ISP 3A 开发指南

文档版本号：V1.0

发布日期：2019 年 03 月 30 日

修改记录

版本号	日期	内容描述
V1.0	2019-03-30	建立 V316 初始版本

目录

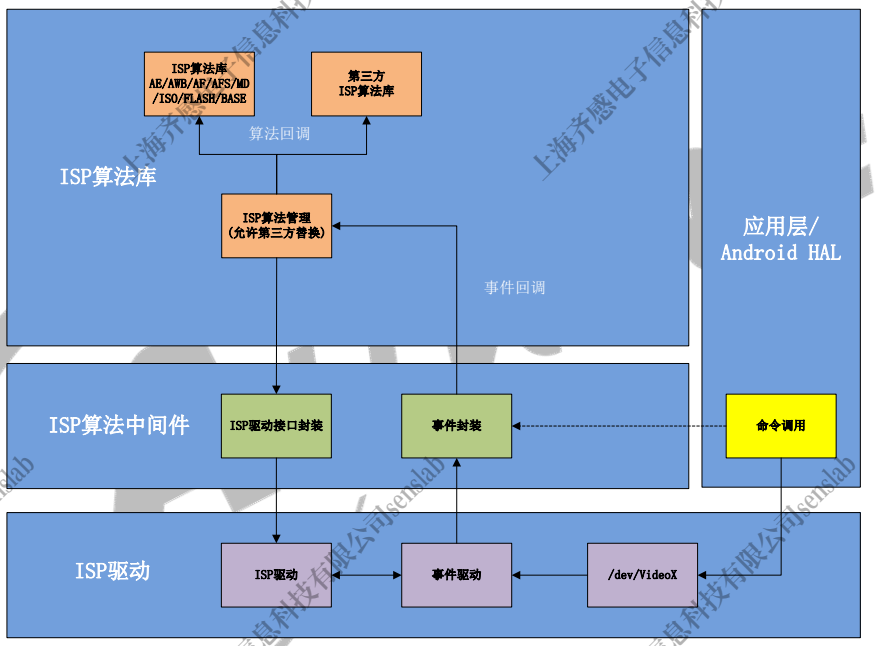
1. 概述

1.1. ISP server 简介

ISP Server 模块主要包括 ISP 算法库和 ISP 中间件部分：

1. ISP 算法库部分，其主要用于在 ISP 运行时图像效果的处理，包括 3A 算法以及一系列 ISP 正常运行所需的基本算法；
2. ISP 中间件部分，其主要用于控制 ISP 以及 Sensor 驱动、响应 Camera 应用以及 Tuning 工具命令、调度 ISP 相关算法等，包括事件管理、Pipeline 管理、Buffer 管理以及算法调度等模块。

