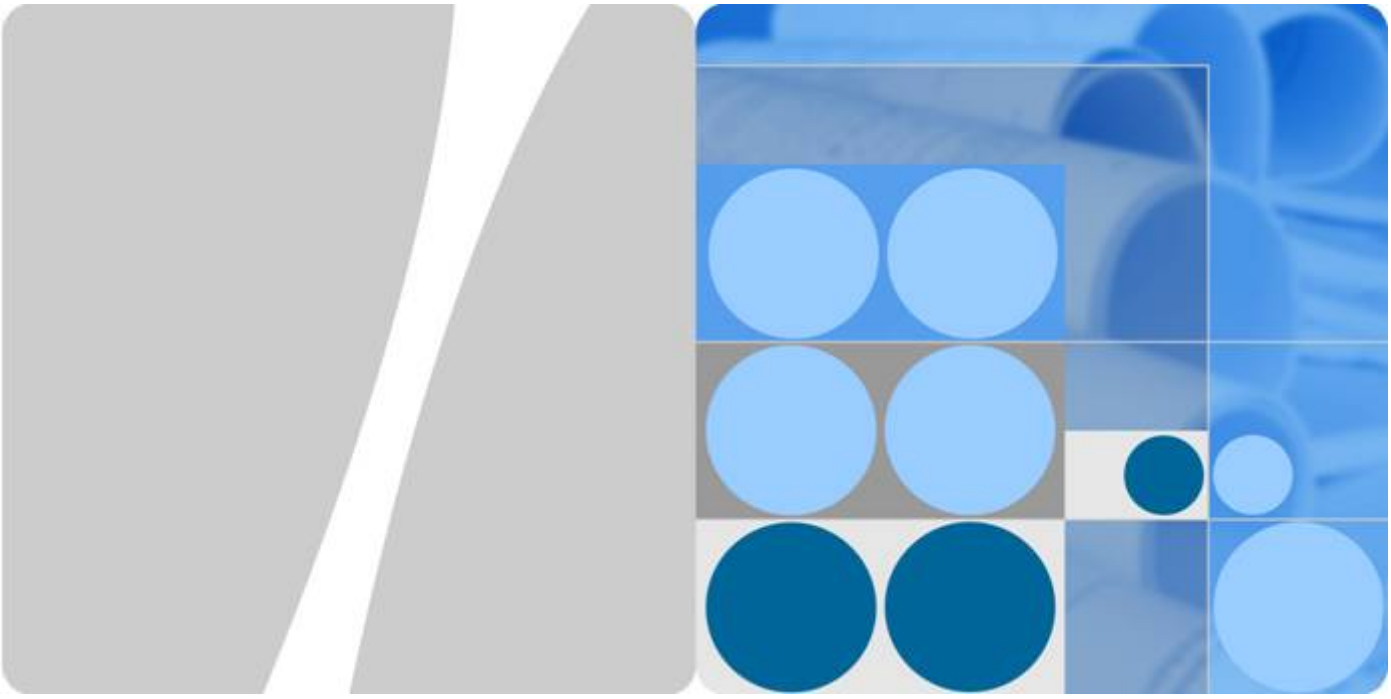


华为 SDC

9.0.0

APP 开发指南

文档版本　01

发布日期　2021-09-05

华为技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
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、服务或特性等应受华为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，华为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。

华为技术有限公司

地址： 深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址： https://e.huawei.com

目 录

1 修改说明

日期	修改描述
2019-03-08	初稿完成。
2019-07-09	修改ptz服务接口。
2019-07-13	1.修订NNIE Forward struct sdc_nnie_forward_ctrl结构体； 2.修改PTZ运动到指定位置struct ptz_3d结构体，添加struct ptz_3d_common结构体； 3.修改PTZ获取当前位置，添加响应扩展头；
2019-8-21	1.修改抓拍接口触发抓拍结构体，去除车道id，改为外设名。 2.删除预置位创建、删除接口（客户端创建、删除无法同步到web同步显示） 3.修改PTZ获取预置位结构体成员顺序
2019-9-10	1.修改VENC帧数据释放，结构体struct sdc_venc _frame中width、height顺序 2.修改抓拍接口触发抓拍结构体，支持设置的抓拍间隔由8个改为4个，增加4个保留字段
2019-10-14	1.algorithm.iaas.sdc服务添加vgs、ive功能接口
2019-10-28	1.ptz.iaas.sdc、cypto.iaas.sdc服务添加接口 2.新增透明通道接口
2019-11-12	1.cypto.iaas.sdc服务接口修改
2019-12-28	1.新增alarm.paas.sdc服务接口
2020-06-06	1.新增osd.iaas.sdc服务接口 2.新增tproxy.paas.sdc服务接口
2020-11-02	1. 修改告警事件发布说明见**6.4.2 告警事件发布** 2. 修改告警事件消除说明见**6.4.3 告警事件消除** 3. 新增告警样例见告警参考样例

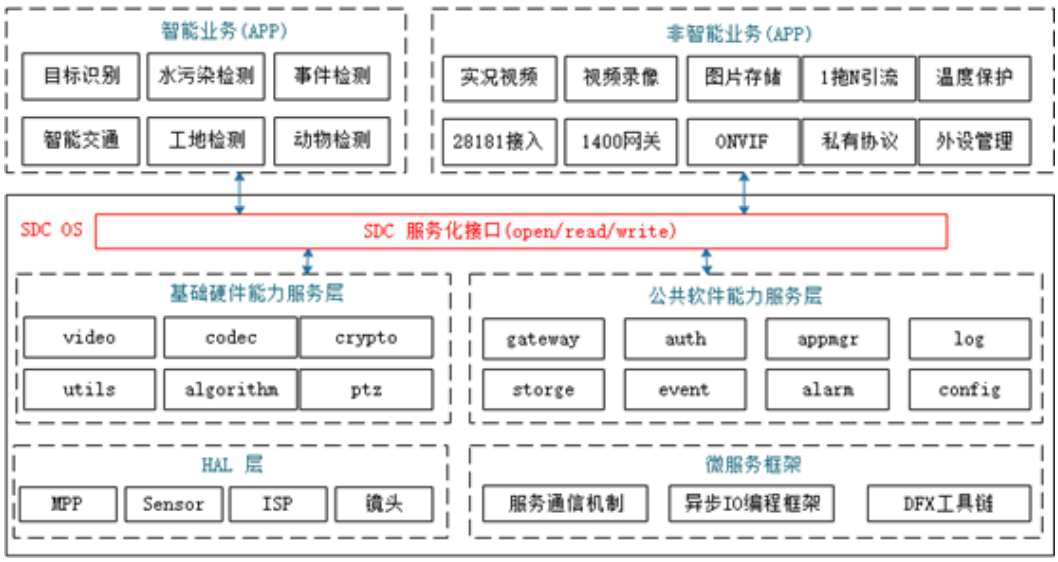
日期	修改描述
2020-11-09	1. 修改audio.iaas.sdc服务接口
2020-11-23	1. 修改crypto.iaas.sdc服务接口
2020-11-24	1. 修改alarm demo 2、多通道视频流
2021-3-18	1. 内容修订

2 SDC 软件架构简介

SDC软件架构总体示意图如下，分为如下4大部分：

1. 按需定制业务软件层：提供最终客户需要的各种业务功能，支持按需加载。可以由华为提供，也可以是任何第三方合作伙伴定制实现。
2. 公共软件能力服务层：为业务软件层提供支持，加速各种定制场景、创新业务的产品化进程。支持按需加载。
3. 基础硬件能力服务层：提供底层硬件资源的使用接口，此能力层完全由华为根据SDC硬件能力提供，保证各种硬件能力下，此层接口的稳定性，提升上层业务软件兼容多种SDC型号的能力。
4. SDC服务化接口层：定义各个能力服务层之间通信接口的规范；简单、高效、扩展性好是其目标。

图 2-1 SDC 软件架构分层示意图



3 SDC 服务化接口总体概述

遵循“服务即文件”的设计思想，服务API即为标准的文件操作接口：open/read/write/ioctl/mmap/fcntl/close。此外，相比普通文件，增强支持poll/epoll/select操作，支持基于异步IO事件的编程模式。

SDC并不发布任何Library供上层业务集成，上层业务不能也没有必要集成硬件层的海思SDK，否则可能导致硬件资源管理上的冲突。业务层和底层硬件的紧耦和将导致业务软件在多型号SDC之间的兼容性受损。

所有服务文件都位于/mnt/srvfs目录下，文件名即服务名。比如基础硬件能力服务层提供的视频数据服务对应/mnt/srvfs/video.iaas.sdc文件。

3.1 文件操作接口使用说明

3.1.1 open

利用open打开服务对应的文件获取到可读写句柄，用read/write即可实现和服务交互功能。open操作类似TCP Socket的建链操作。其中flags参数可以设置为O_NONBLOCK为非阻塞读写操作。

使用约束：

1）客户端open的flags不能包含O_CREAT，否则会得到EPERM错误码。

2）open返回的句柄不支持多线程并发访问。

3.1.2 read/write

服务间消息采用类似UDP的包传输机制，C/S之间的write/read的读写次数和大小是一一对应的。

read传入的buffer长度小于待接收数据包的大小，数据会截断返回，需要业务层自己检查数据的完整性。如果业务层希望准确获取待读取数据包大小以决定输入buffer大小，可以用ioctl获取当前可读取数据包的具体大小（参见ioctl一节）。

write写入的buffer长度大于发送缓冲区的空闲大小，则写入失败。在首次写入数据之前，用户可以利用ioctl设置写buffer的大小。

读写接口返回-1表示失败，有多种错误码：

1. ECONNRESET。表明是对端关闭连接，可能是主动关闭或者是本端发送消息不满足服务化接口协议定义导致关闭；还有一种情况是截断读取数据时，被截断的数据中包括了需要在内核转换的物理地址和虚拟地址。
2. ETIMEDOUT。客户端可能收到此错误码，表示服务端在指定时间（缺省3秒）内未处理客户端的open建链请求。
3. EFAULT。用户传递的读写BUFFER错误。
4. EAGAIN。暂时不可读写，应用层可以等待可读写事件。

3.1.3 mmap

基于mmap支持零拷贝的读写，主要用于服务端提升性能。

3.1.4 Ioctl

#define SDC_FT_PHYMEM 5

3.1.4.1 写缓冲区

#include <sys/ioctl.h>

#define SDC_FT_BUFFER 2

#define SDC_BUFF_SETWRSIZE _IO(SDC_FT_BUFFER,0x11)

#define SDC_BUFF_GETWRSIZE _IOR(SDC_FT_BUFFER,0x12,unsigned int)

注意：

1. 内部的实际大小会按8字节对齐。
2. C/S端的读/写共一个缓冲区，一端的读对应另一端的写，他们都可以设置此缓冲的大小，最先设置者生效。

样例：

设置写缓冲区（对端的读缓冲区）大小：

unsigned int wsize = 4096 * 1024;

int nret = ioctl(fd, SDC_BUFF_SETWRSIZE, wsize);

读取写缓冲区（对端的读缓冲区）大小:

unsigned int wsize;

int nret = ioctl(fd, SDC_BUFF_GETWRSIZE, &wsize);

3.1.4.2 读缓冲区

#include <sys/ioctl.h>

#define SDC_FT_BUFFER 2

#define SDC_BUFF_SETRDSIZE _IO(SDC_FT_BUFFER,0x13)

#define SDC_BUFF_GETRDSIZE _IOR(SDC_FT_BUFFER,0x14,unsigned int)

注意：

1. 内部的实际大小会按8字节对齐。
2. C/S端的读/写共一个缓冲区，一端的读对应另一端的写，他们都可以设置此缓冲的大小，最先设置者生效。

样例：

设置读缓冲区（对端的写缓冲区）大小：

unsigned int rsize = 4096 * 1024;

int nret = ioctl(fd, SDC_BUFF_SETRDSIZE, rsize);

读取读缓冲区（对端的写缓冲区）大小:

unsigned int rsize;

int nret = ioctl(fd, SDC_BUFF_GETRDSIZE, &rsize);

3.1.4.3 当前可读数据包大小

#include <sys/ioctl.h>

#define SDC_FT_BUFFER 2

#define SDC_BUFF_GETMSGSIZE _IOR(SDC_FT_BUFFER,0x15,unsigned int)

样例：

unsigned int data_size;

int nret = ioctl(fd, SDC_BUFF_GETMSGSIZE, &data_size);

3.1.4.4 读写缓冲区可映射的大小

#include <sys/ioctl.h>

#define SDC_FT_BUFFER 2

#define SDC_BUFF_GETRDMMAPSIZE _IOR(SDC_FT_BUFFER,0x16,unsigned int)

#define SDC_BUFF_GETWRMMAPSIZE _IOR(SDC_FT_BUFFER,0x17,unsigned int)

样例：

unsigned int rdmmap_size;

int nret = ioctl(fd, SDC_BUFF_ GETRDMMAPSIZE, & rdmmap_size);

3.1.4.5 物理内存地址映射

映射后的地址无cache属性。

#include <sys/ioctl.h>

struct sdc_mem {

void* addr_phy;


```
void* addr_virt;
unsigned int size;
};
#define SDC_FT_PHYMEM 5
#define SDC_PHYMEM_MMAP _IOR(SDC_FT_PHYMEM,0x00,struct sdc_mem)
```

样例：

```
struct sdc_mem mem = { .addr_phy = (void*)addr, .size = mem_size };
int nret = ioctl(fd, SDC_PHYMEM_MMAP,&mem);
memset(mem.addr_virt, 0, mem.size);
...
```

3.1.4.6 物理内存地址支持 Cache 属性的映射

映射后的地址具有cache属性。

须知

如果其他进程或者底层芯片需要更新物理地址中的数据，不能保证本进程能够读取到更新后的数据，除非业务通过其他同步机制通知本进程，确保在本进程调用CACHE FLUSH后再执行数据更新动作。

```
#include <sys/ioctl.h>
struct sdc_mem {
void* addr_phy;
void* addr_virt;
uint32_t size;
};
#define SDC_FT_PHYMEM 5
#define SDC_PHYMEM_MMAPCACHED _IOR(SDC_FT_PHYMEM,0x01,struct sdc_mem)
```

样例：

```
struct sdc_mem mem = { .addr_phy = (void*)addr, .size = mem_size };
int nret = ioctl(fd, SDC_PHYMEM_MMAPCACHED,&mem);
memset(mem.addr_virt, 0, mem.size);
...
```


3.1.4.7 物理内存地址映射解除

#include <sys/ioctl.h>

struct sdc_mem {

void* addr_phy;

void* addr_virt;

uint32_t size;

};

#define SDC_FT_PHYMEM 5

#define SDC_PHYMEM_MUNMAP _IOW(SDC_FT_PHYMEM,0x02,struct sdc_mem)

样例：

struct sdc_mem mem = {.addr_virt = (void*)addr, .size = mem_size };

int nret = ioctl(fd, SDC_PHYMEM_MUNMAP,&mem);

...

3.1.4.8 物理内存映射地址 cache flush

如果虚拟地址读写支持cache属性，在应用层基于虚拟地址更新数据之后，需要调用此接口，才能保证底层芯片可以从物理地址读取到最新数据。

#include <sys/ioctl.h>

struct sdc_mem {

void* addr_phy;

void* addr_virt;

uint32_t size;

};

#define SDC_FT_PHYMEM 5

#define SDC_PHYMEM_CACHEFLUSH _IOW(SDC_FT_PHYMEM,0x03,struct sdc_mem)

注：物理地址和虚拟地址都必须正确，否则结果可能不确定。

样例：

struct sdc_mem mem = {.phy_addr = ..., .addr_virt = ..., .size = ... };

int nret = ioctl(fd, SDC_PHYMEM_CACHEFLUSH,&mem);

...

3.1.5 fcntl

无特定约束，一般用于设置阻塞或者非阻塞读写模式。

3.1.6 close

注意：和服务端交互过程中产生的动态资源的生命周期和文件句柄的生命周期绑定。一旦文件句柄关闭，服务端会将客户端未释放资源主动回收。

3.1.7 poll/epoll/select

支持POLLIN/POLLOUT/POLLRDHUP事件。

POLLIN：缓冲区有数据可读

POLLOUT：缓冲区有空间可写

POLLRDHUP：对端关闭

3.2 基于共享缓存机制

SDC的服务之间都是进程间通信，SDC OS提供一种通用的零拷贝机制提升服务间通信效率。由各个服务选用（event.paas.sdc使用了这种机制）。

SDC基于海思芯片的底层本身提供了MMZ物理内存的申请释放功能，它主要用于业务层和底层芯片之间的数据通信，比如媒体编解码处理，算法模型加载运算处理。如果将MMZ内存应用于普遍的服务间通信，将和底层抢占MMZ资源；此外MMZ的内存在业务层的生命周期管理较为困难，因为其物理地址是永久有效，如果业务层忘记释放MMZ内存或者释放之后误用之前的物理地址都会导致难以预测的结果。SDC基于CPU管理的内存提供共享内存缓存机制，将解决这些问题。

3.2.1 CACHE 资源操作接口说明

CACHE资源由设备/dev/cache进行管理，业务只需要打开此文件获取文件句柄，即可实现资源的申请、虚拟地址的映射和释放操作。

注意：此句柄中有关资源申请、映射和释放操作支持并发，一个进程创建一个句柄即可。

3.2.1.1 CACHE 申请

就是基于设备文件/dev/cache提供的ioctl接口申请。

```
#include <sys/ioctl.h>
struct sdc_ shm_cache
{
void* addr_virt;
unsigned long addr_phy;
unsigned int size;
unsigned int cookie;
int ttl;
};
#define SDC_FT_CACHE 7
#define SDC_CACHE_ALLOC _IOR(SDC_FT_CACHE,0x00, struct SDC_shm_cache)
struct sdc_shm_cache shm = { 0 };
int nret;
int fd = open("/dev/cache", O_RDWR);
if(fd < 0) exit(-1);
/////
shm.size = size;
nret = ioctl(fd, SDC_CACHE_ALLOC,&shm);
if(nret) exit(2);
// 获取shm.addr_virt进行数据读写
nret = write_data( shm.addr_virt, shm.size);
...
```

3.2.1.2 CACHE 释放

是标准的munmap接口。

```
munmap(p, size);
```

3.2.1.3 基于 CACHE 物理地址映射虚拟地址

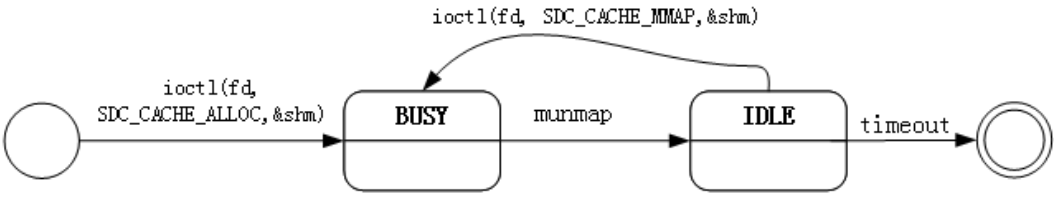
服务通过服务化接口获取到对方传入的CACHE物理地址，需要映射为本进程可访问的虚拟地址才可以访问共享内存的内容。如果物理地址无效，返回ENOENT错误码。

