FlashlightDuty;           /*有闪光灯的 SensorCapture 时的快门设置*/
    MUINT16  FlashlightOffset;         /*有闪光灯的 SensorCapture 时的快门设置*/
    MUINT16  FlashlightShutFactor;     /*有闪光灯的 SensorCapture 时的快门设置*/
    MUINT16  FlashlightMinShutter;     /*有闪光灯的 SensorCapture 时的快门设置*/
    ACDK_CAMERA_OPERATION_MODE_ENUM    MetaMode;
    MUINT32  DefaultPclk;              // Sensor 默认的像素时钟频率(Ex:24000000)
    MUINT32  Pixels;
    MUINT32  Lines;
    MUINT32  Shutter;
    MUINT32  FrameLines;
} ACDK_SENSOR_CONFIG_STRUCT;
```

c) OV5647GetResolution

UINT32 OV5647GetResolution(MSDK_SENSOR_RESOLUTION_INFO_STRUCT

*pSensorResolution) 此函数只有一个参数*pSensorResolution,

找到结构体: MSDK_SENSOR_RESOLUTION_INFO_STRUCT

```
#define      MSDK_SENSOR_RESOLUTION_INFO_STRUCT
ACDK_SENSOR_RESOLUTION_INFO_STRUCT
```

typedef struct

```
{
    MUINT16 SensorPreviewWidth; //预览时的图像宽度
    MUINT16 SensorPreviewHeight; //预览时的图像高度
```

```

    MUINT16 SensorFullWidth;
    MUINT16 SensorFullHeight;
}
ACDK_SENSOR_RESOLUTION_INFO_STRUCT,
*PACDK_SENSOR_RESOLUTION_INFO_STRUCT;

```

到这里可以发现同样是获取 **Sensor** 的信息, **GetInfo** 函数获取并设置了 **Sensor** 所处的模式、设置好需要的各种时钟、快门和拍照获取图像的信息整个流程的参数, 而 **GetResolution** 函数却是设置图像在预览模式下的参数, 实际上是通过实际捕获的图像缩放来提高预览时图像的解析度。

d) OV5647FeatureControl

```

UINT32 OV5647FeatureControl(MSDK_SENSOR_FEATURE_ENUM FeatureId,
UINT8 *pFeaturePara, UINT32 *pFeatureParaLen)

```

在 FeatureControl 这个函数中有三个参数:

```

#define MSDK_SENSOR_FEATURE_ENUM
ACDK_SENSOR_FEATURE_ENUM
typedef enum
{
    SENSOR_FEATURE_BEGIN = SENSOR_FEATURE_START,
    SENSOR_FEATURE_GET_RESOLUTION,
    SENSOR_FEATURE_GET_PERIOD,
    SENSOR_FEATURE_GET_PIXEL_CLOCK_FREQ,
    SENSOR_FEATURE_SET_ESHUTTER,
    SENSOR_FEATURE_SET_NIGHTMODE,
    SENSOR_FEATURE_SET_GAIN,
    SENSOR_FEATURE_SET_FLASHLIGHT,
    SENSOR_FEATURE_SET_ISP_MASTER_CLOCK_FREQ,
    SENSOR_FEATURE_SET_REGISTER,
    SENSOR_FEATURE_GET_REGISTER,
    SENSOR_FEATURE_SET_CCT_REGISTER,
    SENSOR_FEATURE_GET_CCT_REGISTER,
    SENSOR_FEATURE_SET_ENG_REGISTER,
    SENSOR_FEATURE_GET_ENG_REGISTER,
    SENSOR_FEATURE_GET_REGISTER_DEFAULT,
    SENSOR_FEATURE_GET_CONFIG_PARA,
    SENSOR_FEATURE_CAMERA_PARA_TO_SENSOR,

```

```
SENSOR_FEATURE_SENSOR_TO_CAMERA_PARA,  
SENSOR_FEATURE_GET_GROUP_COUNT,  
SENSOR_FEATURE_GET_GROUP_INFO,  
SENSOR_FEATURE_GET_ITEM_INFO,  
SENSOR_FEATURE_SET_ITEM_INFO,  
SENSOR_FEATURE_GET_ENG_INFO,  
SENSOR_FEATURE_GET_LENS_DRIVER_ID,  
SENSOR_FEATURE_SET_YUV_CMD,  
SENSOR_FEATURE_SET_VIDEO_MODE,  
SENSOR_FEATURE_SET_CALIBRATION_DATA,  
SENSOR_FEATURE_SET_SENSOR_SYNC,  
SENSOR_FEATURE_INITIALIZE_AF,  
SENSOR_FEATURE_CONSTANT_AF,  
SENSOR_FEATURE_MOVE_FOCUS_LENS,  
SENSOR_FEATURE_GET_AF_STATUS,  
SENSOR_FEATURE_GET_AF_INF,  
SENSOR_FEATURE_GET_AF_MACRO,  
SENSOR_FEATURE_CHECK_SENSOR_ID,  
SENSOR_FEATURE_SET_AUTO_FLICKER_MODE,  
SENSOR_FEATURE_SET_TEST_PATTERN,  
SENSOR_FEATURE_SET_SOFTWARE_PWDN,  
SENSOR_FEATURE_SINGLE_FOCUS_MODE,  
SENSOR_FEATURE_CANCEL_AF,  
SENSOR_FEATURE_SET_AF_WINDOW,  
SENSOR_FEATURE_GET_EV_AWB_REF,  
SENSOR_FEATURE_GET_SHUTTER_GAIN_AWB_GAIN,  
SENSOR_FEATURE_MAX  
} ACDK_SENSOR_FEATURE_ENUM;
```

FeatureId 这个参数提供了低层给上层接口的准备。***pFeaturePara** 和***pFeatureParaLen** 分别是 **FeatureId** 的具体值。