ISP 算法库、中间件、驱动以及 Camera 应用相互关系如下图所示：



ISP Server 基本框架

1.2. ISP server 代码结构

ISP Server 开源部分代码结构如下：

```
Isp_server:
│   isp.c                    ;ISP 对外接口实现（对接调试工具、Camera 应用）
│   isp.h                    ;ISP 对外接口头文件
│   Makefile
├─include
│   ├─device                 ; ISP 设备头文件
```

```
│           isp_dev.h
│           video.h
│   ├─V4l2Camera                ; V4L2 头文件
│           linux/
│           sunxi_camera_v2.h
│       isp_3a_ae.h             ; 自动曝光算法头文件
│       isp_3a_af.h             ; 自动对焦算法头文件
│       isp_3a_awb.h            ; 自动白平衡算法头文件
│       isp_3a_afs.h            ; 自动工频检测算法头文件
│       isp_tuning.h            ; ISP 效果 tuning 接口头文件
│       isp_base.h              ; 基本算法头文件
│       isp_ini_parse.h         ; ini 文件解析头文件
│       isp_iso_config.h        ; 动态参数设置头文件
│       isp_cmd_intf.h          ; 命令处理头文件
│       isp_comm.h              ; 公共头文件
│       isp_debug.h             ; Debug 头文件
│       isp_manage.h            ; 算法管理模块头文件
│       isp_module_cfg.h        ; 硬件模块头文件
│       isp_rolloff.h           ; 镜头阴影矫正算法头文件
│       isp_tone_mapping.h      ; 色调映射算法头文件
│       isp_type.h              ; 类型定义头文件
│
├─iniparser
│   ├─src
│           iniparser.c
│           iniparser.h
│           dictionary.c
│           dictionary.h
│
```

```
├─isp_cfg
│       isp_ini_parse.c                ; Sensor 模组 ISP 配置文件解析
│   ├─SENSOR_H                         ; Sensor 模组 ISP 配置头文件
│           isp_default_ini_4v5.h       ; 默认 ISP 配置
│           imx290_default_ini_4v5.h    ; imx290 ISP 配置
│           imx317_default_ini_4v5.h    ; imx317 ISP 配置
│           ar0238_default_ini_4v5.h    ; ar0238 ISP 配置
│           ov2718_wdr_ini_4v5.h        ; ov2718 wdr ISP 配置
│           Makefile
│
├─isp_dev
│       video.c                        ; 用于管理标准 v4l2 视频设备
│       isp_dev.c                      ; ISP 设备封装，用于管理相关设备
│       isp_v4l2_helper.c              ;V4l2 帮助函数
│       isp_v4l2_helper.h              ;V4l2 帮助函数头文件
│       media.c                        ;media 框架帮助函数
│       media.h                        ;media 框架帮助函数头文件
│       tools.h                        ;工具文件
│       Makefile
│
├─isp_events
│       events.c                       ;事件管理模块，用于监听分发驱动事件
│       events.h                       ;事件管理模块头文件
│
├─isp_manage
│       isp_manage.c                   ;ISP 算法管理模块
│       isp_helper.c                   ;ISP 调试和命令接口帮助函数
├─isp_manage
│       isp_math_util.c                ;ISP 数学运算帮助函数
```

```
│       isp_ math_util.h
│       isp_matrix.c                ;ISP 矩阵运算帮助函数
│       isp_ matrix.h
│
└─isp_tuning
│       isp_tuning.c                ;效果调试接口
│       isp_tuning_priv.h           ;效果调试私有头文件
│
└─out
        libisp_ae.a                 ;自动曝光算法库
        libisp_af.a                 ;自动对焦算法库
        libisp_afs.a                ;自动去工频算法库
        libisp_awb.a                ;自动白平衡算法库
        libisp_base.a               ;基础算法库
        libisp_ini.a                ;ISP 配置参数获取库
        libisp_iso.a                ;动态参数设置算法库
        libisp_md.a                 ;运动检测算法库
        libisp_gtm.a                ;全局色调映射算法库
        libisp_pltm.a               ;局部色调映射算法库
        libisp_rolloff.a            ;自动 color shading 矫正算法库
```