映射后的虚拟地址同样通过munmap接口释放。

```
#include <sys/ioctl.h>
#define SDC_FT_CACHE 7
#define SDC_CACHE_MMAP _IOR(SDC_FT_CACHE,0x01, struct SDC_shm_cache)
struct sdc_shm_cache shm = { 0 };
int fd = open("/dev/cache", O_RDWR);
//...
shm.addr_phy = addr_phy;
shm.size = size;
shm.cookie = cookie;
nret = ioctl(fd, SDC_CACHE_MMAP, &shm);
if( 0 == nret) return shm.addr_virt;
```

3.2.2 CACHE 生命周期管理机制说明

共享CACHE内存自由在服务之间流动，SDC实现CACHE生命周期的自动管理机制。每个申请的CACHE空间都有一个老化时间，在没有任何业务进程拥有指向它的虚拟地址之后（即所有业务进程都调用了munmap释放虚拟地址）其内存进入空闲状态，在超时之前如果有任何业务进程重新将其物理地址映射为进程可用的虚拟地址则重新进入忙状态，没有进程应用，则此CACHE空间会被回收。



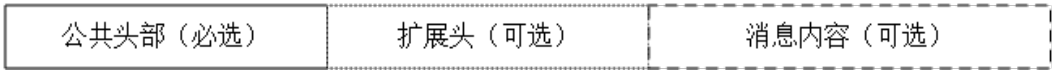
此延时回收所需要的最小超时时间基本上和服务间消息传递时延对应，和业务层各种场景下数据的缓存时间没有关系。比如SDC北向的GB-T 1400协议中会定义和云端通信中断后上报数据的最小缓存时间，业务层的缓存时间决定了业务代码调用munmap释放虚拟地址资源的时机。

如果业务繁忙无法及时接收服务间消息，业务获取到已经老化了的CACHE物理地址，映射的时候会返回ENOENT错误码，这是一种天然的流控机制，避免后端阻塞时占用过多资源。

如果业务空闲，可用内存资源较多，则老化时间会自动延长，可以保证特定业务时延抖动较大场景下的连续性。

3.3 服务化接口消息规范（HBTP）

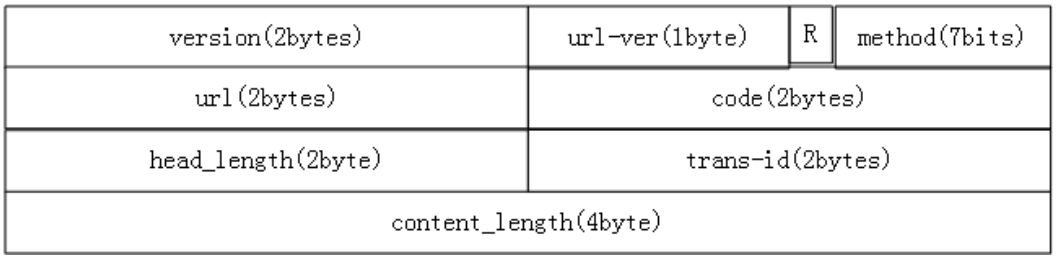
所有消息最多包括三部分：公共头部、扩展头和消息内容。其中扩展头和消息内容都是可选。每个服务接口的具体消息内容和扩展头的内容参见各个服务的接口定义章节。



注意：以下服务化接口的消息结构都按照网络序（高字节在低位）定义；在当前SDC的硬件环境上都是小端序（高字节在高位）。

3.3.1 公共头部

按网络序（大端序）如下定义：



字段	长度	语义
version	2bytes	协议版本，当前固定为0x5331
url_ver	1byte	用于接口版本间兼容。
R	1bit	请求和响应消息的指示。1代表响应。
method	7bits	方法定义： 1：CREATE 2：GET 3：UPDATE 4：DELETE
url	2bytes	资源标识
code	2bytes	响应码（仅在R==1时有效）
head_length	2bytes	公共头和扩展头的总长度
trans-id	2bytes	事务标识，服务响应头中的trans_id和请求中的trans_id对应，在类似http的pipeline场景中，服务端不能保证响应和请求的顺序保持一致，客户端可以基于此字段实现请求和响应的匹配。
content_length	4bytes	消息内容的总长度

利用gcc有关字节序的宏，数据结构定义可以参考如下形式：

struct sdc_common_head


```
{
uint16_t hbtp_ver;
uint8_t uri_ver;
#if defined(__BYTE_ORDER__) && defined(__ORDER_LITTLE_ENDIAN__) &&
defined(__ORDER_BIG_ENDIAN__)
#if (__BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__)
uint8_t method: 7;
uint8_t response: 1;
#elif (__BYTE_ORDER__ == __ORDER_BIG_ENDIAN__)
uint8_t response: 1;
uint8_t method: 7;
#else
#error "unknown __BYTE_ORDER__"
#endif
#else
#error "don't define __BYTE_ORDER__ or __ORDER_LITTLE_ENDIAN__ or
__ORDER_BIG_ENDIAN__"
#endif
uint16_t uri;
uint16_t code;
uint16_t head_length;
uint16_t trans_id;
uint32_t content_length;
};
```

响应码code在R为1时有效，返回服务端响应结果，参考定义如下：

```
HBTP_CODE_200 = 200, // OK
HBTP_CODE_400 = 400, // Bad Request
HBTP_CODE_401 = 401, // Unauthorized
HBTP_CODE_403 = 403, // Forbidden
HBTP_CODE_404 = 404, // Not Found
HBTP_CODE_500 = 500, // Internal Server Error
HBTP_CODE_509 = 509，//flow control
```


3.3.2 扩展头定义

扩展头的具体类型值和此类型对应的数据结构的定义由各个服务接口确定。总长度保证8字节对齐。

字段名	长度	语义
type	2	其含义有各个服务定义
length	2	扩展头及其内容的总长度，不包括padding字符
扩展头内容	hdr_len - 8	实际携带的消息体内容
padding	((hdr_len + 7) & ~7) – hdr_len	保证整个扩展头长度按8字节对齐

扩展头参考定义形式如下：

```
struct sdc_extend_head
{
uint16_t hdr_type;
uint16_t hdr_len;
uint32_t reserve;
};
```

说明

无论是服务端还是客户端，都能遵循忽略自己不认识的扩展头的处理原则。

4 APP 软件发布、安装、运行环境规范

4.1 编译工具链

https://www.linaro.org/downloads/

CPU芯片系列	toolchain
海思3519A	arm-linux-gnueabihf(32-bit Armv7 Cortex-A, hard-float, little-endian)
海思3516D	arm-linux-gnueabihf(32-bit Armv7 Cortex-A, hard-float, little-endian)
海思3559A	aarch64-linux-gnu(64-bit Armv8 Cortex-A, little-endian)

4.2 软件包规范

APP软件包分为基础包和补丁包，无论哪种，均需要满足如下规范：

1. 软件包必须是rpm格式, 仅支持rpm基本安装（暂不支持依赖检查、签名校验、安装后处理等）。软件包建议不超过150MB。实际安装包将基础软件包.rpm、证书文件、签名文件一起打包成tar包安装，该包用于SDC APP安装。
2. 基础包命名格式为{app name}-{version}-{release}.{aarch}.rpm（不区分大小写）。其中{app name}仅支持字母数字和下划线“_”，只能以字母或者下划线开始；{version}仅支持数字和点“.”； {release}仅支持数字； {aarch}仅支持armv7和aarch64。备注：由于文件名是不可靠的，因此该命名规范仅仅是为了更容易管理文件，名称可随意修改，虽然强烈不建议这么做。
3. 通过补丁包机制，app可以更新部分文件（比如License，功能配置文件等），可以将这些部分文件也打包为rpm格式的patch包，补丁包的name命名格式为：

{app name}.{patch name}-{version}-{release}.{aarch}.rpm，这类软件包会覆盖安装到相同名称的基础包的安装目录，app可以通过检查特定文件的变化情况，或者手工重启app使得更新生效。

4. 所谓升级就是安装同{app name}但不同{version}的新的软件包，所有已安装的文件都会丢失(可以理解为系统盘重装了)，除非这些文件保存在外部的数据盘中 (参见SDC系统环境变量一节中data_disk.app.sdc的设置说明 ）。升级之后，业务需要自己检测数据盘中的数据是否存在；如果存在，则根据需要APP业务实现数据盘数据版本间兼容访问。
5. 支持补丁包安装，安装补丁包必须要求SDC已经安装了同名称，同版本号的基础包。补丁包安装不支持系统盘数据盘分配。补丁包的配置文件被忽略，最终安装时视作普通文件，仍然会覆盖到系统盘，因此不建议补丁包携带sdc.conf文件。此外考虑到补丁包安装后会覆盖旧版本文件，因此不支持补丁包卸载，可以通过将旧版本文件制作为新的补丁包并安装的方式实现补丁回退，但是APP的已安装补丁会记录这两个补丁。
6. 软件包安装的%{buildroot}为系统自动为app分配的根目录，app仅对%{buildroot}目录具有可写权限，其他目录都是只读。app通过环境变量获取到%{buildroot}的真实目录。
7. app的主程序必须位于%{buildroot}/bin目录下，且命名为main(可执行程序)或者main.sh(shell脚本)，如果同时存在，则可执行程序main作为主程序。主程序由SDC OS负责启动，启动后主进程必须是常驻进程，它的生命周期即为整个APP的运行时生命周期。app主程序启动后的当前目录即为%{buildroot}，可以通过环境变量rootdir.app.sdc获取。
8. 正常停止业务的时候，SDC OS会向app的主进程发送SIGTERM信号，如果主进程忽略了此信号处理，会在10秒超时后强制停止app进程。

4.3 SDC 系统环境变量

app在rpm的根目录下可以生成sdc.conf文件来定义app对底层硬件资源的需求，这个文件内容每行的格式为key=value。除磁盘定义外，其他定义的value不支持空格/TAB等空白字符。

key	含义	value
max_mem.app.sdc	APP申请的最大内存，单位MB。默认值表示不限制。	数字，如50指app最多可以分配50MB运行内存。
max_cpu.app.sdc	APP可以使用CPU运行时间的比例，相对于所有CPU总运行时间而言。取值0～100，0表示不受限制，等价于100。默认值表示不限制。	例如100%，表示APP可以使用所有CPU的所有运行时间；50%，表示APP可以使用一半的CPU运行时间。需要注意：CPU运行时间是不区分大小核的，相同运行时间，使用小核和使用大核得到的CPU运行资源是不一样的。
cpus.app.sdc	APP只能运行在指定的CPU上，两种格式：1,2或0-3。默认值表示不做限制。	例如0-3表示可以运行在0/1/2/3四个CPU上，1,2表示可以运行在1/2两个CPU上。0表示只能运行在0号CPU上。当前3559A平台0/1是大核，2/3是小核；3519A平台0是大核，1是小核。如果该参数配置不当，比如3519A芯片平台配置为0-3，则会导致app安装/启动失败。

key	含义	value
sys_disk.app.sdc	定义的系统盘空间大小。只支持一个系统盘。默认值时SDCOS根据rpm信息猜测空间的小，在尽量满足空间诉求的情况下减少浪费。猜测算法有一定局限性，在小文件非常多的场景可能导致分配空间不足。	[flags] <size> 必选：定义系统盘大小，当前仅支持字节单位。0为无效值，为0时，SDC OS自动确认系统盘大小，自动确认情况下系统盘余量较小，约20%左右。 [sd|flash] 可选：指定存储位置，外置SD卡、内置Flash，如果未指定由SDC自动选择存储位置，如果均没有找到合适大小的位置，则分配失败。 注：如果需要指定存储位置或者未指定场景SDC自动分配为sd卡时，需要注意相机sd卡空间分配。由于sd卡设计上优先配给录像抓拍存储，用户需要根据APP实际需求人为设置sd卡占用空间，设置路径为存储管理->录像抓拍最大占用空间（%），数据盘也是如此。 样例： sys_disk.app.sdc=50000000 sd

key	含义	value
data_disk.app.sdc	定义数据盘挂载目录和空间大小。当前仅支持0个或者1个数据盘。默认值表示不分配数据盘。 注：900 TR6（2021.8.1以后）版本开始支持0-5个数据盘申请，app可根据实际需求定义。	[flags] name: 磁盘名称，必选，磁盘分配后挂载到系统盘的根路径同名目录，如name为conf，则挂载到系统盘根路径的conf目录。 <size> 必选：定义大小，当前仅支持字节单位。该字段必须指定，否则数据盘分配失败。size至少需要支撑ext4分区的创建，因此最小为1MB，少于1MB则创建磁盘失败。 [sd | flash] 可选：指定存储位置，外置sd卡或者内置flash，如果未指定则由SDC自动选择存储位置，如果均没有找到合适大小的位置，则分配失败。 注： 1.sdc 900 TR6（2021.8.1以后）版本开始支持多数据盘申请，最多支持5个数据盘，数据盘定义以data_disk*.app.sdc为key，key只支持样例中的五种定义，其余定义皆不能生效，数据盘name不能重名。 2.app版本迭代中涉及到旧数据拷贝，以data_disk*.app.sdc为key将旧数据拷贝到新数据盘，如果高版本app存在删减数据盘的情况，旧数据盘不会被删除，但也不会挂载到容器内部，app卸载时会一并删除。数据盘容量只支持扩大，不支持缩小。 样例: data_disk.app.sdc=mydisk 50000000 sd data_disk1.app.sdc=mydisk1 50000000 sd data_disk2.app.sdc=mydisk2 50000000 sd data_disk3.app.sdc=mydisk3 50000000 flash data_disk4.app.sdc=mydisk4 50000000 flash
sdc_version	获取SDC 版本号	app可以使用获取环境变量的方式获取版本号 样例：getenv("sdc_version");

注：

1. sdc.conf中所有的key=value（除sys_disk.app.sdc和data_disk*.app.sdc之外），都会注入到app的main进程，app可以通过libc的getenv获取。

2. key不存在或者value为空均表示key取默认值。

此外，SDC还会注入如下环境变量。

环境变量名	含义	样例
rootdir.app.sdc	APP的根目录，绝对路径，对应rpm包中的%{buildroot}；注意，该路径是容器内的绝对路径，与在host上看到的绝对路径可能不同。	/usr/app/3rdApp 注：这就是main/main.sh运行时的当前目录。

4.4 APP 运行时环境介绍

为了保证安全，每个APP都在独立的容器中运行，其运行环境有以下特点：

1、APP只能访问本APP安装的文件目录、临时文件目录/tmp和服务文件所在目录/mnt/srvfs;

2、APP的权限无法访问海思SDK，只能访问SDC OS提供的服务化接口；

3、APP无法访问网络，只能通过SDC提供的HTTP代理或者TCP/UDP代理和外界交互。

5 基础硬件能力服务化接口参考

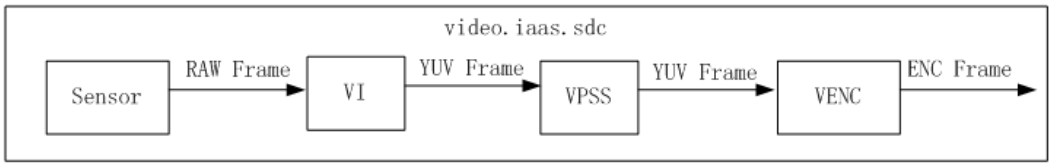
总共划分6大服务提供SDC的基础软件硬件能力，其功能描述和当前版本状态如下。

服务名	功能介绍	当前版本支持情况
video.iaas.sdc	视频数据服务。获取视频原始帧和H264/H265编码后的帧数据	已经支持
codec.iaas.sdc	编解码服务。提供底层芯片支持的图像编解码和合成功能	已经支持
crypto.iaas.sdc	加解密服务。提供底层芯片支持的加解密功能	已经支持
algorithm.iaas.sdc	支持深度学习等智能算法的服务化接口，当前仅支持NNIE。	已经支持
ptz.iaas.sdc	云台服务。提供云台控制功能。	已经支持
utils.iaas.sdc	提供底层资源的其他管理功能，比如MMZ、外设、串口等管理控制功能	已经支持
osd.iaas.sdc	提供OSD设置变量功能	已经支持

所有服务的接口都基于SDC服务化接口规范HBTP来定义，具体请参考SDC 服务化接口总体概述。

5.1 video.iaas.sdc 服务化接口定义

5.1.1 功能定义



主要提供YUV/ENC Frame输出数据的订阅功能。

5.1.2 逻辑通道定义

总体上分为两类通道，一类是YUV（格式默认为YVU_420SP）视频帧数据通道；另一类是编码后视频帧（H264/H265）通道。用户可以查询每个通道的属性及其使用状态，根据自己的需要设定数据输出格式。如果存在多个用户，需要用户之间协作来消除潜在的通道资源、配置的冲突。

5.1.2.1 YUV 帧数据通道定义

资源	类型	取值范围
YUV抓拍帧通道（仅ITS款型支持）	uint32_t	0
YUV帧数据通道	uint32_t	[1-99] (通过YUV通道查询接口获取到具体通道号，不同硬件支持的通道数量有差异)。

5.1.2.2 VENC 帧数据通道定义

资源	类型	取值范围
VENC帧数据通道	uint32_t	[100-104] (不同硬件支持有差异，具体和web配置保持一致，主码流-100，子码流1-101，子码流2-102，子码流3-103，子码流4-104)

5.1.3 YUV 逻辑通道属性设置

注：当前仅支持YUV帧通道的属性设置；VENC帧通道的属性只能通过WEB页面修改。

5.1.3.1 请求 Common Head 方法资源

METHOD	URI含义	URI取值
UPDATE	SDC_URL_YUV_CHANNEL	0x00

5.1.3.2 请求 Content

#define SDC_YVU_420SP0 //YUV帧，当前仅支持YVU_420SP

struct sdc_yuv_channel_param

{

uint32_t channel;//通道0位抓拍通道（its设备）

uint32_t width;

uint32_t height;

uint32_t fps; //取值0：表示使用底层缺省的帧率。

uint32_t on_off; //取值0：表示关闭通道；取值非0值：表示开启通道

uint32_t format; //SDC_YVU_420SP

};

支持批量设置，通道数量=hbtp.content_length / sizeof(sdc_yuv_channel_param);

只要输入参数正确，则总是生效，业务之间的通道冲突由业务层解决。

5.1.3.3 请求扩展头

无。

5.1.3.4 响应码

如果功能正常，响应码200，输入错误为400，服务端错误为500。

5.1.3.5 响应 Content

无。

5.1.3.6 响应扩展头

无。

5.1.3.7 参考样例

```
#include "sdc.h"
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/uio.h>

int main(int argc,char* argv[])
{
struct sdc_yuv_channel_param param = {
.channel = 1,
.width = 1280,
.height = 720,
.fps = 25,
.on_off = 1,
.format = SDC_YVU_420SP,
};

struct sdc_common_head head = {
.version = SDC_VERSION, //0x5331
.url = SDC_URL_YUV_CHANNEL, //0x00
.method = SDC_METHOD_UPDATE, //0x02
.content_length = sizeof(param),
.head_length = sizeof(head),
};
struct iovec iov[2] = { {.iov_base = &head, .iov_len = sizeof(head)},
{.iov_base = &param, .iov_len = sizeof(param) }};
int nret;
int fd = open("/mnt/srvfs/video.iaas.sdc", O_RDWR);
if(fd < 0) goto fail;

nret = writev(fd, iov, 2);
if(nret < 0) goto fail;

nret = read(fd,&head, sizeof(head));
if(head.code == SDC_CODE_200 /** 200 */) {
//...
}else{
//...
}
close(fd);
return 0;
fail:
```

```
exit(1); //fd will be closed after exit
}
```