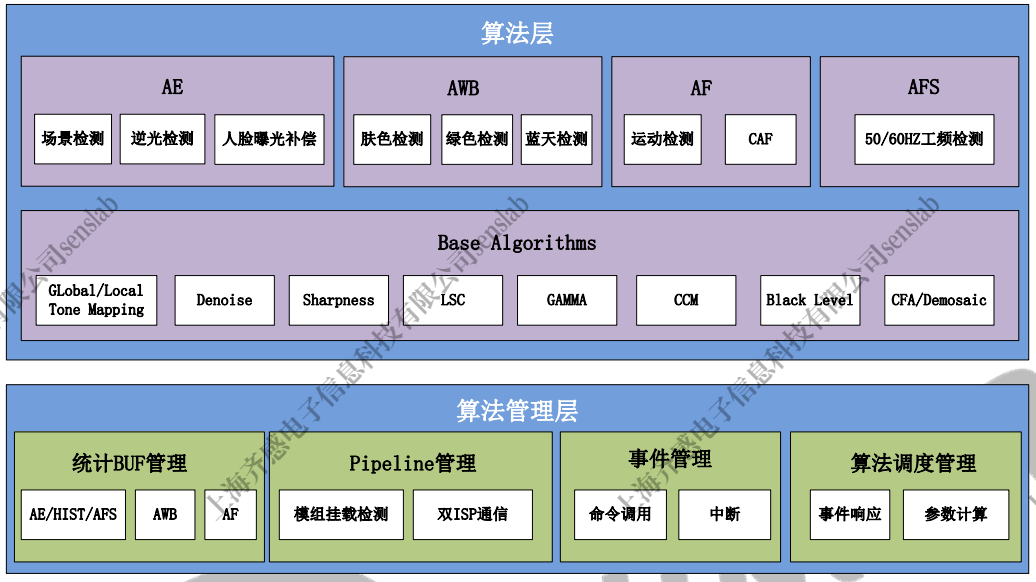
ISP Server 开源代码可以概括为 3 部分：

1. 设备管理部分，主要包括 isp_dev、isp_events 目录下的代码，其中 isp_dev 部分负责统一管理视频设备、video 设备、Sensor 设备、统计设备；
   a) 在初始化时，其会建立好相应的 Sensor->CSI->ISP ->Video 通路，对只需要图像数据的设备来说，仅需操作 Video 设备即可获取想要的数据。
   b) 在设备运行时，isp_events 模块会根据 CSI 或者 ISP 返回来的事件来通知不同模块进行必要的事件处理，主要事件一般有 isp stream off 事件、S_CTRL 命令事件以及统计值 Ready 事件等。
2. Sensor 配置管理部分，主要包括 iniparser、isp_cfg 以及 isp_tuning 部分；其中 iniparser 为标准 ini 文件解析库；isp_cfg 为 sensor 配置文件匹配部分，有两种方法获取配置文件：a. 读取 ini 文件，

b. 头文件预定义；isp_tuning 为外部调整 ISP 效果提供接口。

3. 算法管理部分，主要包括 isp_manage、isp_math 部分，其负责各个子算法的初始化、运行、退出等动作。

ISP Server 的构成结构如下图所示：



ISP 算法库的构成

1.3. ISP 3A 开发方法

ISP Server 框架可以为客户提供三种开发模式：

1. 极简模式，使用全志提供的全套算法库，ISP Server 部分对客户不可见，客户只需要通过 MPI 操作相应视频设备节点即可获取图像数据，图像效果完全由 Tuning 工具给出的配置控制。

2. 一般模式，使用全志提供的全套算法库，ISP Server 部分对客户不可见，客户可以通过 MPI 操作相应视频设备节点获取图像数据，同时可以操作 ISP、Sensor 设备，通过禁用全志 ISP 内部某些特定算法，然后通过 MPI 接口获取统计信息，再通过 MPI 接口设置给 ISP，可达到替换某些特定算法的目的。

3. 高级模式，对于开发能力强的核心客户，可以部分替换 ISP Server 中的算法库，ISP Server 开源部分可以开放给这类客户。

2. 使用指南

2.1. ISP 使用流程

ISP Server 需要与 VI 采集协同工作，使用时应当先初始化 VI 采集相关配置，在初始化 ISP 相关配置，打开 VI 采集流之后运行 ISP run 即可，ISP Server 此时会动态调整 Sensor、Lens 以及 ISP 相关配置。

ISP Server 使用方法非常简单，示例代码如下：

```
int main(int argc __attribute__((__unused__)), char *argv[] __attribute__((__unused__)))
{
    media_dev_init();       //初始化多媒体设备
    isp_init(0);            //初始化 isp0
    isp_run(0);             //运行 isp0 线程
    isp_pthread_join(0);    //等待 isp0 线程结束
    isp_exit(0);            //退出 isp0
    media_dev_exit();       //退出多媒体设备
    return 0;
}
```

2.2. Sensor 设备操作

Sensor 与 ISP 的对应关系在一般情况下可以由内核中的 Device tree 配置，执行 media_dev_init()时，ISP Server 中的设备管理模块便可获取其相对应关系，应用操作 Sensor 无需直接找到对应 Sensor 设备，只需要操作 ISP 接口即可，如：

```
/*isp_dev.h  中定义的 Sensor 相关的操作集*/
int isp_sensor_get_configs(struct hw_isp_device *isp, struct sensor_config *cfg);
int isp_sensor_set_exp_gain(struct hw_isp_device *isp, struct sensor_exp_gain *exp_gain);
```

2.3. 3A 算法库注册

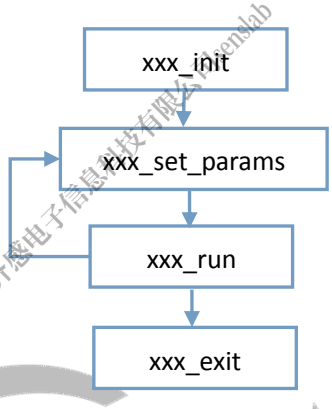
所有 ISP 软件算法都有一组统一的注册接口，如 AE 算法：

```
/*isp_3a_ae.h  中定义的 AE 算法相关的操作集*/
typedef struct isp_ae_core_ops {
    HW_S32 (*isp_ae_set_params)(void * ae_core_obj, ae_param_t *param,  ae_result_t *result);
```

```
    HW_S32 (*isp_ae_get_params)(void *ae_core_obj, ae_param_t *param);
    HW_S32 (*isp_ae_run)(void *ae_core_obj, ae_stats_t *stats, ae_result_t *result);
} isp_ae_core_ops_t;


void* ae_init(isp_ae_core_ops_t **ae_core_ops);
void ae_exit(void *ae_core_obj);
```

运行算法的通用基本流程为：



例如 AE 算法：

```
/****************AE init**************/
void __isp_ae_init(struct isp_lib_context *isp_gen)
{
    isp_gen->ae_entity_ctx.ae_entity = ae_init(&isp_gen->ae_entity_ctx.ops);
    if (isp_gen->ae_entity_ctx.ae_entity == NULL || NULL == isp_gen->ae_entity_ctx.ops) {
        ISP_ERR("AE Entity is BUSY or NULL!\n");
        return -1;
    } else {
        clear(isp_gen->ae_entity_ctx.ae_param);
        isp_gen->ae_entity_ctx.ae_param.u.isp_platform_id = isp_gen->module_cfg.isp_platform_id;
        isp_ae_set_params_helper(ISP_AE_PLATFORM_ID);
    }
}

/****************AE set_params**************/
```

```
void __isp_ae_set_params(struct isp_lib_context *isp_gen)
{
        clear(isp_gen->ae_entity_ctx.ae_param);
        isp_gen->ae_entity_ctx.ae_param.u.ae_frame_id = isp_gen->ae_frame_cnt;
        isp_ae_set_params_helper(ISP_AE_FRAME_ID);

        clear(isp_gen->ae_entity_ctx.ae_param);
        isp_gen->ae_entity_ctx.ae_param.u.ae_setting = isp_gen->ae_settings;
        isp_ae_set_params_helper(ISP_AE_SETTINGS);

        //ae_sensor_info.
        clear(isp_gen->ae_entity_ctx.ae_param);
        isp_gen->ae_entity_ctx.ae_param.u.ae_sensor_info = isp_gen->sensor_info;
        isp_ae_set_params_helper(ISP_AE_SENSOR_INFO);
}

/****************AE exit**************/
void __isp_ae_exit(struct isp_lib_context *isp_gen)
{
        ae_exit(isp_gen->ae_entity_ctx.ae_entity);
}
```