5.1.4 YUV 逻辑通道属性查询

5.1.4.1 请求 Common Head 方法资源

方法	URL含义	URL取值
GET	SDC_URL_YUV_CHANNEL	0x00

5.1.4.2 请求 Content

uint32_t channel;

支持批量查询逻辑通道属性，通道数量=hbtp.content_length / sizeof(uint32_t);

如果无Content，即hbtp.content_length == 0，则返回所有通道的属性。

5.1.4.3 请求扩展头

无。

5.1.4.4 响应码

如果功能正常，响应码200，输入错误为400，服务端错误为500。

5.1.4.5 响应 Content

```
struct sdc_resolution {
uint32_t width;
uint32_t height;
};

struct sdc_yuv_channel_info
{
struct sdc_yuv_channel_param param;
struct sdc_resolution max_resolution; //本通道支持的最大分辨率
uint32_t is_snap_channel; //是否是抓拍通道，0：普通通道，1：抓拍通道
uint32_t src_id; //数据源，数据源以IP地址标识此通道数据的来源是本地还是其他摄像机，对于本地多目摄像机，则以127.0.0.x区分，其中x从1开始递增。
```


uint32_t subscriber_cnt; //此通道数据订阅数量。大于1表示有多个用户订阅同一个通道的数据。因为视频数据都是零拷贝的形式传递到用户，如果某个用户修改了从此通道获取的数据内容，则其他用户都会读取到修改后的数据。

uint32_t resolution_moditfy; //此通道分辨率参数是否支持修改。0---不支持修改，1--支持修改。

};

批量查询逻辑通道属性时，响应Content也是批量返回。批量返回的通道数量=hbtp.content_length / sizeof(struct sdc_yuv_channel_info);

5.1.4.6 响应扩展头

无。

5.1.4.7 参考样例

```
#include "sdc.h"
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>

static void display_yuv_channel_info(struct sdc_yuv_channel_info* info);

int main(int argc,char* argv[])
{
int nret,i;
char buf[1024] = { 0 };
struct sdc_yuv_channel_info* info;
struct sdc_common_head* head = (struct sdc_common_head*)buf;

/** query all channels' info */
head->version = SDC_VERSION; //0x5331
head->url = SDC_URL_YUV_CHANNEL; //0x00
head->method = SDC_METHOD_GET; //0x02
head->head_length = sizeof(*head);

int fd = open("/mnt/srvfs/video.iaas.sdc", O_RDWR);
if(fd < 0) goto fail;

nret = write(fd, head, head->head_length + head->content_length);
if(nret < 0) goto fail;

nret = read(fd,buf,sizeof(buf));
if(nret < 0 || head->code != SDC_CODE_200) goto fail;

info = (struct sdc_yuv_channel_info*)&buf[head->head_length];
for(i = 0; i < head->content_length / sizeof(*info); ++i, ++info) {
display_yuv_channel_info(info);
}

close(fd);
return 0;
fail:
exit(1); //fd will be closed after exit
}

static void display_yuv_channel_info(struct sdc_yuv_channel_info* info)
{
}
```

5.1.5 YUV 帧数据订阅

一个连接（客户端的一个句柄）可以多次订阅视频数据，但只有最后一次订阅条件生效。

注：订阅数据的fd关闭后，其上订阅的所有YUV帧数据会被服务端自动回收。

5.1.5.1 请求 Common Head 方法资源

方法	URL含义	VALUE
GET	SDC_URL_YUV_DATA	0x01

5.1.5.2 请求 Content

uint32_t channel;

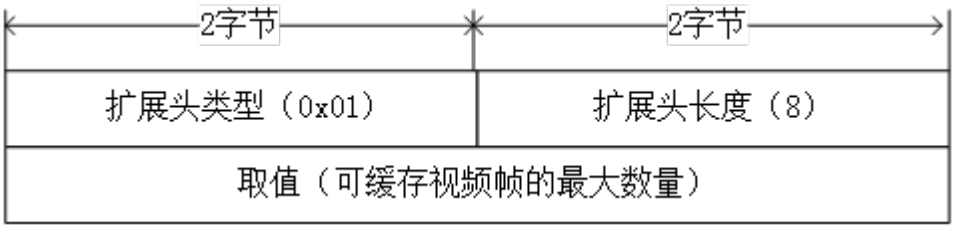
支持获取多通道，通道数量=hbtp.content_length / sizeof(uint32_t);

5.1.5.3 请求扩展头

5.1.5.3.1 指示多通道同步扩展头（SDC_HEAD_YUV _SYNC）

取值保存在扩展头的Reserver字段。订阅多个通道，如果无此扩展头，缺省按照同步策略处理。

5.1.5.3.2 设置流控阈值扩展头(SDC_HEAD_YUV _CACHED_COUNT_MAX)



取值保存在扩展头的Reserver字段。如果无此扩展头，则由服务端缺省缓存数量=每个通道的帧率之和 * 2（即最多占用2秒的YUV帧资源）。

如果用户缓存（未释放）的视频帧总数量超过此预知，则服务端启动流控功能，不再转发订阅的视频帧数据。因此用户在不需要的时候应该及时调用YUV帧的释放接口。

5.1.5.3.3 订阅随帧参数扩展头(SDC_HEAD_YUV _PARAM_MASK)

一般用于抓拍帧通道，当前仅支持0x08，响应头中会携带业务层触发的抓拍时传递的id值。

5.1.5.4 响应码

如果功能正常，响应码200；如果启动流控，响应码509（Bandwidth Limit Exceeded），无内容；其他如果输入错误，响应码为400；如果服务端错误，响应码为500，无内容。

5.1.5.5 响应 Content

订阅成功后会连续收到多个报文，第一个响应报文没有content。

后续报文包括YUV帧数据，结构如下定义：

```
struct sdc_yuv_frame
{
uint64_t addr_phy;
uint64_t addr_virt；//默认cache映射
uint32_t size;
uint32_t width;
uint32_t height;
uint32_t stride;
uint32_t format;
uint32_t reserve;
uint32_tcookie[4]; //服务业务功能或者调测需要
}

struct sdc_yuv_data
{
uint32_t channel;
uint32_t reserve;
uint64_t pts; //底层芯片携带的时间戳，单位微妙
uint64_t pts_sys; //服务端获取到数据的时间戳，单位微妙
```


struct sdc_yuv_frame frame;

};

批量订阅通道数据时，通道数量=hbtp.content_length / sizeof(struct sdc_yuv_data);

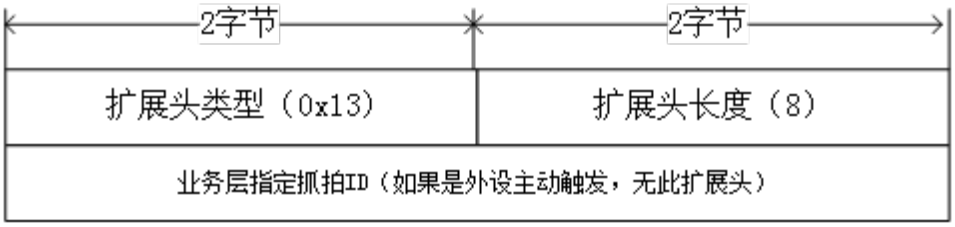
说明：多目SDC的底层芯片的时间可能无法保持绝对一致，多通道的同步依赖系统时间，即pts_sys而非pts字段。

注：多个报文中他们的响应头中的trans_id都会和订阅请求发送的trans_id保持一致。

5.1.5.6 响应扩展头

5.1.5.6.1 视频帧缓存计数扩展头(SDC_HEAD_YUV _CACHED_COUNT)

5.1.5.6.2 抓拍随帧参数扩展头(SDC_HEAD_YUV_PARAM_SNAP)



指定抓拍ID，参见抓拍帧接口的输入。

5.1.5.7 参考样例

```
#include "sdc.h"
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>

static void display_extend_head(struct sdc_extend_head* extend_head) {}
static void display_yuv_data(struct sdc_yuv_data* yuv_data) {}
int main(int argc,char* argv[])
{
int nret,i,fd;
char buf[1024] = { 0 };
struct sdc_common_head* head = (struct sdc_common_head*) buf;
struct sdc_extend_head* extend_head;
struct sdc_yuv_data* yuv_data;
struct sdc_yuv_snap_param* snap_param;
uint32_t* channel;

fd = open("/mnt/srvfs/video.iaas.sdc", O_RDWR);
if(fd < 0) goto fail;

head->version = SDC_VERSION;
head->url = SDC_URL_YUV_DATA;
head->method = SDC_METHOD_GET;
head->head_length = sizeof(*head);

/** 最大缓存YUV帧数量为10 */
extend_head = (struct sdc_extend_head*)&buf[head->head_length];
extend_head->type = SDC_HEAD_YUV_CACHED_COUNT_MAX;
extend_head->length = sizeof(*extend_head);
extend_head->reserve = 10;
/**
* #define sdc_extend_head_length(extend_head) (((extend_head)->length + 7) & ~7)
* 现有SDC服务接口中定义的扩展头长度都已经按8字节对齐，如果确保如此可以这样写head->head_length += extend_head->length，这需要开发者保证。
*/
head->head_length += sdc_extend_head_length(extend_head);

/** 订阅2个通道，无需同步，因为其中一个通道为抓拍通道 */
extend_head = (struct sdc_extend_head*)&buf[head->head_length];
extend_head->type = SDC_HEAD_YUV_SYNC;
extend_head->length = sizeof(*extend_head);
extend_head->reserve = 0;
```

```
head->head_length += sdc_extend_head_length(extend_head);
/** 订阅2个通道数据 */
channel = (uint32_t*)&buf[head->head_length];
channel[0] = 0; //0缺省为抓拍通道 ， 通过查询接口获取具体抓拍通道号更好
channel[1] = 1;
head->content_length = 2 * sizeof(channel[0]);
nret = write(fd, head, head->head_length + head->content_length);
if(nret < 0) goto fail;
for(;;) {
nret = read(fd, buf,sizeof(buf));
if(nret < 0) goto fail;
switch(head->url){
case SDC_URL_YUV_SNAP:
/** 处理抓拍的响应 */
if(head->code != SDC_CODE_200) {
// log error info
}
continue;
case SDC_URL_YUV_DATA:
break;
default:
continue;
}
/**
*#define sdc_extend_head_next(extend_head) ((struct sdc_extend_head*)
((char*)extend_head + sdc_extend_head_length(extend_head)))
* #define sdc_extend_head_first(common_head) ((struct sdc_extend_head*)
(common_head + 1))
* #define sdc_for_each_extend_head(common_head, extend_head) \
* for( extend_head = sdc_extend_head_first(common_head); (char*)extend_head
- (char*)common_head < common_head->head_length; extend_head =
sdc_extend_head_next(extend_head))
*/
sdc_for_each_extend_head(head, extend_head) {
display_extend_head(extend_head);
}
for(i = 0, yuv_data = (struct sdc_yuv_data*)&buf[head->head_length]; i < head-
>content_length / sizeof(*yuv_data); ++i, ++yuv_data) {
display_yuv_data(yuv_data);
}
```



```
/** free yuv_data */
head->response = head->code = 0;
head->method = SDC_METHOD_DELETE;
(void)write(fd,head,head->head_length + head->content_length); // server ignore extended headers
/** 触发抓拍动作 */
if( 1 ) {
head->url = SDC_URL_YUV_SNAP;
head->method = SDC_METHOD_CREATE;
head->head_length = sizeof(*head);
head->content_length = sizeof(*snap_param);
snap_param = (struct sdc_yuv_snap_param*)&buf[head->head_length];
snap_param->id = 100;
snap_param->num = 1;
snap_param->interval_msec = 0;
nret = write(fd, head, head->head_length + head->content_length);
if(nret < 0) goto fail;
/** 马上读取不能保证是抓拍响应 */
}
}
return 0;
fail:
exit(1);
}
```

5.1.6 YUV 帧数据释放

YUV帧数据都是零拷贝传递，订阅的时候获取的YUV帧都加入用户缓存视频帧的计数值，用户使用完毕之后需要及时释放，否则用户缓存视频帧的计数值达到阈值后，用户将无法再收到YUV帧数据。

注：资源释放类接口都无响应，类似C++的析构函数的行为，无返回值。

5.1.6.1 请求 Common Head 方法资源

方法	URL	VALUE
DELETE	SDC_URL_YUV_DATA	0x01

5.1.6.2 请求 Content

struct sdc_yuv_data;

支持批量释放，地址数量=hbtp.content_length / sizeof(struct sdc_yuv_data);

5.1.6.3 请求扩展头

无。

5.1.6.4 响应码

无。

5.1.6.5 响应 Content

无。

5.1.6.6 响应扩展头

无。

5.1.6.7 参考样例

参见**YUV帧数据订阅一节样例。**

5.1.7 VENC 逻辑通道属性设置

此功能暂不支持，建议使用web页面进行通道属性设置。

5.1.7.1 请求 Common Head 方法资源

无。

5.1.7.2 请求 Content

无。

5.1.7.3 请求扩展头

无。

5.1.7.4 响应码

无。

5.1.7.5 响应 Content

无

5.1.7.6 响应扩展头

无。

5.1.7.7 参考样例

无。

5.1.8 VENC 逻辑通道属性查询

一个连接（客户端的一个句柄）可以多次订阅视频数据，但只有最后一次订阅条件生效。

注：订阅数据的fd关闭后，其上订阅的所有VENC帧数据会被服务端自动回收。

5.1.8.1 请求 Common Head 方法资源

方法	URL含义	VALUE
GET	SDC_URL_VENC_CHANNEL	0x02

5.1.8.2 请求 Content

uint32_t channel;

支持批量查询逻辑通道属性，通道数量=hbtp.content_length / sizeof(uint32_t);

如果无Content，即hbtp.content_length == 0，则返回所有通道的属性。

5.1.8.3 请求扩展头

无

5.1.8.4 响应码

如果功能正常，响应码200.输入错误为400，服务端错误为500。

5.1.8.5 响应 Content

struct sdc_resolution {

uint32_t width;

uint32_t height;

};

#define SDC_PT_H264 96

#define SDC_PT_H265 265

#define SDC_PT_MJPEG 1002

struct sdc_venc_channel_ability {

uint32_t max_fps; //该通道支持的最大帧率

uint32_t format[3]; //该通道支持的编码格式，H264->96、H265->265、MJPEG->1002

uint32_t resolution_num; //该通道支持分辨率类型数

struct sdc_resolution chn_resolution[0]; //该通道支持的所有分辨率类型

};


```
struct sdc_venc_channel_param
{
uint32_t channel;
uint32_t width;
uint32_t height;
uint32_t fps;
uint32_t on_off;
uint32_t format;
}
struct sdc_venc_channel_info
{
struct sdc_venc_channel_param param;
uint32_t src_id; //数据源，数据源以IP地址标识此通道数据的来源是本地还是其他摄像机，对于本地多目摄像机，则以127.0.0.x区分，其中x从1开始递增。
uint32_t subscriber_cnt; //此通道数据订阅数量。大于1表示有多个用户订阅同一个通道的数据。因为视频数据都是零拷贝的形式传递到用户，如果某个用户修改了从此通道获取的数据内容，则其他用户都会读取到修改后的数据。
struct sdc_venc_channel_ability stchnability; //该通道支持的编码能力信息
};
```

批量查询逻辑通道属性时，响应Content也是批量返回。批量返回的通道数量=hbtp.content_length / sizeof(struct sdc_venc_channel_info);

5.1.8.6 响应扩展头

无

5.1.8.7 参考样例

```
#include "sdc.h"
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>

int main(int argc,char* argv[])
{
int nret,i;
char buf[1024] = { 0 };
struct sdc_venc_channel_info* info;
struct sdc_common_head* head = (struct sdc_common_head*)buf;

/** query all channels' info */
head->version = SDC_VERSION; //0x5331
head->url = SDC_URL_VENC _CHANNEL; //0x02
head->method = SDC_METHOD_GET; //0x02
head->head_length = sizeof(*head);

int fd = open("/mnt/srvfs/video.iaas.sdc", O_RDWR);
if(fd < 0) goto fail;

nret = write(fd, head, head->head_length + head->content_length);
if(nret < 0) goto fail;

nret = read(fd,buf,sizeof(buf));
if(nret < 0 || head->code != SDC_CODE_200) goto fail;
/** deal with all channels' info */

close(fd);
return 0;
fail:
exit(1); //fd will be closed after exit
}
```

5.1.9 VENC 帧数据订阅

一个连接（客户端的一个句柄）可以多次订阅视频数据，但只有最后一次订阅条件生效。

注：订阅数据的fd关闭后，其上订阅的所有VENC帧数据会被服务端自动回收。

5.1.9.1 请求 Common Head 方法资源

方法	URL含义	VALUE
GET	SDC_URL_VENC_DATA	0x03

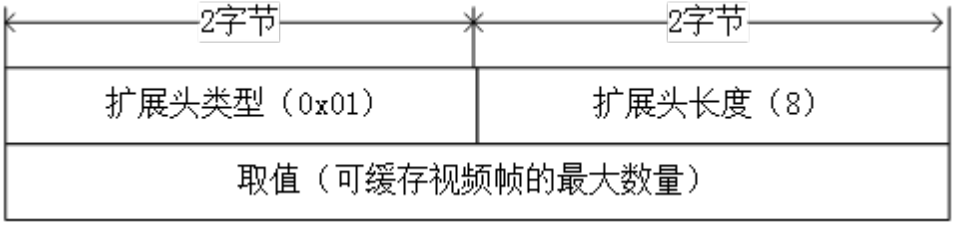
5.1.9.2 请求 Content

uint32_t channel;

支持获取多通道，通道数量=hbtp.content_length / sizeof(uint32_t);

5.1.9.3 请求扩展头

设置流控阈值扩展头(SDC_HEAD_VENC_ CACHED_COUNT_MAX)



取值保存在扩展头的Reserver字段。如果无此扩展头，则由服务端缺省缓存数量=每个通道的帧率之和 * 2（即最多占用2秒的编码视频帧资源，这是以数量为准和每帧大小无关，客户端应该都是缓存连续的视频帧）。

如果用户缓存（未释放）的视频帧总数量超过此预知，则服务端启动流控功能，不再转发订阅的视频帧数据。因此用户在不需要的时候应该及时调用编码视频帧的释放接口。

5.1.9.4 响应码

如果功能正常，响应码200，

如果启动流控，响应码509（Bandwidth Limit Exceeded），无内容。

其他如果输入错误，响应码为400；如果服务端错误，响应码为500，无内容。

5.1.9.5 响应 Content

订阅成功后会连续收到多个报文，第一个响应报文没有content。

后续报文包括编码后的视频帧数据，结构如下定义：

```
#define SDC_VENC_FRAME_I 0

#define SDC_VENC_FRAME_P 1

#define SDC_VENC_FRAME_B 2

struct sdc_venc _frame

{

uint64_t addr_phy;
```