3. 3A 开发指南

3.1. 概述

对于高阶开发者，可以使用自己开发的算法替换 ISP out 目录下相应的算法模块，主要涉及 AE、AWB 以及 AF 部分，ISP Server 中每个独立算法的形式基本一致，因此可以提出一个固定模板来对接具体的算法，通用模板格式如下：

```
/*
 ******************************************************************************
*/
#include "../include/isp_manage.h"
#include "../include/isp_3a_xxx.h"
#include "../isp_math/isp_math_util.h"
typedef struct isp_xxx_config_entity
{
}xxx_config_t;

typedef struct isp_xxx_core_entity
{
}xxx_core_entity_t;

int    __IspxxxIsr(xxx_core_entity_t *entity, xxx_stats_t *stats, xxx_result_t *result)
{
        return 0;
}

xxx_core_entity_t *__IspAllocxxxEntity(void)
{
        xxx_core_entity_t *entity = NULL;
        entity = malloc(sizeof(*entity));
```

```
    if(entity == NULL) {
        ISP_ERR("xxx Entity No memory!");
        return NULL;
    }
    memset(entity, 0, sizeof(*entity));
    entity->busy_flag = 1;
    return entity;
}

void __IspFreexxxEntity(xxx_core_entity_t *xxx_core_obj)
{
    if(xxx_core_obj) {
        xxx_core_obj->busy_flag = 0;
        free(xxx_core_obj);
    }
}

#define ISP_xxx_SET_PARAMS(key)\
{\
    entity->config.key = param->u.key;\
}
#define ISP_xxx_GET_PARAMS(key)\
{\
    param->u.key = entity->config.key;\
}

int __AwxxxSetParams(void * xxx_core_obj, xxx_param_t *param, xxx_result_t *result)
{
    int ret = 0;
```

```
    xxx_core_entity_t *entity;


    if(xxx_core_obj && param)
    {
        entity = (xxx_core_entity_t *)xxx_core_obj;
    }
    else
    {
        ret = -1;
        goto set_param_end;
    }

    switch (param->type)
    {
        case ISP_xxx_PLATFORM_ID:
            ISP_xxx_SET_PARAMS(isp_platform_id);
            break;
        case ISP_xxx_FRAME_ID:
            ISP_xxx_SET_PARAMS(xxx_frame_id);
            break;
        default:
            ret = -1;
    }


set_param_end:
    return ret;
}


int __AwxxxGetParams(void *xxx_core_obj, xxx_param_t *param)
```

```
{
    int ret = 0;
    xxx_core_entity_t *entity;
    if(xxx_core_obj && param)
    {
        entity = (xxx_core_entity_t *)xxx_core_obj;
    }
    else
    {
        ret = -1;
        goto get_param_end;
    }
    ISP_LIB_LOG(ISP_LOG_AF, "aw_af_get_params param->type = %d\n", param->type);
    switch (param->type)
    {
        case ISP_xxx_PLATFORM_ID:
            ISP_xxx_GET_PARAMS(isp_platform_id);
            break;
        case ISP_xxx_FRAME_ID:
            ISP_xxx_GET_PARAMS(xxx_frame_id);
            break;
        default:
            ret = -1;
    }

get_param_end:
    return ret;
}
```

```
int __AwxxxRun(void *xxx_core_obj, xxx_stats_t *stats, xxx_result_t *result)
{
        int ret = 0;
        xxx_core_entity_t *entity;
        if(xxx_core_obj &&stats && result)
        {
                entity = (xxx_core_entity_t *)xxx_core_obj;
        }
        ret = __IspxxxIsr(entity, stats, result);
        return ret;
}

static isp_xxx_core_ops_t AwxxxOps =
{
        .isp_xxx_set_params = __AwxxxSetParams,
        .isp_xxx_get_params = __AwxxxGetParams,
        .isp_xxx_run = __AwxxxRun,
};

void __xxxInitEntity(xxx_core_entity_t *entity)
{
        entity->xxx_detect_flicker_type = 0xff;
        entity->xxx_stat_cnt = 0;
        entity->xxx_weight[0] = 1;
        entity->xxx_weight[1] = -1;
        entity->xxx_weight[2] = 0;
        return;
}
```

```
void* xxx_init(isp_xxx_core_ops_t **xxx_core_ops)
{
        xxx_core_entity_t *entity;
        entity = __IspAllocxxxEntity();
        if(entity)
        {
                __xxxInitEntity(entity);
                *xxx_core_ops = &AwxxxOps;
                return (void *)entity;
        }
        ISP_ERR("xxx_init Error!\n");
        return NULL;
}

void    xxx_exit(void *xxx_core_obj)
{
        __IspFreexxxEntity((xxx_core_entity_t *)xxx_core_obj);
}
```

3.2. AE 算法开发

算法管理单元通过 ae_init 函数初始化 AE 算法结构体，并返回 isp_ae_core_ops_t 操作集合：

```
/*isp_3a_ae.h  中定义的 AE 算法相关的操作集*/
typedef struct isp_ae_core_ops {
        HW_S32 (*isp_ae_set_params)(void * ae_core_obj, ae_param_t *param,   ae_result_t *result);
        HW_S32 (*isp_ae_get_params)(void *ae_core_obj, ae_param_t *param);
        HW_S32 (*isp_ae_run)(void *ae_core_obj, ae_stats_t *stats, ae_result_t *result);
} isp_ae_core_ops_t;

void* ae_init(isp_ae_core_ops_t **ae_core_ops);
void    ae_exit(void *ae_core_obj);
```

通过该操作集合，可以控制 AE 参数，调度 AE 算法，具体流程详见 2.3 节 3A 算法库注册部分，接口中用到主要结构体描述如下：

isp_ae_stats_s

●PROTOTYPE

```
struct isp_ae_stats_s {
    HW_U32 win_pix_n;
    HW_U32 accum_r[ISP_AE_ROW][ISP_AE_COL];
    HW_U32 accum_g[ISP_AE_ROW][ISP_AE_COL];
    HW_U32 accum_b[ISP_AE_ROW][ISP_AE_COL];
    HW_U32 avg[ISP_AE_ROW*ISP_AE_COL];

    HW_U32 hist[ISP_HIST_NUM];
};
```

●MEMBERS

win_pix_n	：	每个窗口像素数。
accum_r	：	分区域 R 累加和。
accum_g	：	分区域 G 累加和。
accum_b	：	分区域 B 累加和。
avg	：	分区域亮度平均值。
hist	：	直方图统计值。

●DESCRIPTION

isp_ae_stats_s 为用于描述 AE 统计值的一个结构体。

ae_param_t

●PROTOTYPE

```
typedef struct isp_ae_param {
    ae_param_type_t type;
```

```
    HW_U32 current_frame_cnt;
    union {
        HW_S32 isp_platform_id;
        HW_S32 ae_frame_id;
        ae_ini_cfg_t ae_ini;
        isp_ae_settings_t    ae_setting;
        HW_S32 ae_pline_index;
        HW_S32 sensor_update_done;
        struct isp_h3a_coor_win ae_coor;
        isp_sensor_info_t ae_sensor_info;
        ae_test_config_t test_cfg;
    } u;
} ae_param_t;
```

●MEMBERS

type ：ISP AE 命令类型，其每种类型与联合体 u 中参数一一对应，定义如下:

```
typedef enum isp_ae_param_type {
    ISP_AE_PLATFORM_ID,
    ISP_AE_FRAME_ID,
    ISP_AE_INI_DATA,

    ISP_AE_SETTINGS,
    ISP_AE_HDR_SETTINGS,
    ISP_AE_UPDATE_AE_TABLE,

    ISP_AE_SET_EXP_IDX,
    ISP_AE_BUILD_TOUCH_WEIGHT,
    ISP_AE_SENSOR_INFO,
    ISP_AE_TEST_CONFIG,

    ISP_AE_PARAM_TYPE_MAX,
```

```
} ae_param_type_t;
```

isp_platform_id	：	ISP 平台 ID。
ae_frame_id	：	AE 算法执行计数。
ae_ini	：	AE 算法初始化 INI 配置。
ae_setting	：	AE 算法运行时配置，包括曝光补偿、手动曝光，场景模式等。
ae_pline_index	：	手动设置 Pline 索引。
sensor_update_done	：	Sensor 更新曝光设定完成。
ae_coor	：	AE ROI 窗口坐标。
ae_sensor_info	：	Sensor 信息，包括 VTS，HTS 以及输出宽高等。
test_cfg	：	AE 测试配置。

●DESCRIPTION

ae_param_t 为用于描述 AE 命令参数的一个结构体。

isp_ae_result

●PROTOTYPE

```
typedef struct isp_ae_result {
    enum ae_status ae_status;
    sensor_setting_t sensor_set;
    sensor_setting_t sensor_set_short;
    HW_S32 BrightPixellValue;
    HW_S32 DarkPixelValue;

    HW_U32 ae_gain;
    HW_S32 ae_target;
    HW_S32 ae_avg_lum;
    HW_S32 ae_weight_lum;
    HW_S32 ae_wdr_ratio;
```

```
    HW_S32 wdr_hi_th;
    HW_S32 wdr_low_th;


    HW_U8 backlight;
} ae_result_t;
```

●MEMBERS

ae_status	：	AE 状态。
sensor_set	：	Sensor 曝光设置。
sensor_set_short	：	Sensor 短曝光设置。
BrightPixellValue	：	AE 亮像素参考值。
DarkPixelValue	：	AE 暗像素参考值。
ae_gain	：	AE 调整变化率。
ae_target	：	AE 目标亮度。
ae_avg_lum	：	AE 平均亮度。
ae_weight_lum	：	AE 加权亮度。
ae_wdr_ratio	：	AE WDR 比率。
wdr_hi_th	：	AE WDR 高阈值。
wdr_low_th	：	AE WDR 低阈值。
backlight	：	AE 背光程度。

●DESCRIPTION

ae_result_t 为用于描述 AE 输出结果的一个结构体。

3.3. AWB 算法开发

算法管理单元通过 aw_init 函数初始化 AWB 算法结构体，并返回 isp_awb_core_ops_t 操作集合：

```
/*isp_3a_awb.h 中定义的 AWB 算法相关的操作集*/
typedef struct isp_awb_core_ops {
    HW_S32 (*isp_awb_set_params)(void * awb_core_obj, awb_param_t *param, awb_result_t *result);
    HW_S32 (*isp_awb_get_params)(void *awb_core_obj, awb_param_t *param);
```

```
    HW_S32 (*isp_awb_run)(void *awb_core_obj, awb_stats_t *stats, awb_result_t *result);
} isp_awb_core_ops_t;


void* awb_init(isp_awb_core_ops_t **awb_core_ops);
void    awb_exit(void *awb_core_obj);
```