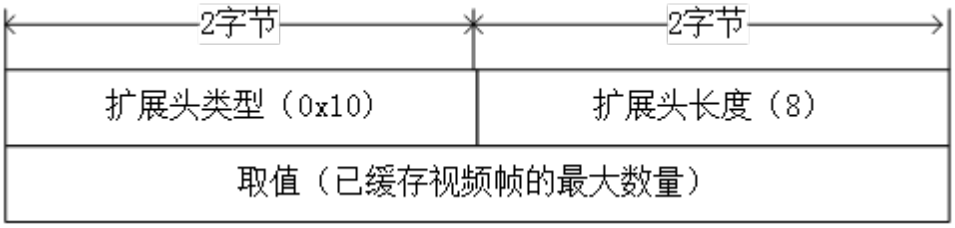
```
uint64_t addr_virt；//只读
uint64_t size;
uint32_t height;
uint32_t width;
uint32_t format; //SDC_H264 OR SDC_H265
uint32_t frame_type; //SDC_VENC__FRAME_I/P/B
uint64_tcookie[8]; //服务端调测使用
}

struct sdc_venc_data
{
uint32_t channel;
uint32_t reserve;
uint64_t frame_pts; //帧时间戳
uint64_t pts_sys; //系统时间时间戳
struct sdc_venc_frame frame;
};
```

批量订阅通道数据时，通道数量=hbtp.content_length / sizeof(struct sdc_venc_data);

注：多个报文中他们的响应头中的trans_id都会和订阅请求发送的trans_id保持一致。

5.1.9.6 响应扩展头

视频帧缓存计数扩展头(SDC_HEAD_VENC_ CACHED_COUNT)



5.1.9.7 参考样例

```
#include "sdc.h"
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>

static void display_extend_head(struct sdc_extend_head* extend_head) {}
static void display_venc_data(struct sdc_venc_data* venc_data) {}
int main(int argc,char* argv[])
{
int nret,i,fd;
char buf[1024] = { 0 };
struct sdc_common_head* head = (struct sdc_common_head*) buf;
struct sdc_extend_head* extend_head;
struct sdc_venc_data* venc_data;
uint32_t* channel;

fd = open("/mnt/srvfs/video.iaas.sdc", O_RDWR);
if(fd < 0) goto fail;

head->version = SDC_VERSION;
head->url = SDC_URL_VENC_DATA;
head->method = SDC_METHOD_GET;
head->head_length = sizeof(*head);

/** 最大缓存帧数量为10 */
extend_head = (struct sdc_extend_head*)&buf[head->head_length];
extend_head->type = SDC_HEAD_VENC_CACHED_COUNT_MAX;
extend_head->length = sizeof(*extend_head);
extend_head->reserve = 10;
head->head_length += sdc_extend_head_length(extend_head);

/** VENC订阅通道100数据 */
channel = (uint32_t*)&buf[head->head_length];
*channel = 100;
head->content_length = sizeof(*channel);

nret = write(fd, head, head->head_length + head->content_length);
if(nret < 0) goto fail;

for(;;) {
nret = read(fd, buf,sizeof(buf));
if(nret < 0) goto fail;
sdc_for_each_extend_head(head, extend_head) {
display_extend_head(extend_head);
```

```
}
for(i = 0, venc_data = (struct sdc_venc_data*)&buf[head->head_length]; i < head->content_length / sizeof(*venc_data); ++i, ++venc_data) {
display_venc_data(venc_data);
}
/** free venc_data */
head->response = head->code = 0;
head->method = SDC_METHOD_DELETE;
(void)write(fd,head,head->head_length + head->content_length); // server ignore extended headers
}
return 0;
fail:
exit(1);
}
```

5.1.10 VENC 帧数据释放

编码视频帧数据都是零拷贝传递，订阅的时候获取的视频帧都加入用户缓存视频帧的计数值，用户使用完毕之后需要及时释放，否则用户缓存视频帧的计数值达到阈值后，用户将无法再收到视频帧数据。

注：资源释放类接口都无响应，类似C++的析构函数的行为，无返回值。

5.1.10.1 请求 Common Head 方法资源

方法	URL	VALUE
DELETE	SDC_URL_VENC_DATA	0x03

5.1.10.2 请求 Content

struct sdc_venc_data;

支持批量释放，地址数量=hbtp.content_length / sizeof(struct sdc_venc_data) ;

5.1.10.3 请求扩展头

无。

5.1.10.4 响应码

无。

5.1.10.5 响应 Content

无。

5.1.10.6 响应扩展头

无

5.1.10.7 参考样例

参见VENC帧数据订阅章节的样例。

5.1.11 ITS 抓拍接口

抓拍帧分为两类，一类是外设主动触发，另一类是业务层主动触发。这两类的抓拍帧数据都需要业务层订阅抓拍通道获取，订阅的时候需要设置扩展头获取抓拍随帧参数。

注：必须在订阅抓拍通道的句柄上发送抓拍请求，否则无法收到抓拍帧数据。

5.1.11.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	SDC_URL_YUV_SNAP	0x04

5.1.11.2 请求 Content

struct sdc_portaddr {

char portname[16];

}

struct sdc_yuv_snap_param{

uint32_t id; //用户标识，用户订阅抓拍帧的时候可以获取到对应标识进行匹配

uint32_t num; //抓拍张数

uint32_t interval_msec[4]; //抓拍间隔，单位毫秒，最多可以设置4个间隔(间隔数=抓拍张数-1)。目前最多支持3张抓拍，即支持设置2个抓拍间隔

uint32_t reverse[4]; //保留字段

struct sdc_portaddr portaddr_name[0];

};

其中用户标识，参见**YUV数据订阅**一节。

5.1.11.3 请求扩展头

无。

5.1.11.4 响应码

功能正常，返回200；其他为错误码.

5.1.11.5 响应 Content

无。

5.1.11.6 响应扩展头

无。

5.1.11.7 参考样例

参见YUV帧数据订阅章节样例。

5.1.12 ITS 启用红灯增强

红灯增强可以通过外设触发，也可以由智能业务主动触发。 可以多次调用此接口，以最后一次为准。

5.1.12.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	SDC_URL_RED_LIGHT_ENHANCED	0x05

5.1.12.2 请求 Content

struct sdc_red_light_enhanced_param{

uint32_t level; //(0,100]增强系数，越大增强效果越明显

uint32_t num; //（ 0, 4]

uint32_t image_width; //原图宽（目标识别使用的图片），用来计算比例

uint32_t image_height; //原图高（目标识别使用的图片），用来计算比例

struct **5.2.1.2 请求Content** regions[0]; //区域的宽高不大于image_width/11，区域起始坐标x<=image_width-region.w, y<=image_height-region.h

};

5.1.12.3 请求扩展头

无。

5.1.12.4 响应码

功能正常，返回200；其他为错误码。

5.1.12.5 响应 Content

无。

5.1.12.6 响应扩展头

无。

5.1.12.7 参考样例

```
#include "sdc.h"
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>

int main(int argc,char* argv[])
{
char buf[1024] = { 0 };
struct sdc_common_head* head = (struct sdc_common_head*)buf;
struct sdc_red_light_enhanced_param* param = (struct
sdc_red_light_enhanced_param*)(head + 1);
int nret,fd;

fd = open("/mnt/srvfs/video.iaas.sdc", O_RDWR);
if(fd < 0) goto fail;

head->version = SDC_VERSION;
head->url = SDC_URL_RED_LIGHT_ENHANCED;
head->head_length = sizeof(*head);

if(argc >= 5) {
head->method = SDC_METHOD_CREATE;
head->content_length = sizeof(*param);

param->level = 50;
param->num = 1;
param-> image_width= 1280;
param-> image_height= 720;
param->regions[0].x = atoi(argv[1]);
param->regions[0].y = atoi(argv[2]);
param->regions[0].w = atoi(argv[3]);
param->regions[0].h = atoi(argv[4]);
}else {
head->method = SDC_METHOD_DELETE;
}

nret = write(fd, head, head->head_length + head->content_length);
if(nret < 0) goto fail;

nret = read(fd, head, sizeof(*head));
if(nret < 0 || head->code != SDC_CODE_200) goto fail;

close(fd);
return 0;
fail:
```

```
exit(1);
}
```

5.1.13 ITS 取消红灯增强

红灯增强可以通过外设触发，也可以由智能业务主动触发。

5.1.13.1 请求 Common Head 方法资源

方法	URL	VALUE
DELETE	SDC_URL_RED_LIGHT_ENHANCED	0x05

5.1.13.2 请求 Content

无。

5.1.13.3 请求扩展头

无。

5.1.13.4 响应码

功能正常，返回200；其他为错误码。

5.1.13.5 响应 Content

无。

5.1.13.6 响应扩展头

无。

5.1.13.7 参考样例

5.1.14 多通道视频流

SDC支持1拖N，允许智能业务处理其他摄像机的视频流，1拖N视频流的订阅接口和本地视频流订阅接口完全相同。

SDC APP支持1拖N取流需要满足以下条件：

- （1）SDC 设备支持1拖N。
- （2）SDC 主机WEB界面添加丛机引流成功。

SDC支持多通道设备YUV取流逻辑通道范围如下：

资源	类型	取值范围
丛机1逻辑通道	**uint32_t**	**9-10**
丛机2逻辑通道	**uint32_t**	13-14
丛机3逻辑通道	**uint32_t**	17-18
丛机4逻辑通道	**uint32_t**	22-23

5.2 codec.iaas.sdc 服务化接口定义

5.2.1 JPEG 编码

给定编码源（物理地址）和编码参数，服务端自动分配资源，返回编码后的物理地址。

注：编码通道的fd关闭后，服务端分配资源会被自动回收。

5.2.1.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	SDC_URL_ENCODED_JPEG	0x00

5.2.1.2 请求 Content

struct sdc_region {

uint32_t x;

uint32_t y;

uint32_t w;

uint32_t h;

};//硬件编码器需要进行字节对齐，x，y，w，h内部会进行对齐，以实际输出的size为准。

struct sdc_osd{

uint8_t format[128];

uint32_t reserve; //reserve为保留字段，后续扩展功能使用，尽量不要使用。

uint32_t content_length; //以wchar_t单位的 content长度/sizeof(wchar_t)

uint8_t content[3072];

};

struct sdc_osd_region {

struct sdc_region region;

struct sdc_osd osd;

};

注：format字符串格式为：prop=value，多个prop之间以';'间隔。支持的prop(忽略大小写)如下：

fgColor=0xRRGGBB

bgColor=0xRRGGBB

fgAlpha=0~128 //将自动向系统支持的数字调整（后续版本会调整到0-100）

bgAlpha=0~128 //将自动向系统支持的数字调整（后续版本会调整到0-100）

fontSize=0~8 // 0（16x16），1（24x24），2（32x32），3（48x48），4（64x64），5（72x72），6（88x88），7（96x96），8（128x128）

code=utf-16 //当前仅支持utf-16，这也是不携带code时的缺省值。

struct sdc_encode_jpeg_param

{

uint16_t qf; //1-99，0则使用默认编码质量60。

uint16_t osd_region_cnt;

uint32_t reserve; // 需设置为0，扩展保留标志位 。

struct sdc_region region;

struct sdc_yuv_frame frame;

struct sdc_osd_region osd_regions[0];

};

注：

1）支持批量编码，content为多个sdc_encode_jpeg_param。

2）用户希望在图像之外叠加OSD,请使用图像合成接口。

3）如果传的参数为0则要以16进制格式传0x0

5.2.1.3 请求扩展头

无。

5.2.1.4 响应码

如果功能正常，响应码200，

如果启动流控，响应码509（Bandwidth Limit Exceeded），无内容。

其他为错误码，无内容。

5.2.1.5 响应 Content

```
struct sdc_jpeg_frame
{
uint64_t addr_phy; //数据存放物理地址
uint64_t addr_virt; //数据存放虚拟地址，默认cache映射
uint32_t size; //数据内存长度
uint32_t reserve;
uint32_tcookie[4]; //服务业务功能或者调测需要
};
```

批量编码返回数组，数组长度=hbtp.content_length / sizeof(struct sdc_jpeg_frame)

5.2.1.6 响应扩展头

无。

5.2.1.7 参考样例

```
#include "sdc.h"
#include <errno.h>
#include <sys/uio.h>

int yuv_2_jpeg(int fd ,const struct sdc_yuv_frame* yuv_frame,const struct
sdc_osd_region* osd_region, struct sdc_jpeg_frame* jpeg_frame)
{
struct sdc_encode_jpeg_param param = {
.qf = 90,
.osd_region_cnt = 1,
.region = { 0, 0, yuv_frame->width, yuv_frame->height },
.frame = *yuv_frame,
};
struct sdc_common_head head = {
.version = SDC_VERSION, //0x5331
.url = SDC_URL_ENCODED_JPEG, //0x00
.method = SDC_METHOD_CREATE,
.head_length = sizeof(head),
.content_length = sizeof(param) + sizeof(*osd_region),
};
struct iovec iov[] = {
{.iov_base = &head, .iov_len = sizeof(head) },
{.iov_base = &param, .iov_len = sizeof(param) },
{.iov_base = (void*)osd_region, .iov_len = sizeof(*osd_region) }
};

int nret;

nret = writev(fd,iov,sizeof(iov)/sizeof(iov[0]));
if(nret < 0) return errno;

iov[1].iov_base = jpeg_frame;
iov[1].iov_len = sizeof(*jpeg_frame);
nret = readv(fd,iov,2);
if(nret < 0) return errno;

if(head.head_length != sizeof(head) || head.content_length !=
sizeof(*jpeg_frame)) return EIO;

return 0;
}
```

5.2.2 JPEG 编码资源释放

智能业务给定订阅的逻辑通道号，则会收到此逻辑通道主动上报的所有YUV数据。

注：资源释放类接口都无响应，类似C++的析构函数的行为，无返回值。

5.2.2.1 请求 Common Head 方法资源

方法	URL	VALUE
DELETE	SDC_URL_ENCODED_JPEG	0x00

5.2.2.2 请求 Content

struct sdc_jpeg_frame;

批量解码资源释放，传递数组，数组长度=hbtp.content_length / sizeof(structsdc_jpeg_frame)。

5.2.2.3 请求扩展头

无。

5.2.2.4 响应码

无。

5.2.2.5 响应 Content

无。

5.2.2.6 响应扩展头

无。

5.2.2.7 参考样例

```
#include "sdc.h"
#include <sys/uio.h>
void yuv_2_jpeg_free(int fd ,const struct sdc_jpeg_frame* jpeg_frame)
{
static const struct sdc_common_head head = {
.version = SDC_VERSION, //0x5331
.url = SDC_URL_ENCODED_JPEG, //0x00
.method = SDC_METHOD_DELETE,
.head_length = sizeof(head),
.content_length = sizeof(*jpeg_frame),
};
struct iovec iov[] = {
{.iov_base = (void*)&head, .iov_len = sizeof(head) },
{.iov_base = (void*)jpeg_frame, .iov_len = sizeof(*jpeg_frame) }
};
(void)writev(fd,iov,sizeof(iov)/sizeof(iov[0]));
}
```

5.2.3 JPEG 解码

给定解码模式和待解码数据源，服务端申请解码后的资源，解码后返回。

注：解码通道fd关闭后，服务端申请的资源会被自动回收。

5.2.3.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	SDC_URL_DECODED_YUV	0x01

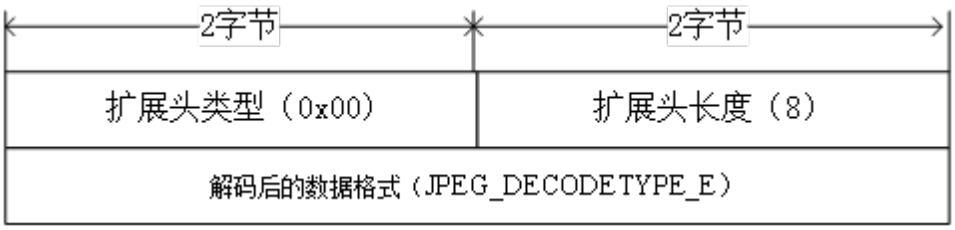
5.2.3.2 请求 Content

struct sdc_jpeg_frame;

支持批量解码，解码数量=hbtp.content_length / sizeof(struct sdc_jpeg_frame);

5.2.3.3 请求扩展头

请求扩展头 SDC_HEAD_DECODED_YUV_ACCEPT_TYPE



#define YVU_420SP0

#define BGR888_PLANAR10

#define RGB888_PLANAR11

5.2.3.4 响应码

如果功能正常，响应码200，

如果启动流控，响应码509（ Bandwidth Limit Exceeded ），无内容。

其他为错误码，无内容。

5.2.3.5 响应 Content

struct sdc_yuv_frame;

批量解码，返回数组，数组长度=hbtp.content_length / sizeof(struct sdc_yuv_frame);

5.2.3.6 响应扩展头

无。

5.2.3.7 参考样例

```
#include "sdc.h"
#include <errno.h>
#include <sys/uio.h>

int jpeg_2_yuv(int fd ,const struct sdc_jpeg_frame* jpeg_frame, int format,struct sdc_yuv_frame* yuv_frame)
{
struct sdc_extend_head extend_head = {
.type = SDC_HEAD_DECODED_YUV_ACCEPT_TYPE,
.length = sizeof(extend_head),
.reserve = format,
};
struct sdc_common_head head = {
.version = SDC_VERSION, //0x5331
.url = SDC_URL_DECODED_YUV, //0x01
.method = SDC_METHOD_CREATE,
.head_length = sizeof(head) + sizeof(extend_head),
.content_length = sizeof(*jpeg_frame),
};
struct iovec iov[] = {
{.iov_base = (void*)&head, .iov_len = sizeof(head) },
{.iov_base = &extend_head, .iov_len = sizeof(extend_head) },
{.iov_base = (void*)jpeg_frame, .iov_len = sizeof(*jpeg_frame) }
};

int nret;

nret = writev(fd,iov,sizeof(iov)/sizeof(iov[0]));
if(nret < 0) return errno;

iov[1].iov_base = yuv_frame;
iov[1].iov_len = sizeof(*yuv_frame);
nret = readv(fd,iov,2);
if(nret < 0) return errno;

if(head.head_length != sizeof(head) || head.content_length != sizeof(*yuv_frame)) return EIO;

return 0;
}
```

5.2.4 JPEG 解码资源释放

编码结果所需内存由服务端申请，需要客户端使用完毕后释放。

须知

资源释放类接口都无响应，类似C++的析构函数的行为，无返回值。

5.2.4.1 请求 Common Head 方法资源

方法	URL	VALUE
DELETE	SDC_URL_DECODED_YUV	0x01

5.2.4.2 请求 Content

struct sdc_yuv_frame;

支持批量释放，释放数量=hbtp.content_length / sizeof(struct sdc_yuv_frame);

5.2.4.3 请求扩展头

无。

5.2.4.4 响应码

无。

5.2.4.5 响应 Content

无。

5.2.4.6 响应扩展头

无。

5.2.4.7 参考样例

```
#include "sdc.h"
#include <sys/uio.h>
void jpeg_2_yuv_free(int fd ,const struct sdc_yuv_frame* yuv_frame)
{
struct sdc_common_head head = {
.version = SDC_VERSION, //0x5331
.url = SDC_URL_DECODED_YUV, //0x01
.method = SDC_METHOD_DELETE,
.head_length = sizeof(head),
.content_length = sizeof(*yuv_frame),
};
struct iovec iov[] = {
{.iov_base = (void*)&head, .iov_len = sizeof(head) },
{.iov_base = (void*)yuv_frame, .iov_len = sizeof(*yuv_frame) }
};
(void)writev(fd,iov,sizeof(iov)/sizeof(iov[0]));
}
```

5.2.5 获取 OSD 显示区域高度

输入OSD，返回OSD显示区域的高度。

合成图片的某些场景中，需要在图片区域之外呈现OSD，这样实际是修改了原始图片的高度和宽度。需要利用此接口获取OSD显示区域的高度来计算出最终合成图片高度和宽度，传递给合成接口使用。

5.2.5.1 请求 Common Head 方法资源

方法	URL	VALUE
GET	SDC_URL_OSD_BOX_HEIGHT	0x02

5.2.5.2 请求 Content

struct sdc_osd_box{

uint32_t text_box_width;

struct sdc_osd osd;

};

多个输入源数量=hbtp.content_length / sizeof(struct sdc_osd_box);

5.2.5.3 请求扩展头

无

5.2.5.4 响应码

如果功能正常，响应码200，如果启动流控，响应码403.（Forbidden），无内容；其他为错误码，无内容。

5.2.5.5 响应 Content

uint32_t height;

批量操作，返回数组，数组长度= hbtp.content_length / sizeof(uint32_t)

5.2.5.6 响应扩展头

无。

5.2.5.7 参考样例

```
#include "sdc.h"
#include <stddef.h>
#include <errno.h>
#include <sys/uio.h>

int osd_height(int fd ,const struct sdc_osd* osd, uint32_t width)
{
struct sdc_common_head head = {
.version = SDC_VERSION, //0x5331
.url = SDC_URL_OSD_BOX_HEIGHT, //0x02
.method = SDC_METHOD_GET,
.head_length = sizeof(head),
.content_length = sizeof(struct sdc_osd_box),
};
struct iovec iov[] = {
{.iov_base = (void*)&head, .iov_len = sizeof(head) },
/**
* 利用iovec来拼装一个数据结构的时候要特别注意是否因为字段对齐隐藏的填充字符
* 现有服务化接口的定义中消除了各种隐式的填充字符
*/
{.iov_base = &width, .iov_len = sizeof(width) /** == offsetof(struct sdc_osd_box,osd) */ },
{.iov_base = (void*)osd, .iov_len = sizeof(*osd) /** == sizeof(struct sdc_osd_box) - offsetof(struct sdc_osd_box,osd) */}
};

int nret;

nret = writev(fd,iov,sizeof(iov)/sizeof(iov[0]));
if(nret < 0) return errno;

nret = readv(fd,iov,2);
if(nret < 0) return errno;

if(head.head_length != sizeof(head) || head.content_length != sizeof(width)) return EIO;

return width;
}
```

5.2.6 JPEG 合成

输入待合成的资源和合成模式，服务端自动分配合成后的资源，完成合成后返回。

注：合成通道的fd关闭后，服务端分配的资源会被自动回收。

5.2.6.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	SDC_URL_COMBINED_IMAGE	0x03

5.2.6.2 请求 Content

struct sdc_combined_yuv

{

struct sdc_region origin_region; //抠图区域，如果无需抠图，应该是整图区域[0,0,w,h]

struct sdc_region combined_region; //在合成图中显示的区域，和origin_region大小（即w/h不全等）不等表示需要进行缩放处理

struct sdc_yuv_frame frame;

}

struct sdc_combined_yuv_param{

uint32_t width;

uint32_t height;

uint32_t yuv_cnt;

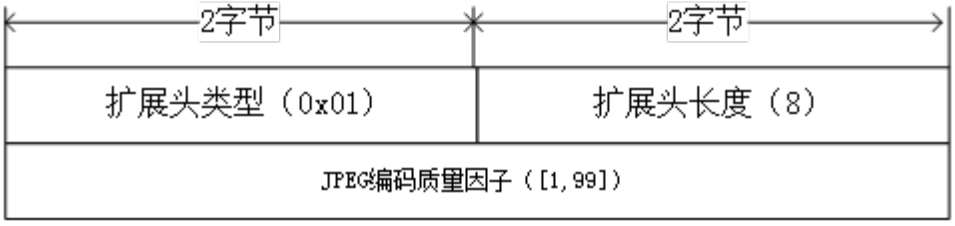
uint32_t reserve;

struct sdc_combined_yuv yuv[0];

}

5.2.6.3 请求扩展头

设置JPEG编码质量因子的扩展头(SDC_HEAD_COMBINED_JPEG_QF)



如果无此扩展头，说明无需编码，直接返回合成后的原图。

设置合成OSD的扩展头(SDC_HEAD_COMBINED_OSD)

真实OSD的数量 = (header.length – 8) / sizeof(struct osd_region)

这里osd_region中的region均是指合成图片内的region定义。

注：如果用户需要在图片之外叠加OSD，则应该事先获取osd显示区域的高度信息，设置正确的合成图片的高度和叠加osd的region。

5.2.6.4 响应码

如果功能正常，响应码200，如果启动流控，响应码509（ Bandwidth Limit Exceeded ），无内容；其他为错误码，无内容。

5.2.6.5 响应 Content

如果内容扩展头格式为原图，则content的内容格式为

struct sdc_yuv_frame;

如果内容扩展头格式为JPEG，则content的内容格式为

struct sdc_jpeg_frame;

5.2.6.6 响应扩展头

响应内容格式扩展头(SDC_HEAD_COMBINED_ CONTENT_TYPE)

指示内容格式。如果请求头中未携带JPEG编码质量因子的扩展头，则内容格式为原图，否则为JPEG. 响应的时候无条件带上这个扩展头指示内容格式。

5.2.6.7 参考样例

```
#include "sdc.h"
#include <errno.h>
#include <sys/uio.h>

int combined_jpeg(int fd, const struct sdc_yuv_frame frames[4], const struct sdc_osd* osd,const struct sdc_resolution* combined_resolution, struct sdc_jpeg_frame* jpeg)
{
int nret;
/** 合成后图片分辨率由combined_resolution决定 */
struct sdc_combined_yuv_param param = {
.width = combined_resolution->width,
.height = combined_resolution->height,
.yuv_cnt = 4,
};
struct sdc_combined_yuv combined_yuv[4] = { { 0 } };

/** 合成后编码，编码的质量因子为 80 */
struct sdc_extend_head jpeg_head = { SDC_HEAD_COMBINED_JPEG_QF, sizeof(jpeg_head), 80 };

/** 合成后需要叠加传入的osd , sizeof(struct sdc_osd_region)本身按8字节对齐的 */
struct sdc_extend_head osd_head = { SDC_HEAD_COMBINED_OSD, sizeof(struct sdc_osd_region) + sizeof(osd_head) };

/** 叠加osd的扩展头包含 struct sdc_osd_region结构，拆分为两个独立成员组合发送 ，一个是region，一个是输入参数 osd */
struct sdc_region region = { 0, 0, combined_resolution->width, combined_resolution->height }; // for osd

/** 头部信息设置正确的head_length和content_length */
struct sdc_common_head head = {
.version = SDC_VERSION,
.url = SDC_URL_COMBINED_IMAGE,
.method = SDC_METHOD_CREATE,
.head_length = sizeof(head) + jpeg_head.length + osd_head.length,
.content_length = sizeof(param) + sizeof(combined_yuv),
};

/** 以上数据组合发送 */
struct iovec iov[] = {
{.iov_base = &head, .iov_len = sizeof(head) },
{.iov_base = &jpeg_head, .iov_len = sizeof(jpeg_head) },
{.iov_base = &osd_head, .iov_len = sizeof(osd_head) },
{.iov_base = &region, .iov_len = sizeof(region) },
```



```
{.iov_base = (void*)osd, .iov_len = sizeof(*osd) },
{.iov_base = &param, .iov_len = sizeof(param) },
{.iov_base = combined_yuv, .iov_len = sizeof(combined_yuv) },
};
/** 输入的4张图片各自占用合成图片的1/4: 左上，右上，左下，右下, 原始图片不做任何切割 */
combined_yuv[0].frame = frames[0];
combined_yuv[0].origin_region.w = frames[0].width;
combined_yuv[0].origin_region.h = frames[0].height;
combined_yuv[0].combined_region.w = combined_resolution->width / 2;
combined_yuv[0].combined_region.h = combined_resolution->height / 2;

combined_yuv[1].frame = frames[1];
combined_yuv[1].origin_region.w = frames[1].width;
combined_yuv[1].origin_region.h = frames[1].height;
combined_yuv[1].combined_region.x = combined_yuv[1].combined_region.w = combined_resolution->width / 2;
combined_yuv[1].combined_region.h = combined_resolution->height / 2;

combined_yuv[2].frame = frames[2];
combined_yuv[2].origin_region.w = frames[2].width;
combined_yuv[2].origin_region.h = frames[2].height;
combined_yuv[2].combined_region.w = combined_resolution->width / 2;
combined_yuv[2].combined_region.y = combined_yuv[2].combined_region.h = combined_resolution->height / 2;

combined_yuv[3].frame = frames[3];
combined_yuv[3].origin_region.w = frames[3].width;
combined_yuv[3].origin_region.h = frames[3].height;
combined_yuv[1].combined_region.x = combined_yuv[1].combined_region.w = combined_resolution->width / 2;
combined_yuv[3].combined_region.y = combined_yuv[3].combined_region.h = combined_resolution->height / 2;

nret = writev(fd,iov,sizeof(iov) / sizeof(iov[0]));
if(nret < 0) return errno;

iov[2].iov_base = jpeg;
iov[2].iov_len = sizeof(*jpeg);
nret = readv(fd,iov,3);
if(nret < 0) return errno;

if(head.code != SDC_CODE_200 || head.head_length != sizeof(head) + sizeof(jpeg_head) || head.content_length != sizeof(*jpeg)) return EIO;
return 0;
```



```
}
```

5.2.7 JPEG 合成资源释放

释放合成功能创建的资源。

须知

资源释放类接口都无响应，类似C++的析构函数的行为，无返回值。

5.2.7.1 请求 Common Head 方法资源

方法	URL	VALUE
DELETE	SDC_URL_COMBINED_IMAGE	0x03

5.2.7.2 请求 Content

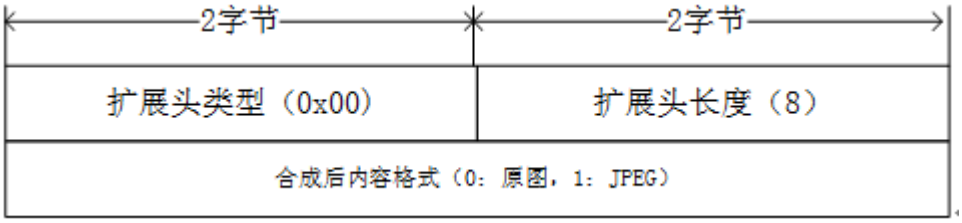
如果内容扩展头格式为原图，则content的内容格式为struct sdc_yuv_frame

支持批量释放，批量释放是的数组大小=hbtp.content_length / sizeof(struct sdc_yuv_frame);

如果内容扩展头格式为JPEG，则content的内容格式为struct sdc_jpeg_frame;

支持批量释放，批量释放是的数组大小=hbtp.content_length / sizeof(struct sdc_jpeg_frame);

5.2.7.3 请求扩展头 (SDC_HEAD_COMBINE _CONTENT_TYPE)



释放时必须携带此扩展头指示内容格式。如果不携带此扩展头，write操作会失败，errno为EINVAL。

5.2.7.4 响应码

无。

5.2.7.5 响应 Content

无。

5.2.7.6 响应扩展头

无。

5.2.7.7 参考样例

```
#include "sdc.h"
#include <sys/uio.h>

void combined_jpeg_free(int fd, const struct sdc_jpeg_frame* jpeg)
{
/** 指定释放数据的类型为jpeg */
struct sdc_extend_head jpeg_head = { SDC_HEAD_COMBINED_CONTENT_TYPE, sizeof(jpeg_head), 1 };

/** 头部信息设置正确的head_length和content_length */
static const struct sdc_common_head head = {
.version = SDC_VERSION,
.url = SDC_URL_COMBINED_IMAGE,
.method = SDC_METHOD_DELETE,
.head_length = sizeof(head) + sizeof(jpeg_head),
.content_length = sizeof(*jpeg),
};

/** 以上数据组合发送 */
struct iovec iov[] = {
{.iov_base = (void*)&head, .iov_len = sizeof(head) },
{.iov_base = &jpeg_head, .iov_len = sizeof(jpeg_head) },
{.iov_base = (void*)jpeg, .iov_len = sizeof(*jpeg) },
};

(void)writev(fd,iov,sizeof(iov) / sizeof(iov[0]));
}
```

5.3 utils.iaas.sdc 服务化接口定义

5.3.1 获取硬件标识

获取SDC唯一且不变的硬件标识，用于License等控制。

5.3.1.1 请求 Common Head 方法资源

方法	URL	VALUE
GET	SDC_URL_HARDWARE_ID	100

5.3.1.2 请求 Content

无。

5.3.1.3 请求扩展头

无。

5.3.1.4 响应码

如果功能正常，响应码200， 其他为错误码。

5.3.1.5 响应 Content

struct sdc_hardware_id

{

char id[33];

};

硬件标识固定为32个字符，响应中在32个字符之后包括了NULL字符。

5.3.1.6 响应扩展头

无。

5.3.1.7 参考样例

```
#include "sdc.h"
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/uio.h>

int main(int argc,char* argv[])
{
struct sdc_common_head head = {
.version = SDC_VERSION,
.url = SDC_URL_HARDWARE_ID,
.method = SDC_METHOD_GET,
.head_length = sizeof(head),
};

struct sdc_hardware_id id;

struct iovec iov[] = {
{ &head, sizeof(head) },
{ &id, sizeof(id) },
};
int nret,fd;

fd = open("/mnt/srvfs/utils.iaas.sdc", O_RDWR);
if(fd < 0) goto fail;

nret = write(fd, &head, sizeof(head));
if(nret < 0) goto fail;

nret = readv(fd, iov, sizeof(iov) / sizeof(iov[0]));
if(nret < 0 || head.code != SDC_CODE_200 || head.head_length != sizeof(head) ||
head.content_length != sizeof(id)) goto fail;

printf("hardware_id: %s\n", id.id);

close(fd);
return 0;
fail:
exit(1);
}
```

5.3.2 申请连续物理内存

注：同海思MMZ内存。

摄像机中大量的视频、图片等处理都需要申请物理内存在多个进程之间进行零拷贝的传递保证性能。基于服务化接口申请的物理内存，在客户端的句柄无效时服务端会自动回收所有内存。这样可以保证客户端异常退出后，物理资源不会泄露。

5.3.2.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	SDC_URL_MMZ	101

5.3.2.2 请求 Content

uint32_t size;

批量申请，申请count = hbtp.content_length / sizeof(uint32_t);

5.3.2.3 请求扩展头

无。

5.3.2.4 响应码

如果功能正常，响应码200， 其他为错误码。

如果启动流控，响应码509（ Bandwidth Limit Exceeded ），无内容。

5.3.2.5 响应 Content

struct sdc_mmz

{

uint64_t addr_phy;

uint64_t addr_virt;

uint32_t size;

uint32_t reserve;

uint32_t cookie[4];

}

5.3.2.6 响应扩展头

无。

5.3.2.7 参考样例

```
#include "sdc.h"
#include <errno.h>
#include <sys/uio.h>

int mmz_alloc_cached(int fd, uint32_t size, struct sdc_mmz* mmz)
{
struct sdc_common_head head = {
.version = SDC_VERSION,
.url = SDC_URL_MMZ,
.method = SDC_METHOD_CREATE,
.head_length = sizeof(head),
.content_length = sizeof(size),
};

struct iovec iov[] = {
{ (void*)&head, sizeof(head) },
{ &size, sizeof(size) }
};

int nret = writev(fd, iov, sizeof(iov)/sizeof(iov[0]));
if(nret < 0) return errno;

iov[1].iov_base = mmz;
iov[1].iov_len = sizeof(*mmz);

nret = readv(fd, iov,2);
if(nret < 0) return errno;

if(head.code != SDC_CODE_200 || head.head_length != sizeof(head) ||
head.content_length != sizeof(*mmz)) return EIO;
return 0;
}
```

5.3.3 释放物理内存空间

注：同海思MMZ内存。

资源释放无响应。

5.3.3.1 请求 Common Head 方法资源

方法	URL	VALUE
DELETE	SDC_URL_MMZ	101

5.3.3.2 请求 Content

struct sdc_mmz mem;

支持批量操作。 count = hbtp.content_length / sizeof(struct sdc_mmz);

5.3.3.3 请求扩展头

无。

5.3.3.4 响应码

无。

5.3.3.5 响应 Content

无。

5.3.3.6 响应扩展头

无。

5.3.3.7 参考样例

```
#include "sdc.h"
#include <errno.h>
#include <sys/uio.h>

void mmz_free(int fd, struct sdc_mmz* mmz)
{
struct sdc_common_head head = {
.version = SDC_VERSION,
.url = SDC_URL_MMZ,
.method = SDC_METHOD_DELETE,
.head_length = sizeof(head),
.content_length = sizeof(*mmz),
};

struct iovec iov[] = {
{ (void*)&head, sizeof(head) },
{ mmz, sizeof(*mmz) }
};

(void)writev(fd, iov, sizeof(iov)/sizeof(iov[0]));
}
```


5.3.4 打开透明通道

打开并订阅透明通道数据。该接口为异步接口，订阅后当透明通道有数据可读时，上报给客户端。该接口依赖于web扩展接口服务，必须将相应485口设置成透明通道模式，调用此接口才可成功。

5.3.4.1 请求 Common Head 方法资源

方法	URL	VALUE
GET	SDC_URI_TRANSPORT_ DATA	81

5.3.4.2 请求 Content

struct sdc_portaddr

{

char portname[16];

}

485串口号。如RS485-1、RS485-2。

5.3.4.3 请求扩展头

无。

5.3.4.4 响应码

若订阅成功，响应码为200，若失败，响应码为400。若485口index错误或者web没有开启该端口的透明通道模式，返回404。若该端口已被其他客户端占用，返回403。

5.3.4.5 响应 Content

uint8_t readData[];

订阅数据后，当透明通道有数据可读时，上报数据。Contentlength代表数据长度

5.3.4.6 响应扩展头

无。

5.3.4.7 参考样例

5.3.5 透明通道发送数据

向占用的透明通道发送数据。

5.3.5.1 请求 Common Head 方法资源

方法	URL	VALUE
UPDATE	SDC_URI_ TRANSPORT _DATA	81

5.3.5.2 请求 Content

uint8_t extDevIn[];

需要发送的字符串。Contentlength代表数据长度

5.3.5.3 请求扩展头

无。

5.3.5.4 响应码

若发送数据成功，响应码为200，若失败，响应码为400。

5.3.5.5 响应 Content

无。

5.3.5.6 响应扩展头

无。

5.3.5.7 参考样例

5.3.6 获取 UTC 时间

获取UTC时间。

5.3.6.1 请求 Common Head 方法资源

方法	URL	VALUE
GET	SSP_CFG_E_URI_UTC_TIME	140

5.3.6.2 请求 Content

无。

5.3.6.3 请求扩展头

无。

5.3.6.4 响应码

若发送数据成功，响应码为200，若失败，响应码为400。

5.3.6.5 响应 Content

```
struct timeval
{
uint64_t tv_sec; // 秒数
uint64_t tv_usec; // 微秒数
}
```