通过该操作集合，可以控制 AWB 参数，调度 AWB 算法，具体流程详见 2.3 节 3A 算法库注册部分，接口中用到主要结构体描述如下：

isp_awb_stats_s

●PROTOTYPE

```
struct isp_awb_stats_s {
    HW_U32 awb_sum_r[ISP_AWB_ROW][ISP_AWB_COL];
    HW_U32 awb_sum_g[ISP_AWB_ROW][ISP_AWB_COL];
    HW_U32 awb_sum_b[ISP_AWB_ROW][ISP_AWB_COL];
    HW_U32 awb_sum_cnt[ISP_AWB_ROW][ISP_AWB_COL];


    HW_U32 awb_avg_r[ISP_AWB_ROW][ISP_AWB_COL];
    HW_U32 awb_avg_g[ISP_AWB_ROW][ISP_AWB_COL];
    HW_U32 awb_avg_b[ISP_AWB_ROW][ISP_AWB_COL];
    HW_U32 avg[ISP_AWB_ROW][ISP_AWB_COL];
};
```

●MEMBERS

awb_sum_r	：分区域 R 累加和。
awb_sum_g	：分区域 G 累加和。
awb_sum_b	：分区域 B 累加和。
awb_sum_cnt	：分区域有效像素数。
awb_avg_r	：分区域 R 像素平均值（归一化为 0~255）。
awb_avg_g	：分区域 G 像素平均值（归一化为 0~255）。
awb_avg_b	：分区域 B 像素平均值（归一化为 0~255）。

avg : 分区域加权平均值。

●**DESCRIPTION**

isp_awb_stats_s 为用于描述 AWB 统计值的一个结构体。

awb_param_t

●**PROTOTYPE**

```
typedef struct isp_awb_param {
    awb_param_type_t type;
    HW_U32 current_frame_cnt;
    union {
        HW_S32 isp_platform_id;
        HW_S32 awb_frame_id;
        isp_awb_setting_t awb_ctrl;
        awb_ini_cfg_t awb_ini;
        isp_sensor_info_t awb_sensor_info;
        awb_test_config_t test_cfg;
    } u;
} awb_param_t;
```

●**MEMBERS**

type : ISP AWB 命令类型，其每种类型与联合体 u 中参数一一对应，定义如下:

```
typedef enum isp_awb_param_type {
    ISP_AWB_PLATFORM_ID,
    ISP_AWB_FRAME_ID,
    ISP_AWB_CTRL_CFG,

    ISP_AWB_INI_DATA,
    ISP_AWB_SENSOR_INFO,
    ISP_AWB_TEST_CONFIG,
```

```
    ISP_AWB_PARAM_TYPE_MAX,
} awb_param_type_t;
```

isp_platform_id	：	ISP 平台 ID。
awb_frame_id	：	AWB 算法执行计数。
awb_ctrl	：	AWB 算法运行时配置。
awb_ini	：	AWB 算法初始化 INI 配置。
awb_sensor_info	：	Sensor 信息，包括 VTS，HTS 以及输出宽高等。
test_cfg	：	AWB 测试配置。

●DESCRIPTION

awb_param_t 为用于描述 AWB 命令参数的一个结构体。

awb_result_t

●PROTOTYPE

```
typedef struct isp_awb_result {
    struct isp_wb_gain wb_gain_output;
    HW_S32 color_temp_output;
} awb_result_t;
```

●MEMBERS

wb_gain_output	：	白平衡输出增益。
color_temp_output	：	色温输出参考值。

●DESCRIPTION

awb_result_t 为用于描述 AWB 输出结果的一个结构体。

3.4. AF 算法开发

算法管理单元通过 af_init 函数初始化 AF 算法结构体，并返回 isp_af_core_ops_t 操作集合：

```
/*isp_3a_af.h 中定义的 AF 算法相关的操作集*/
typedef struct isp_af_core_ops {
```

```
    HW_S32 (*isp_af_set_params)(void * af_core_obj, af_param_t *param, af_result_t *result);
    HW_S32 (*isp_af_get_params)(void *af_core_obj, af_param_t *param);
    HW_S32 (*isp_af_run)(void *af_core_obj, af_stats_t *stats, af_result_t *result);
} isp_af_core_ops_t;


void* af_init(isp_af_core_ops_t **af_core_ops);
void    af_exit(void *af_core_obj);
```