5.3.6.6 响应扩展头

无。

5.3.6.7 参考样例

```
#include "sdc.h"
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/uio.h>
int main(int argc, char *argv[])
{
struct sdc_common_head head;
head.version = SDC_VERSION;
head.url = 140;
head.method = SDC_METHOD_GET;
head.head_length = sizeof(head);
struct timeval tv;
struct iovec iov[] = {
{&head, sizeof(head)},
{&tv, sizeof(tv)},
};
int nret, fd;
fd = open("/mnt/srvfs/utils.iaas.sdc", O_RDWR);
if (fd < 0) goto fail;
nret = write(fd, &head, sizeof(head));
if (nret < 0) goto fail;
nret = readv(fd, iov, sizeof(iov) / sizeof(iov[0]));
if (nret < 0 || head.code != SDC_CODE_200)
{
printf("nret:%d\n", nret);
printf("headCode:%d\n", head.code);
goto fail;
}
printf("tv_sec: %lu, tv_usec:%lu\n", tv.tv_sec, tv.tv_usec);
close(fd);
return 0;
fail:
exit(1);
}
```

5.4 algorithm.iaas.sdc 服务化接口定义

5.4.1 NNIE 模型创建

基于输入的WK文件创建可执行的模型。

5.4.1.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	SDC_URL_NNIE_MODEL	0

5.4.1.2 请求 Content

如果扩展头SDC_HEAD_NNIE_MODEL_CONTENT_TYPE为0或未定义扩展头，则内容为MMZ：struct sdc_mmz;

如果扩展头SDC_HEAD_NNIE_MODEL_CONTENT_TYPE为1，则内容为文件名：char filename[]

hbtp.content_length指明文件名长度，所以这里文件名是否以NULL字符结束不影响功能。

5.4.1.3 请求扩展头

5.4.1.3.1 模型内容输入方式（SDC_HEAD_NNIE_MODEL_CONTENT_TYPE）。

如果没有扩展头，等同于MMZ输入。

#define NNIE_MODEL_CONTENT_MMZ 0

#define NNIE_MODEL_CONTENT_FILE 1

5.4.1.4 响应码

如果功能正常，响应码200， 其他为错误码。

5.4.1.5 响应 Content

SVP_NNIE_MODEL_S model;

5.4.1.6 响应扩展头

无。

5.4.1.7 参考样例

```
int SDC_LoadModel(unsigned int uiLoadMode, char *pucModelFileName,
SVP_NNIE_MODEL_S *pstModel)
{
int s32Ret = 0;
int ret = 0;
int u32TotalSize = 0;

struct sdc_extend_head* extend_head;
char buf[1024] = {0};
struct sdc_common_head *phead = (struct sdc_common_head *)buf;

unsigned int uFileSize;
struct sdc_mmz stMmzAddr;

if ((NULL == pstModel) || (NULL == pucModelFileName))
{
fprintf(stdout,"Err in SDC_LoadModel, pstModel or pucModelFileName is null
\n");
return -1;
}

fprintf(stdout,"Load model, pucModelFileName:%s!\n", pucModelFileName);

struct sdc_common_head head;
struct rsp_strcut {
struct sdc_common_head head;
SVP_NNIE_MODEL_S model;
}rsp_strcut_tmp;

struct iovec iov[2] = {
[0] = { .iov_base = buf, .iov_len = sizeof(struct sdc_common_head) +
sizeof(struct sdc_extend_head)},
[1] = { .iov_len = MAX_MODULE_PATH}
};
//memset(&head, 0, sizeof(head));
phead->version = SDC_VERSION;
phead->url = SDC_URL_NNIE_MODEL;
phead->method = SDC_METHOD_CREATE;
phead->head_length = sizeof(struct sdc_common_head);
phead->content_length = MAX_MODULE_PATH;

/*模式0，不带扩展头，默认内存方式加载*/
if (uiLoadMode == 0)
{
```



```
FILE *fp = fopen(pucModelFileName, "rb");
if(fp == NULL)
{
fprintf(stdout,"modelfile fopen %s fail!\n", pucModelFileName);
return -1;
}
ret = fseek(fp,0L,SEEK_END);
if(ret != 0)
{
fprintf(stdout,"check nnie file SEEK_END, fseek fail.");
fclose(fp);
return -1;
}
uFileSize = ftell(fp);
ret = fseek(fp,0L,SEEK_SET);
if(0 != ret)
{
fprintf(stdout,"check nnie file SEEK_SET, fseek fail.");
fclose(fp);
return -1;
}
stMmzAddr.size = uFileSize;
ret = SDC_MmzAlloc(uFileSize, 0, &stMmzAddr); // param 2: 0 no cache, 1 cache
if(ret != stMmzAddr.size)
{
fprintf(stdout,"SDC_MmzAlloc ret %d, readsize %d", ret, stMmzAddr.size);
return -1;
}
ret = fread((HI_VOID*)(uintptr_t)stMmzAddr.addr_virt, 1, stMmzAddr.size, fp);
if(ret != stMmzAddr.size)
{
fprintf(stdout,"filesize %d, readsize %d", ret, stMmzAddr.size);
return -1;
}
/*用户执行调用算法程序对传入文件进行解码*/
if(SDC_ModelDecript(&stMmzAddr))
{
fprintf(stdout,"SDC_ModelDecript Fail!",);
```



```
return -1;
}
iov[1].iov_base = &stMmzAddr;
iov[0].iov_len = sizeof(struct sdc_common_head);
}
else if (uiLoadMode == 1)/*模式1，带扩展头，扩展头参数指定为内存方式加载*/
{
FILE *fp = fopen(pucModelFileName, "rb");
if(fp == NULL)
{
fprintf(stdout,"modelfile fopen %s fail!\n", pucModelFileName);
return -1;
}
ret = fseek(fp,0L,SEEK_END);
if(ret != 0)
{
fprintf(stdout,"check nnie file SEEK_END, fseek fail.");
fclose(fp);
return -1;
}
uFileSize = ftell(fp);
ret = fseek(fp,0L,SEEK_SET);
if(0 != ret)
{
fprintf(stdout,"check nnie file SEEK_SET, fseek fail.");
fclose(fp);
return -1;
}
stMmzAddr.size = uFileSize;
ret = SDC_MmzAlloc(uFileSize, 0, &stMmzAddr); // param 2: 0 no cache, 1 cache
if(ret != stMmzAddr.size)
{
fprintf(stdout,"SDC_MmzAlloc ret %d, readsize %d", ret, stMmzAddr.size);
return -1;
}
ret = fread((HI_VOID*)(uintptr_t)stMmzAddr.addr_virt, 1, stMmzAddr.size, fp);
if(ret != stMmzAddr.size)
{
fprintf(stdout,"filesize %d, readsize %d", ret, stMmzAddr.size);
```



```
return -1;
}
/*用户执行调用算法程序对传入文件进行解码*/
if(SDC_ModelDecript(&stMmzAddr))
{
fprintf(stdout,"SDC_ModelDecript Fail!",);
return -1;
}
extend_head = (struct sdc_extend_head*)&buf[phead->head_length];
extend_head->type = 1;//NNIE_NNIE_MODEL_OP
extend_head->length = sizeof(*extend_head);
extend_head->reserve = 0;/*0或者不带是内存方式，1是文件名方式*/
phead->head_length += sizeof(struct sdc_extend_head);
iov[1].iov_base = &stMmzAddr;
}
else /*模式2，带扩展头，扩展头参数指定为文件名方式加载*/
{
extend_head = (struct sdc_extend_head*)&buf[phead->head_length];
extend_head->type = 1;//NNIE_NNIE_MODEL_OP
extend_head->length = sizeof(*extend_head);
extend_head->reserve = 1;/*0或者不带是内存方式，1是文件名方式*/
phead->head_length += sizeof(struct sdc_extend_head);
iov[1].iov_base = pucModelFileName;//pcModelName;
}
s32Ret = writev(fd_algorithm, iov, 2);
if (s32Ret < 0)
{
fprintf(stdout,"creat nnie,write to algorithm.iaas.sdc fail: %m\n");
}
/*模型加载后立即释放*/
if (uiLoadMode < 2)mmz_free(fd_config, &stMmzAddr);
s32Ret = read(fd_algorithm, &rsp_strcut_tmp, sizeof(rsp_strcut_tmp));
if(s32Ret == -1)
{
fprintf(stdout,"get_channel_data fail: %m\n");
return -1;
}
if(s32Ret > sizeof(rsp_strcut_tmp))
```



```
{
fprintf(stdout,"get_channel_data truncated, data len: %d > %zu\n", s32Ret,
sizeof(rsp_strcut_tmp));
return -1;
}
if (s32Ret < 0 || rsp_strcut_tmp.head.code != SDC_CODE_200 ||
rsp_strcut_tmp.head.content_length <= 0)
{
fprintf(stdout,"get nnie create response, read from algorithm.iaas.sdc
fail,s32Ret:%ld, code=%d,length=%d\n",
s32Ret, rsp_strcut_tmp.head.code, rsp_strcut_tmp.head.content_length);
}
else
{
s_stSsdModel.stModel = rsp_strcut_tmp.model;
memcpy(pstModel, &rsp_strcut_tmp.model,sizeof(SVP_NNIE_MODEL_S));
}
return s32Ret;
}
```

5.4.2 NNIE 模型删除

释放WK文件创建可执行的模型。

须知

资源释放类接口都无响应，类似C++的析构函数的行为，无返回值。

5.4.2.1 请求 Common Head 方法资源

方法	URL	VALUE
DELETE	SDC_URL_NNIE_MODEL	0

5.4.2.2 请求 Content

SVP_NNIE_MODEL_S

5.4.2.3 请求扩展头

无。

5.4.2.4 响应码

无。

5.4.2.5 响应 Content

无。

5.4.2.6 响应扩展头

无。

5.4.2.7 参考样例

```
int SDC_UnLoadModel(SVP_NNIE_MODEL_S *pstModel)
{
int nRet = -1;
if (NULL != pstModel)
{
struct sdc_common_head head;
struct iovec iov[2] = {
[0] = {.iov_base = &head , .iov_len = sizeof(head)},
[1] = {.iov_base = pstModel, .iov_len = sizeof(SVP_NNIE_MODEL_S)}
};
// fill head struct
memset(&head, 0, sizeof(head));
head.version = SDC_VERSION;
head.url = SDC_URL_NNIE_MODEL;
head.method = SDC_METHOD_DELETE;
head.head_length = sizeof(head);
head.content_length = sizeof(SVP_NNIE_MODEL_S);
nRet = writev(fd_algorithm, iov, sizeof(iov)/sizeof(iov[0]));
if (nRet < 0)
{
fprintf(stdout,"Errin SDC_UnLoadModel:failed to unload nnie module!\n");
}
}
else
{
fprintf(stdout,"Err in SDC_UnLoadModel:module pointer is NULL!\n");
}
return 0;
}
```

5.4.3 NNIE Forward

多节点输入输出的CNN类型网络预测。等同于将海思的HI_MPI_SVP_NNIE_GetTskBufSize,HI_MPI_SVP_NNIE_Forward和HI_MPI_SVP_NNIE_Query接口合一了，后者就是响应。

5.4.3.1 请求 Common Head 方法资源

方法	URL	VALUE
GET	SDC_URL_NNIE_FORWARD	1

5.4.3.2 请求 Content

```
/**
* 相对SVN_NNIE_FORWARD_CTRL_S有如下优点：
* 1、用户不再需要管理辅助内存段，接口使用更简单
* 2、用户无需指定NNIE_ID，由服务端自主调度，避免多个model执行时的资源冲突
* 3、多个model共享辅助内存段，最大程度降低对内存资源的述求
*/
struct sdc_nnie_forward_ctrl
{
uint32_t netseg_id;
uint32_t max_batch_num;
uint32_t max_bbox_num;
uint32_t reserve;
};
struct {
SVP_NNIE_MODEL_S model;
struct sdc_nnie_forward_ctrl foward_ctl;
SVP_SRC_BLOB_S astSrc[16];
SVP_DST_BLOB_S astDst[16];
};
```

5.4.3.3 请求扩展头

5.4.3.3.1 任务优先级（SDC_HEAD_PRI）

扩展头类型（0xFFFE）	扩展头长度（0x08）
优先级（取值范围[-127,127],数字越小代表优先级越高）	

优先级取值越小，优先级越高。低优先级的任务会有最大等待时延(由服务端控制，缺省300ms)，超过此等待时延，高优先级任务也不能抢占低优先级的任务。

5.4.3.4 响应码

成功返回200，其他错误码。

如果启动流控，响应码509（Bandwidth Limit Exceeded），无内容。

5.4.3.5 响应 Content

无。

收到响应后，astDst传递的物理内存中的内容被更新了。

5.4.3.6 响应扩展头

无。

5.4.3.7 参考样例

```
void SDC_Nnie_Forward(struct sdc_nnie_forward *p_sdc_nnie_forward)
{
int nRet;
struct sdc_common_head rsp_head;
struct sdc_common_head head;
struct iovec iov[2] = {
[0] = {.iov_base = &head, .iov_len = sizeof(head)},
[1] = {.iov_base = p_sdc_nnie_forward, .iov_len = sizeof(*p_sdc_nnie_forward)}
};
// fill head struct
memset(&head, 0, sizeof(head));
head.version = SDC_VERSION;
head.url = SDC_URL_NNIE_FORWARD;
head.method = SDC_METHOD_GET;
head.head_length = sizeof(head);
head.content_length = sizeof(*p_sdc_nnie_forward);
// write request
nRet = writev(fd_algorithm, iov, sizeof(iov)/sizeof(iov[0]));
if (nRet < 0)
{
fprintf(stdout,"Error:failed to write info to NNIE Forward!\n");
}
// read response
iov[0].iov_base = &rsp_head;
iov[0].iov_len = sizeof(rsp_head);
nRet = readv(fd_algorithm, iov, 1);
if (rsp_head.code != SDC_CODE_200 || nRet < 0)
{
fprintf(stdout,"Error:failed to read info from NNIE Forward!\n");
}
}
```

5.4.4 NNIE ForwardWithBbox

多节点输入输出的CNN类型网络预测。

注：等同于将海思的
HI_MPI_SVP_NNIE_GetTskBufSize,HI_MPI_SVP_NNIE_ForwardWithBbox和
HI_MPI_SVP_NNIE_Query接口合一了，后者就是响应。

5.4.4.1 请求 Common Head 方法资源

方法	URL	VALUE
GET	SDC_URL_NNIE_FORWARD_BBOX	2

5.4.4.2 请求 Content

```
/**
* 相对SVN_NNIE_FORWARD_WITHBBOX_CTRL_S有如下优点：
* 1、用户不再需要管理辅助内存段，接口使用更简单
* 2、用户无需指定NNIE_ID，由服务端自主调度，避免多个model执行时的资源冲突
* 3、多个model共享辅助内存段，最大程度降低对内存资源的述求
*/
struct sdc_nnie_forward_with_bbox_ctrl
{
uint32_t proposal_num;
uint32_t netseg_id;
uint32_t max_batch_num;
uint32_t max_bbox_num;
};
struct {
SVP_NNIE_MODEL_S model;
struct sdc_nnie_forward_with_bbox_ctrl forward_ctl;
SVP_SRC_BLOB_S astSrc[16];
SVP_SRC_BLOB_S astBbox[16];
SVP_DST_BLOB_S astDst[16];
};
```

5.4.4.3 请求扩展头

5.4.4.3.1 任务优先级（SDC_HEAD_PRI）

扩展头类型（0xFFFE）	扩展头长度（0x08）
优先级（取值范围[-127,127],数字越小代表优先级越高）	

优先级取值越小，优先级越高。低优先级的任务会有最大等待时延(由服务端控制，缺省300ms)，超过此等待时延，高优先级任务也不能抢占低优先级的任务。

5.4.4.4 响应码

成功返回200，其他错误码。

如果启动流控，响应码509（ Bandwidth Limit Exceeded ），无内容。

5.4.4.5 响应 Content

无。

收到响应后，astDst传递的物理内存中的内容被更新了。

5.4.4.6 响应扩展头

无。

5.4.4.7 参考样例

```
void SDC_Nnie_Forward_Withbox(struct sdc_nnie_forward_withbox
*p_sdc_nnie_forward_withbox)
{
int nRet;
struct sdc_common_head rsp_head;
struct sdc_common_head head;
struct iovec iov[2] = {
[0] = {.iov_base = &head , .iov_len = sizeof(head)},
[1] = {.iov_base = p_sdc_nnie_forward_withbox, .iov_len =
sizeof(*p_sdc_nnie_forward_withbox)}
};
// fill head struct
memset(&head, 0, sizeof(head));
head.version = SDC_VERSION;
head.url = SDC_URL_NNIE_FORWARD_BBOX;
head.method = SDC_METHOD_GET;
head.head_length = sizeof(head);
head.content_length = sizeof(*p_sdc_nnie_forward_withbox);
// write request
nRet = writev(fd_algorithm, iov, sizeof(iov)/sizeof(iov[0]));
if (nRet < 0)
{
fprintf(stdout,"Error:failed to write info to NNIE Forward_With_Box!\n");
}
// read response
iov[0].iov_base = &rsp_head;
iov[0].iov_len = sizeof(rsp_head);
nRet = readv(fd_algorithm, iov, 1);
if (rsp_head.code != SDC_CODE_200 || nRet < 0)
{
fprintf(stdout,"Error:failed to read info from NNIE Forward_With_Box!\n");
}
return;
}
```

5.4.5 VGS 创建缩放任务

使用VGS进行数据缩放、搬移，可以实现图片叠加等功能

5.4.5.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	SDC_URL_VGS_TASK	4

5.4.5.2 请求 Content

struct sdc_vgs_data

{

struct sdc_yuv_frame frame; //帧数据

struct sdc_region region; //帧数据的有效区域，region为空表示数据缩放、搬运

};

struct sdc_vgs _frame

{

struct sdc_vgs_data src_data; //源数据

struct sdc_vgs_data dst_data；//目的数据内存，需要用户申请内存

};

5.4.5.3 请求扩展头

无

5.4.5.4 响应码

成功返回200，其他错误码。

5.4.5.5 响应 Content

处理后的数据放到请求content指向的dst_data内存

5.4.5.6 响应扩展头

无。

5.4.5.7 参考样例

5.4.6 IVE 创建 DMA 任务

创建直接内存访问任务，支持快速拷贝、内存填充：可实现数据从一块内存快速拷贝到另一块内存，或者对一块内存进行填充操作。当前支持直接拷贝任务。

5.4.6.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	SDC_URL_IVE_DMA	6

5.4.6.2 请求 Content

struct sdc_ive_data

{

struct sdc_yuv_frame src_data;

struct sdc_yuv_frame dst_data;

}

src_data: 支持分辨率32x1～1920x1080

dst_data：直接拷贝分辨率同src_data

5.4.6.3 请求扩展头

无

5.4.6.4 响应码

成功返回200，其他错误码。

5.4.6.5 响应 Content

处理后的数据放到请求content指向的dst_data内存

5.4.6.6 响应扩展头

无

5.4.6.7 参考样例

5.4.7 IVE 创建 CSC 任务

创建色彩空间转换任务，可实现 YUV2RGB\RGB2YUV 的色彩空间转换。

5.4.7.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	SDC_URL_IVE_CSC	7

5.4.7.2 请求 Content

struct sdc_ive_data

注：YUV2RGB接口，源Format为SDC_YVU_420SP，目的Format为非SDC_YVU_420SP,目的size为RGB大小

RGB2YUV接口，源Format为非SDC_YVU_420SP，目的Format为SDC_YVU_420SP,目的size为YUV大小

5.4.7.3 请求扩展头

无

5.4.7.4 响应码

成功返回200，其他错误码。

5.4.7.5 响应 Content

处理后的数据放到请求content指向的dst_data内存

5.4.7.6 响应扩展头

无

5.4.7.7 参考样例

5.4.8 IVE 创建 RESIZE 任务

创建图像缩放任务，支持多张 U8C1\ U8C3_PLANAR 图像同时输入做一种类型的缩放。

5.4.8.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	SDC_URL_IVE_RESIZE	8

5.4.8.2 请求 Content

struct sdc_ive_data

输入图片数量=hbtp.content_length / sizeof(struct sdc_ive_data);

src_data :支持分辨率32x12～1920x1080

dst_data:支持分辨率32x12～1920x1080

5.4.8.3 请求扩展头

无

5.4.8.4 响应码

成功返回200，其他错误码。

5.4.8.5 响应 Content

处理后的数据放到请求content指向的dst_data内存

5.4.8.6 响应扩展头

无

5.4.8.7 参考样例

5.4.9 IVE 创建 DILATE 任务

创建二值图像 5x5 模板膨胀任务。

5.4.9.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	SDC_URL_IVE_DILATE	9

5.4.9.2 请求 Content

struct sdc_ive_data

src_data :支持分辨率 64x64～1920x1024

dst_data:分辨率同src_data

5.4.9.3 请求扩展头

无

5.4.9.4 响应码

成功返回200，其他错误码。

5.4.9.5 响应 Content

处理后的数据放到请求content指向的dst_data内存

5.4.9.6 响应扩展头

无

5.4.9.7 参考样例

5.4.10 IVE 创建 ERODE 任务

创建二值图像 5x5 模板腐蚀任务。

5.4.10.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	SDC_URL_IVE_ERODE	10

5.4.10.2 请求 Content

struct sdc_ive_data

src_data :支持分辨率 64x64～1920x1024

dst_data:分辨率同src_data

5.4.10.3 请求扩展头

无

5.4.10.4 响应码

成功返回200，其他错误码

5.4.10.5 响应 Content

处理后的数据放到请求content指向的dst_data内存

5.4.10.6 响应扩展头

无

5.4.10.7 参考样例

5.5 IVE 通用接口创建运行环境

设置IVE通用接口运行环境，加载IVE配置文件。

5.5.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	SDC_URL_IVE	20

5.5.2 请求 Content

char filepath[256]; //定义配置文件的路径

特别说明：注册链接必须是长连接，连接关闭则自动注销该IVE链接。

注册IVE运行配置文件遵循INI格式如下：

libpath=.so #库的路径和名称 如：/lib/sdc_ive.so

libdeppath=/lib #算法运行依赖库所在路径，如APP需要的OPENCV等。

libentry=xxxxx #函数入口,如SDC_IVE_ExecBlob

注：ini文件行最大长度256字节。

5.5.3 请求扩展头

无

5.5.4 响应码

如果功能正常，响应码200，入参错误返回400，其他执行错误为500。

5.5.5 响应 Content

无

5.5.6 响应扩展头

无

5.5.7 参考样例

5.6 IVE 通用接口执行 IVE 运算

执行IVE运行输出结果

5.6.1 请求 Common Head 方法资源

方法	URL	VALUE
UPDATE	SDC_URL_IVE	20

5.6.2 请求 Content

#define IVE_MAX_ BLOB_NUM (8)

struct SDC_IVE_SRC_BLOB

{

uint64_t phyAddr;

uint64_t virAddr;

uint32_t stride;

uint32_t width;

uint32_t height;

uint32_t type;

uint32_t reserve[2];

}

struct SDC_IVE_EXEC_INFO

{

Struct SDC_IVE_SRC_BLOB stSrcBlob[IVE_MAX_ BLOB_NUM];

Struct SDC_IVE_SRC_BLOB stDstBlob[IVE_MAX_ BLOB_NUM];

};

特别说明1：

函数原型：int32_t SDC_IVE_ExecBlob(struct SDC_IVE_EXEC_INFO *pstIveParam)

函数返回值：0-成功, 其他-失败

动态库函数需严格按照原型规范定义，除函数名称外，函数返回值类型、入参形式数量不可更改。

5.6.3 请求扩展头

无

5.6.4 响应码

如果动态库执行函数返回0，则响应码为200，入参错误返回400，执行错误返回500.

5.6.5 响应 Content

无

5.6.6 响应扩展头

无

注意事项：由于跨进程虚拟地址无法直接使用，动态库.so中如果需要使用虚拟地址，需要先使用MMAP操作。

5.6.7 参考样例

#include "sdc.h"

#include <inttypes.h>

#include <fcntl.h>

#include <sys/uio.h>

```
#include <sys/epoll.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
#include <time.h>
#define IVE_MAX_BLOB_NUM (8)
struct sdc_ive_blob_info
{
uint64_t phyAddr;
uint64_t virAddr;
uint32_t stride;
uint32_t width;
uint32_t height;
uint32_t type;
uint32_t reserve[2];
};
struct sdc_ive_exec_info
{
struct sdc_ive_blob_info stSrcBlob[IVE_MAX_BLOB_NUM];
struct sdc_ive_blob_info stDstBlob[IVE_MAX_BLOB_NUM];
};
#define SDC_URI_IVE (20)

static int load_ive_env(int fd)
{
int nret,i;
char buf[4096] = { 0 };
struct sdc_common_head* head = (struct sdc_common_head*) buf;
char file[256] = "/usr/app/bin/ive_env.ini";
char *data = NULL ;
```



```
head->version = SDC_VERSION;
head->url = SDC_URI_IVE;
head->method = SDC_METHOD_CREATE;
head->head_length = sizeof(*head);

data = (char*)&buf[head->head_length];
memcpy(data,file,sizeof(file));
head->content_length = sizeof(file);

nret = write(fd, head, head->head_length + head->content_length);
if(nret < 0)
{
printf("writev load_ive_env fail,response:%d,url:%d,code:%d,method:%d\n",head->response,head->url,head->code, head->method);
goto fail;
}
nret = read(fd, buf,sizeof(buf));
if(nret < 0)
{
printf("read load_ive_env fail,response:%d,url:%d,code:%d,method:%d\n",head->response,head->url,head->code, head->method);
goto fail;
}
if(head->code != SDC_CODE_200) {
printf("read load_ive_env fail,response:%d,url:%d,code:%d,method:%d\n",head->response,head->url,head->code, head->method);
goto fail;
}
return 0;
fail:
return 1;
}

static int ive_exec_blob(int fd)
{
int nret,i;
```



```
char buf[4096] = { 0 };
struct sdc_common_head* head = (struct sdc_common_head*) buf;
char *data = NULL ;
struct sdc_ive_exec_info ive_exec_info = {0} ;
head->version = SDC_VERSION;
head->url = SDC_URI_IVE;
head->method = SDC_METHOD_UPDATE;
head->head_length = sizeof(*head);
data = (char*)&buf[head->head_length];
memcpy(data,&ive_exec_info,sizeof(ive_exec_info));
head->content_length = sizeof(ive_exec_info);
nret = write(fd, head, head->head_length + head->content_length);
if(nret < 0)
{
printf("writev ive_exec_blob fail,response:%d,url:%d,code:%d,method:%d\n",head->response,head->url,head->code, head->method);
goto fail;
}
nret = read(fd, buf,sizeof(buf));
if(nret < 0)
{
printf("read ive_exec_blob fail,response:%d,url:%d,code:%d,method:%d\n",head->response,head->url,head->code, head->method);
goto fail;
}
if(head->code != SDC_CODE_200) {
printf("read ive_exec_blob fail,response:%d,url:%d,code:%d,method:%d\n",head->response,head->url,head->code, head->method);
goto fail;
}
return 0;
fail:
return 1;
}
```



```
int main(int argc, char *argv[])
{
int fd;
int32_t ret = -1;

fd = open("/mnt/srvfs/algorithm.iaas.sdc", O_RDWR);
if (fd < 0)
{
printf("open alg.iaas.sdc srv fail\n");
return -1;
}
ret = load_ive_env(fd);
if (ret != 0) {
printf("load ive env fail\n");
}
ret = ive_exec_blob(fd);
if (ret != 0) {
printf("ive exec ive fail\n");
}
close(fd);
return 0;
}
```

5.7 ptz.iaas.sdc 服务化接口定义

5.7.1 PTZ 获取预置位

获取已设置的预置点坐标信息、预置点编号。预置点号[0, 255]

5.7.1.1 请求 Common Head 方法资源

方法	URL	VALUE
GET	PTZ_URL_PRESET_POSITION	0

5.7.1.2 请求 Content

uint32_t preset_id，指明获取第几号预置点的位置信息，如果客户端未设置预置点编号（无content内容）则返回全部的预置点信息。

5.7.1.3 请求扩展头

无

5.7.1.4 响应码

如果功能正常，响应码200，其他为错误码。

5.7.1.5 响应 Content

struct preset_pos

{

double h_location; //水平角度，[0,360]。精度0.1。注：0度和360度在坐标系中为同一位置。

double v_location; //垂直角度，[0,110]。精度0.1

int64_t zoom_pos; //镜头变倍坐标 运行到指定位置时下发SDC上获取到的正确坐标信息

int64_t focus_pos; //镜头聚焦坐标 运行到指定位置时下发SDC上获取到的正确坐标信息

}

struct preset_attr

{

char desc[32]; //预置点名，此为预留字段，目前不支持

struct preset_pos preset_position;

}

struct preset_info

{

uint32_t preset_id; //预置点号

uint32_t reserve;

struct preset_attr preset_attr;

}

多个预置点返回数组，数组长度=hbtp.content_length / sizeof(struct preset_info)

5.7.1.6 响应扩展头

无。

5.7.1.7 参考样例

```
#include "sdc.h"
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include "stdio.h"

void display_preset_data(struct preset_info* info)
{
printf("ID %d \n",info->preset_id);
printf("Hl %lf \n",info->preset_attr.preset_position.hLocation);
printf("Vl %lf \n",info->preset_attr.preset_position.vLocation);
printf("fp %lld \n",info->preset_attr.preset_position.FocusPos);
printf("zp %lld \n",info->preset_attr.preset_position.ZoomPos);

}
int main(int argc,char* argv[])
{
int nret,i;
char buf[1024] = { 0 };
struct preset_info* info;
struct sdc_common_head* head = (struct sdc_common_head*)buf;

/** query all channels' info */
head->version = SDC_VERSION; //0x5331
head->url = PTZ_URL_PRESET_POSITION; //0x00
head->method = SDC_METHOD_GET; //0x02
head->head_length = sizeof(*head);

int fd = open("/mnt/srvfs/ptz.iaas.sdc", O_RDWR);
if(fd < 0) goto fail;

nret = write(fd, head, head->head_length + head->content_length);
if(nret < 0) goto fail;

nret = read(fd,buf,sizeof(buf));
if(nret < 0 || head->code != SDC_CODE_200) goto fail;
```

```
for(i = 0, info = (struct preset_info*)&buf[head->head_length]; i < head->content_length / sizeof(*info); ++i, ++info) {
display_preset_data(info);
}
close(fd);
return 0;
fail:
exit(1); //fd will be closed after exit
}
```

5.7.2 PTZ 创建预置位

设置预置点信息

5.7.2.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	PTZ_URL_PRESET_POSITION	0

5.7.2.2 请求 Content

struct preset_attr;

支持批量创建，多个预置点设置，数组长度=hbtp.content_length / sizeof(struct preset_attr)

5.7.2.3 请求扩展头

无

5.7.2.4 响应码

如果功能正常，响应码200，其他为错误码。

5.7.2.5 响应 Content

多个预置点编号返回数组，数组长度=hbtp.content_length / sizeof(uint32_t);

5.7.2.6 响应扩展头

无。

5.7.2.7 参考样例

```
#include "sdc.h"
#include <unistd.h>
```

```
#include <fcntl.h>
#include <stdlib.h>
#include "stdio.h"
#include <stddef.h>
#include <errno.h>
#include <sys/uio.h>

int main(int argc,char* argv[])
{
int nret,i;
char buf[1024] = { 0 };
struct preset_attr param = {
.preset_position.FocusPos = -2458 ,
.preset_position.hLocation = 203.89,
.preset_position.vLocation= 19.990000 ,
.preset_position.ZoomPos = 1000 ,
};
struct sdc_common_head head = {
.version = SDC_VERSION, //0x5331
.url = PTZ_URL_PRESET_POSITION, //0x00
.method = SDC_METHOD_CREATE, //0x02
.content_length = sizeof(param),
.head_length = sizeof(head),
};

struct iovec iov[2] = {
{.iov_base = &head, .iov_len = sizeof(head)},
{.iov_base = ¶m, .iov_len = sizeof(param) }};

int fd = open("/mnt/srvfs/ptz.iaas.sdc", O_RDWR);
if(fd < 0) goto fail;

nret = writev(fd, iov, 2);
if(nret < 0) goto fail;

nret = read(fd,&head, sizeof(head));
```



```
if(head.code != SDC_CODE_200)
{
printf("recv error");
}
close(fd);
return 0;
fail:
exit(1); //fd will be closed after exit
}
```

5.7.3 PTZ 删除预置位

删除已设置的预置点坐标信息、预置点编号。

5.7.3.1 请求 Common Head 方法资源

方法	URL	VALUE
DELETE	PTZ_URL_PRESET_POSITION	0

5.7.3.2 请求 Content

uint32_t id，支持批量删除，数组长度=hbtp.content_length / sizeof(uint32_t)。

5.7.3.3 请求扩展头

无

5.7.3.4 响应码

如果功能正常，响应码200，其他为错误码。

5.7.3.5 响应 Content

uint32_t id

如果有预置点删除失败，则返回删除失败预置点数组，数组长度=hbtp.content_length / sizeof(uint32_t)，响应码返回400，响应码为200则表示全部删除成功。

5.7.3.6 响应扩展头

无。

5.7.3.7 参考样例

```
#include "sdc.h"
```

```
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include "stdio.h"
#include <stddef.h>
#include <errno.h>
#include <sys/uio.h>

int main(int argc,char* argv[])
{
int nret,i;
char buf[1024] = { 0 };

uint32_t preset_id = 2;
struct sdc_common_head head = {
.version = SDC_VERSION, //0x5331
.url = PTZ_URL_PRESET_POSITION, //0x00
.method = SDC_METHOD_DELETE, //0x02
.content_length = sizeof(uint32_t),
.head_length = sizeof(head),
};

struct iovec iov[2] = { {.iov_base = &head, .iov_len = sizeof(head)},
{.iov_base = &preset_id, .iov_len = sizeof(preset_id) }};

int fd = open("/mnt/srvfs/ptz.iaas.sdc", O_RDWR);
if(fd < 0) goto fail;

nret = writev(fd, iov, 2);
if(nret < 0) goto fail;

nret = read(fd,&head, sizeof(head));
if(head.code != SDC_CODE_200)
{
printf("recv error");
}
close(fd);
```


return 0;

fail:

exit(1); //fd will be closed after exit

}

5.7.4 PTZ 运动到指定位置

控制云台运动到指定位置或进行3D定位，或通过下发预置点好控制云台运动到指定预置点。由于设备的运动精度为0.01，执行运动操作后，获取位置信息与指定的目的位置信息允许有偏差，精度0.01。

5.7.4.1 请求 Common Head 方法资源

方法	URL	VALUE
UPDATE	PTZ_URL_LOCATION	1

5.7.4.2 请求 Content

struct preset_pos //默认解析类型

或

struct ptz_3d

{

double zoom_ratio; //放大、缩小倍数，大于1放大，小于1缩小：目的宽/原宽（目的高/原高），精度0.1

uint32_t x;//中心点坐标

uint32_t y;

uint32_t original_image_width; //原图宽（此中心点坐标是相对那幅图）

uint32_t original_image_height; //原图高

}

或

struct ptz_3d_common

{

struct sdc_region object_region;

uint32_t dst_width; //放大/缩小的目的宽dst_width/object_region.width：大于1放大，小于1缩小

uint32_t dst_height;

uint32_t original_image_width; //原图宽（目标识别使用的图片）

uint32_t original_image_height; //原图高

}

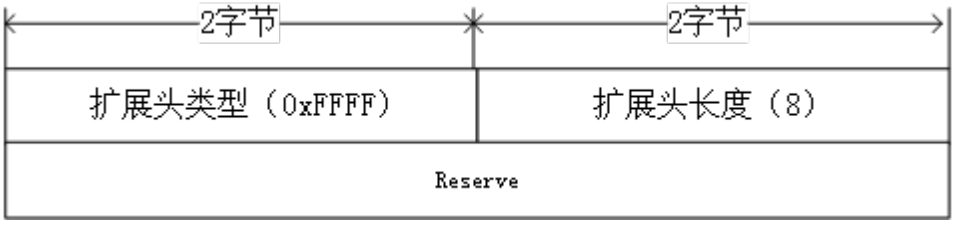
或

uint32_t preset_id;

注：通过struct ptz_3d、struct ptz_3d_common接口下发倍率与实际倍率可能存在大概10%左右的误差，需要用户根据实际设备能力可进行适当的业务调整

5.7.4.3 请求扩展头

5.7.4.3.1 指示运动控制方式扩展头（SDC_HEAD _PTZ_CONTENT_TYPE）



扩展头Reserve字段的取值为0表示请求content内容为struct preset_pos类型，为1表示为struct ptz_3d类型，为2表示struct ptz_3d_common表示放大/缩小倍率由服务计算，为3表示通过下发预置点ID控制云台运动到预置点位置。

#define CONTENT_TYPE_PRESET_POS 0

#define CONTENT_TYPE_PTZ_3D 1

#define CONTENT_TYPE_PTZ_3D_COMMON 2

#define CONTENT_TYPE_PRESET_ID 3

5.7.4.4 响应码

如果功能正常，响应码200， 其他为错误码。

摄像机运动到指定位置后返回消息。

5.7.4.5 响应 Content

无

5.7.4.6 响应扩展头

成功返回200，其他异常返回其他错误码。

5.7.4.7 参考样例

#include "sdc.h"

#include <unistd.h>

#include <fcntl.h>

#include <stdlib.h>

#include "stdio.h"

```
#include <stddef.h>
#include <errno.h>
#include <sys/uio.h>
int main(int argc,char* argv[])
{
int nret,i;
char buf[1024] = { 0 };
uint32_t preset_id = 0;

struct sdc_extend_head content_head = {
SDC_HEAD_PTZ_CONTENT_TYPE, sizeof(content_head),
CONTENT_TYPE_PRESET_POS
};
struct preset_pos param = {
.FocusPos = -2458 ,
.hLocation = 203.89,
.vLocation= 19.990000 ,
.ZoomPos = 0 ,
};
struct sdc_common_head head = {
.version = SDC_VERSION, //0x5331
.url = PTZ_URL_LOCATION, //0x00
.method = SDC_METHOD_UPDATE, //0x02
.content_length = sizeof(param),
.head_length = sizeof(head)+sizeof(content_head),
};

struct iovec iov[3] = { {.iov_base = &head, .iov_len = sizeof(head)},
{.iov_base = &content_head, .iov_len = sizeof(content_head)},
{.iov_base = &param, .iov_len = sizeof(param) }};

int fd = open("/mnt/srvfs/ptz.iaas.sdc", O_RDWR);
if(fd < 0) goto fail;

nret = writev(fd, iov, 3);
if(nret < 0) goto fail;
```



```
nret = read(fd,&head, sizeof(head));
if(head.code != SDC_CODE_200)
{
printf("recv error");
}
close(fd);
return 0;
fail:
exit(1); //fd will be closed after exit
}
```