通过该操作集合，可以控制 AF 参数，调度 AF 算法，具体流程详见 2.3 节 3A 算法库注册部分，接口中用到主要结构体描述如下：

isp_af_stats_s

●PROTOTYPE

```
struct isp_af_stats_s {
    HW_U32 af_count[ISP_AF_ROW][ISP_AF_COL];
    HW_U32 af_h_d1[ISP_AF_ROW][ISP_AF_COL];
    HW_U32 af_h_d2[ISP_AF_ROW][ISP_AF_COL];
    HW_U32 af_v_d1[ISP_AF_ROW][ISP_AF_COL];
    HW_U32 af_v_d2[ISP_AF_ROW][ISP_AF_COL];
};
```

●MEMBERS

af_count	：锐像素计数。
af_h_d1	：水平 AF 统计值 1。
af_h_d2	：水平 AF 统计值 2。
af_v_d1	：垂直 AF 统计值 1。
af_v_d2	：垂直 AF 统计值 2。

●DESCRIPTION

isp_af_stats_s 为用于描述 AF 统计值的一个结构体。

af_param_t

●PROTOTYPE

```
typedef struct isp_af_param {
    af_param_type_t type;
    HW_U32 current_frame_cnt;
    union{
        HW_S32 isp_platform_id;
        HW_S32 af_frame_id;
        af_ini_cfg_t af_ini;
        HW_S32 focus_absolute;
        HW_S32 focus_relative;
        enum auto_focus_run_mode af_run_mode;
        enum auto_focus_metering_mode af_metering_mode;
        enum auto_focus_range_new af_range;
        struct vcm_para vcm;
        bool focus_lock;
        isp_sensor_info_t sensor_info;
        af_test_config_t test_cfg;
        HW_S32 auto_focus_trigger;
    }u;
} af_param_t;
```

●MEMBERS

type ：ISP AF 命令类型，其每种类型与联合体 u 中参数一一对应，定义如下:

```
typedef enum isp_af_param_type {
    ISP_AF_PLATFORM_ID,
    ISP_AF_FRAME_ID,
    ISP_AF_INI_DATA,
    ISP_AF_FOCUS_ABSOLUTE,
    ISP_AF_FOCUS_RELATIVE,
```

```
    ISP_AF_RUN_MODE,
    ISP_AF_METERING_MODE,
    ISP_AF_RANGE,
    ISP_AF_VCM_PARAM,
    ISP_AF_LOCK,
    ISP_AF_SENSOR_INFO,
    ISP_AF_TEST_CONFIG,
    ISP_AF_TRIGGER,

    ISP_AF_PARAM_TYPE_MAX,
} af_param_type_t;
```

isp_platform_id	：ISP 平台 ID。
af_frame_id	：AF 算法执行计数。
af_ini	：AF 算法初始化 INI 配置。
focus_absolute	：对焦位置绝对值。
focus_relative	：对焦位置相对值。
af_run_mode	：AF 运行模式。
af_metering_mode	：AF 测量模式。
af_range	：AF 范围。
vcm	：AF VCM 配置。
focus_lock	：焦距锁定。
sensor_info	：Sensor 信息，包括 VTS，HTS 以及输出宽高等。
test_cfg	：AF 测试配置。
auto_focus_trigger	：自动对焦触发。

●DESCRIPTION

af_param_t 为用于描述 AF 命令参数的一个结构体。

af_result_t

●**PROTOTYPE**

```
typedef struct isp_af_result {
    enum auto_focus_status af_status_output;
    HW_U32 last_code_output;
    HW_U32 real_code_output;
    HW_U32 std_code_output;
    HW_U16 af_sap_lim_output;
    HW_U32 af_sharp_output;
} af_result_t;
```

●**MEMBERS**

af_status_output	：	AF 状态输出。
last_code_output	：	VCM 上一次 Code 值输出（归一化到 0－1224）。
real_code_output	：	VCM 实际 Code 值输出（对应配置文件）。
std_code_output	：	VCM 标准 Code 值输出（归一化到 0－1224）。
af_sap_lim_output	：	AF 锐度限制输出。
af_sharp_output	：	清晰度参考值。

●**DESCRIPTION**

af_result_t 为用于描述 AF 输出结果的一个结构体。

Declaration

The information furnished by Allwinner is believed to be accurate and reliable. Allwinner reserves the right to make changes in circuit design and/or specifications at any time without notice. Allwinner does not assume any responsibility and liability for its use. Nor for any infringements of patents or other rights of the third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of Allwinner. This datasheet neither states nor implies warranty of any kind, including fitness for any particular application.