5.7.5 PTZ 获取当前位置

获取球机PTZ的当前坐标信息及运动、静止状态。

5.7.5.1 请求 Common Head 方法资源

方法	URL	VALUE
GET	PTZ_URL_LOCATION	1

5.7.5.2 请求 Content

无

5.7.5.3 请求扩展头

无

5.7.5.4 响应码

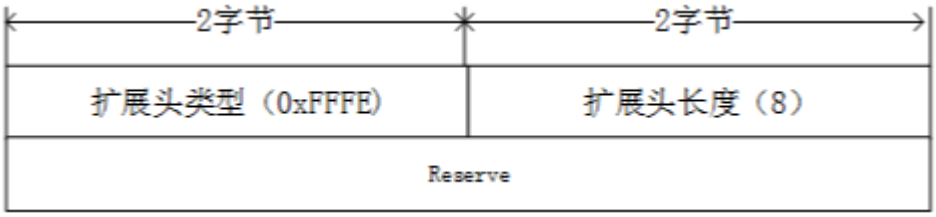
如果功能正常，响应码200， 其他为错误码。

5.7.5.5 响应 Content

struct preset_pos;

5.7.5.6 响应扩展头

5.7.5.6.1 指示 PTZ 运动状态扩展头（SDC_HEAD _PTZ_CURRENT_LOCATION）



扩展头Reserve字段的取值为0代表摄像机处于静止状态，1代表运动中状态

5.7.5.7 参考样例

```
#include "sdc.h"
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include "stdio.h"

int main(int argc,char* argv[])
{
int nret,i;
char buf[1024] = { 0 };
struct preset_pos* info;
struct sdc_common_head* head = (struct sdc_common_head*)buf;

/** query all channels' info */
head->version = SDC_VERSION; //0x5331
head->url = PTZ_URL_LOCATION; //0x00
head->method = SDC_METHOD_GET; //0x02
head->head_length = sizeof(*head);

int fd = open("/mnt/srvfs/ptz.iaas.sdc", O_RDWR);
if(fd < 0) goto fail;

nret = write(fd, head, head->head_length + head->content_length);
if(nret < 0) goto fail;

nret = read(fd,buf,sizeof(buf));
if(nret < 0 || head->code != SDC_CODE_200) goto fail;

info = (struct preset_pos*)&buf[head->head_length];
printf("Hl %lf \n",info->hLocation);
printf("Vl %lf \n",info->vLocation);
printf("fp %lld \n",info->FocusPos);
printf("zp %lld \n",info->ZoomPos);

close(fd);
```



```
return 0;
fail:
exit(1); //fd will be closed after exit
}
```

5.7.6 PTZ 指定方向运动

5.7.6.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	PTZ_URL_CTRLDIRECTION	2

5.7.6.2 请求 Content

```
struct ptz_ctrldirection
{
uint32_t direction ;//方向，从1开始为上、下、左、右、右上、左上、右下、左下
float speed;//运动速度
};
```

5.7.6.3 请求扩展头

无

5.7.6.4 响应码

如果功能正常，响应码200， 其他为错误码。

5.7.6.5 响应 Content

无

5.7.6.6 响应扩展头

无

5.7.6.7 参考用例

无

5.7.7 PTZ 停止运动

方法	URL	VALUE
DELETE	PTZ_URL_CTRLDIRECTION	2

5.7.7.1 请求 Common Head 方法资源

无

5.7.7.2 请求 Content

无

5.7.7.3 请求扩展头

无

5.7.7.4 响应码

如果功能正常，响应码200， 其他为错误码。

5.7.7.5 响应 Content

无

5.7.7.6 响应扩展头

无

5.7.7.7 参考用例

5.7.8 PTZ 能力参数

5.7.8.1 请求 Common Head 方法资源

方法	URL	VALUE
GET	PTZ_URL_ABILITY	3

5.7.8.2 请求 Content

无

5.7.8.3 请求扩展头

无

5.7.8.4 响应码

如果功能正常，响应码200， 其他为错误码。

5.7.8.5 响应 Content

```
struct ptz_ability
{
double horAngel;//水平最大角度
double verAngel;//垂直最大角度
float horSpeed;//水平最大速度
float verSpeed;//垂直最大速度
};
```

5.7.8.6 响应扩展头

无

5.7.8.7 参考用例

5.8 crypto.iaas.sdc 服务化接口定义

5.8.1 AES/DES 加密

5.8.1.1 创建 AES/DES 加密

设置加密参数，具体参数参照请求content

5.8.1.1.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	CRYPTO_URI_AES_DES_ENCRYPT	0

5.8.1.1.2 请求 Content

```
struct crpto_encdec_config{
uint32_t algorithm; //加解密算法
uint32_t workmode; //算法工作模式
unsigned char keyParam[32]; //密钥
uint32_t keyLength; //密钥长度
```


unsigned char ivParam[16]; //初始向量值

uint32_t ivLength; //iv长度

uint32_t bitWidth; //数据加解密位宽

uint32_t reserve[1]; //保留，对齐

struct crypto_ccmgcm_config config[0]; //CCM/GCM参数

};

参数名称	描述	取值
algorithm	加解密算法类型	2—AES加解密 注：DES和3DES加解密安全性低，不支持使用
workmode	算法工作模式	0—ECB模式 1—CBC模式 2—CFB模式 3—OFB模式 4—CTR模式 7—CBC_CTS模式 注：当前接口不支持CCM/GCM模式。DES、3DES不支持CTR模式。
keyParam[32]	密钥	-
keyLength	密钥长度	0—默认长度。AES模式下为128bit，DES模式下为64bit。 1—AES192bit 2—AES256bit 3—DES3Key 4—DES2key
ivParam[16]	初始向量值	-
ivLength	iv长度	除ECB模式外其余模式需要初始向量值。 其余模式取16位

参数名称	描述	取值
bitWidth	数据加解密位宽	0—64bit 1—8bit 2—1bit 3—128bit 注：DES算法CFB 和 OFB 模式支持1/8/64位宽 AES算法CFB模式支持1/8/128bit，OFB模式仅支持128bit。
reserve[1]	保留字段，结构体对齐	-
config[0]	CCM/GCM模式参数，其余模式无此参数	当前接口不支持CCM/GCM模式，无需设置

5.8.1.1.3 请求扩展头

无

5.8.1.1.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.1.1.5 响应 Content

无

5.8.1.1.6 响应扩展头

无。

5.8.1.1.7 参考样例

5.8.1.2 加密 AES/DES 数据

加密数据。具体参数参照请求content

5.8.1.2.1 请求 Common Head 方法资源

方法	URL	VALUE
UPDATE	CRYPTO_URI_AES_DES_ENCRYPT	0

5.8.1.2.2 请求 Content

struct crypto_config_data{

uint32_t dataStatus; //数据状态

uint32_t dataNum; //加解密数据个数

struct crypto_data_param data[0]; //加解密数据结构体

struct crypto_data_param tag[0]; //tag值

};

参数名称	描述	取值
dataStatus	数据状态	0—update。 1—final。 注：加解密采用流式加解密方式，内部16字节对齐，当状态为final时会把对齐剩余数据结果返回
dataNum	加解密数据个数	-
data[0]	加解密数据结构体	见下表
tag[0]	TAG值。CCM/GCM模式专用参数	当前接口无TAG值。

struct crypto_data_param{

uint32_t length; //数据长度

unsigned char data[0]; //数据

};

参数名称	描述	取值
length	数据长度	-
data[0]	数据	-

5.8.1.2.3 请求扩展头

无

5.8.1.2.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.1.2.5 响应 Content

struct crypto_config_data；具体参数同请求content

5.8.1.2.6 响应扩展头

无。

5.8.1.2.7 参考样例

5.8.1.3 销毁 AES/DES 加密

销毁加密信息

5.8.1.3.1 请求 Common Head 方法资源

方法	URL	VALUE
DELETE	CRYPTO_URI_AES_DES_ENCRYPT	0

5.8.1.3.2 请求 Content

无

5.8.1.3.3 请求扩展头

无

5.8.1.3.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.1.3.5 响应 Content

无

5.8.1.3.6 响应扩展头

无。

5.8.1.3.7 参考样例

5.8.2 AES/DES 解密

5.8.2.1 创建 AES/DES 解密

设置解密参数，具体参数参照请求content

5.8.2.1.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	CRYPTO_URI_AES_DES_DECRYPT	1

5.8.2.1.2 请求 Content

```
struct crypto_config_data {
uint32_t algorithm; //加解密算法
uint32_t workmode; //算法工作模式
unsigned char keyParam[32]; //密钥
uint32_t keyLength; //密钥长度
unsigned char ivParam[16]; //初始向量值
uint32_t ivLength; //iv长度
uint32_t bitWidth; //数据加解密位宽
uint32_t reserve[1]; //保留，对齐
struct crypto_data_param config[0]; //CCM/GCM参数
};
```

参数名称	描述	取值
algorithm	加解密算法类型	2—AES加解密 注：DES和3DES加解密安全性低，不支持使用
workmode	算法工作模式	0—ECB模式 1—CBC模式 2—CFB模式 3—OFB模式 4—CTR模式 7—CBC_CTS模式 注：当前接口不支持CCM/GCM模式。DES、3DES不支持CTR模式。
keyParam[32]	密钥	-
keyLength	密钥长度	0—默认长度。AES模式下为128bit，DES模式下为64bit 1—AES192bit 2—AES256bit 3—DES3Key 4—DES2key
ivParam[16]	初始向量值	-

参数名称	描述	取值
ivLength	iv长度	除ECB模式外其余模式需要初始向量值。 其余模式取16位
bitWidth	数据加解密位宽	0—64bit 1—8bit 2—1bit 3—128bit 注：DES算法CFB 和 OFB 模式支持1/8/64位宽 AES算法CFB模式支持1/8/128bit，OFB模式仅支持128bit。
reserve[1]	保留字段，结构体对齐	-
config[0]	CCM/GCM模式参数，其余模式无此参数	当前接口不支持CCM/GCM模式，无需设置

5.8.2.1.3 请求扩展头

无

5.8.2.1.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.2.1.5 响应 Content

无

5.8.2.1.6 响应扩展头

无。

5.8.2.1.7 参考样例

5.8.2.2 AES/DES 解密数据

解密数据。具体参数参照请求content

5.8.2.2.1 请求 Common Head 方法资源

方法	URL	VALUE
UPDATE	CRYPTO_URI_AES_DES_DECRYPT	1

5.8.2.2.2 请求 Content

struct crypto_config_data{

uint32_t dataStatus; //数据状态

uint32_t dataNum; //加解密数据个数

struct crypto_data_param data[0]; //加解密数据结构体

struct crypto_data_param tag[0]; //tag值

};

参数名称	描述	取值
dataStatus	数据状态	0—update。 1—final。 注：加解密采用流式加解密方式，内部16字节对齐，当状态为final时会把对齐剩余数据结果返回
dataNum	加解密数据个数	-
data[0]	加解密数据结构体	见下表
tag[0]	TAG值。CCM/GCM模式专用参数	当前接口无TAG值。

struct crypto_data_param{

uint32_t length; //数据长度

unsigned char data[0]; //数据

};

参数名称	描述	取值
length	数据长度	-
data[0]	数据	-

5.8.2.2.3 请求扩展头

无

5.8.2.2.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.2.2.5 响应 Content

struct crypto_config_data；具体参数同请求content

5.8.2.2.6 响应扩展头

无。

5.8.2.2.7 参考样例

5.8.2.3 销毁 AES/DES 解密

销毁解密信息

5.8.2.3.1 请求 Common Head 方法资源

方法	URL	VALUE
DELETE	CRYPTO_URI_AES_DES_DECRYPT	1

5.8.2.3.2 请求 Content

无

5.8.2.3.3 请求扩展头

无

5.8.2.3.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.2.3.5 响应 Content

无

5.8.2.3.6 响应扩展头

无。

5.8.2.3.7 参考样例

5.8.3 CCM/GCM 加密

Hi3516DV300平台不支持CCM/GCM加密

5.8.3.1 创建 CCM/GCM 加密

Hi3516DV300平台不支持设置CCM/GCM加密参数，返回错误码404

5.8.3.1.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	CRYPTO_URI_CCM_GCM_ENCRYPT	2

5.8.3.1.2 请求 Content

struct crpto_encdec_config{

uint32_t algorithm; //加解密算法

uint32_t workmode; //算法工作模式

unsigned char keyParam[32]; //密钥

uint32_t keyLength; //密钥长度

unsigned char ivParam[16]; //初始向量值

uint32_t ivLength; //iv长度

uint32_t bitWidth; //数据加解密位宽

uint32_t reserve[1]; //保留，对齐

struct crypto_ccmgcm_config config[0]; //CCM/GCM参数

};

参数名称	描述	取值
algorithm	加解密算法类型	2—AES加解密 注：CCM/GCM模式算法类型只能选择AES算法
workmode	算法工作模式	5—CCM模式 6—GCM模式
keyParam[32]	密钥	-
keyLength	密钥长度	0—默认长度。AES模式下为128bit 1—AES192bit 2—AES256bit
ivParam[16]	初始向量值	-
ivLength	iv长度	CCM模式：[7, 13] GCM模式：[1, 16]

参数名称	描述	取值
bitWidth	数据加解密位宽	0—64bit 1—8bit 2—1bit 3—128bit
reserve[1]	保留字段，结构体对齐	-
config[0]	CCM/GCM模式参数	见下表

struct crypto_ccmgcm_config{

uint32_t tagLength; //tag值长度。

uint32_t dataLength; //关联数据长度

unsigned char relatedData[0]; //关联数据

};

参数名称	描述	取值
tagLength	Tag值长度，CCM/GCM模式专用	CCM：可取{4,6,8,10,12,14,16} GCM：可取{12,13,14,15,16}
dataLength	关联数据A	-
relatedData[0]	关联数据A的长度	-

5.8.3.1.3 请求扩展头

无

5.8.3.1.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.3.1.5 响应 Content

无

5.8.3.1.6 响应扩展头

无。

5.8.3.1.7 参考样例

5.8.3.2 CCM/GCM 加密数据

Hi3516DV300平台不支持CCM/GCM加密，返回错误码404

5.8.3.2.1 请求 Common Head 方法资源

方法	URL	VALUE
UPDATE	CRYPTO_URI_CCM_GCM_ENCRYPT	2

5.8.3.2.2 请求 Content

struct crypto_config_data{

uint32_t dataStatus; //数据状态

uint32_t dataNum; //加解密数据个数

struct crypto_data_param data[0]; //加解密数据结构体

struct crypto_data_param tag[0]; //tag值

};

参数名称	描述	取值
dataStatus	数据状态	0—update。 1—final。 注：加解密采用流式加解密方式，内部16字节对齐，当状态为final时会把对齐剩余数据结果返回
dataNum	加解密数据个数	-
data[0]	加解密数据结构体	见下表。
tag[0]	TAG值。CCM/GCM模式专用参数	见下表。发送数据时无需添加此项，接收数据时会返回该项

struct crypto_data_param{

uint32_t length; //数据长度

unsigned char data[0]; //数据

};

参数名称	描述	取值
length	数据长度	-

参数名称	描述	取值
data[0]	数据	-

5.8.3.2.3 请求扩展头

无

5.8.3.2.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.3.2.5 响应 Content

struct crypto_config_data；具体参数同请求content

5.8.3.2.6 响应扩展头

无。

5.8.3.2.7 参考样例

5.8.3.3 销毁 CCM/GCM 加密

Hi3516DV300平台不支持销毁CCM/GCM加密参数，返回错误码404

5.8.3.3.1 请求 Common Head 方法资源

方法	URL	VALUE
DELETE	CRYPTO_URI_CCM_GCM_ENCRYPT	2

5.8.3.3.2 请求 Content

无

5.8.3.3.3 请求扩展头

无

5.8.3.3.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.3.3.5 响应 Content

无

5.8.3.3.6 响应扩展头

无。

5.8.3.3.7 参考样例

5.8.4 CCM/GCM 解密

Hi3516DV300平台不支持CCM/GCM解密

5.8.4.1 创建 CCM/GCM 解密

Hi3516DV300平台不支持设置CCM/GCM解密参数，返回错误码404

5.8.4.1.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	CRYPTO_URI_CCM_GCM_DECRYPT	3

5.8.4.1.2 请求 Content

struct crpto_encdec_config{

uint32_t algorithm; //加解密算法

uint32_t workmode; //算法工作模式

unsigned char keyParam[32]; //密钥

uint32_t keyLength; //密钥长度

unsigned char ivParam[16]; //初始向量值

uint32_t ivLength; //iv长度

uint32_t bitWidth; //数据加解密位宽

uint32_t reserve[1]; //保留，对齐

struct crypto_ccmgcm_config config[0]; //CCM/GCM参数

};

参数名称	描述	取值
algorithm	加解密算法类型	2—AES加解密 注：CCM/GCM模式算法类型只能选择AES算法
workmode	算法工作模式	5—CCM模式 6—GCM模式
keyParam[32]	密钥	-

参数名称	描述	取值
keyLength	密钥长度	0—默认长度。AES模式下为128bit 1—AES192bit 2—AES256bit
ivParam[16]	初始向量值	-
ivLength	iv长度	CCM模式：[7, 13] GCM模式：[1, 16]
bitWidth	数据加解密位宽	0—64bit 1—8bit 2—1bit 3—128bit
reserve[1]	保留字段，结构体对齐	-
config[0]	CCM/GCM模式参数	见下表

struct crypto_ccmgcm_config{

uint32_t tagLength; //tag值长度。

uint32_t dataLength; //关联数据长度

unsigned char relatedData[0]; //关联数据

};

参数名称	描述	取值
tagLength	Tag值长度，CCM/GCM模式专用	CCM：可取{4,6,8,10,12,14,16} GCM：可取{12,13,14,15,16}
dataLength	关联数据A	-
relatedData[0]	关联数据A的长度	-

5.8.4.1.3 请求扩展头

无

5.8.4.1.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.4.1.5 响应 Content

无

5.8.4.1.6 响应扩展头

无。

5.8.4.1.7 参考样例

5.8.4.2 CCM/GCM 解密数据

Hi3516DV300平台不支持CCM/GCM解密，返回错误码404

5.8.4.2.1 请求 Common Head 方法资源

方法	URL	VALUE
UPDATE	CRYPTO_URI_CCM_GCM_DECRYPT	3

5.8.4.2.2 请求 Content

struct crypto_config_data{

uint32_t dataStatus; //数据状态

uint32_t dataNum; //加解密数据个数

struct crypto_data_param data[0]; //加解密数据结构体

struct crypto_data_param tag[0]; //tag值

};

参数名称	描述	取值
dataStatus	数据状态	0—update。 1—final。 注：加解密采用流式加解密方式，内部16字节对齐，当状态为final时会把对齐剩余数据结果返回
dataNum	加解密数据个数	-
data[0]	加解密数据结构体	见下表。
tag[0]	TAG值。CCM/GCM模式专用参数	见下表。发送数据时无需添加此项，接收数据时会返回该项

struct crypto_data_param{

uint32_t length; //数据长度

unsigned char data[0]; //数据

};

参数名称	描述	取值
length	数据长度	-
data[0]	数据	-

5.8.4.2.3 请求扩展头

无

5.8.4.2.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.4.2.5 响应 Content

struct crypto_config_data；具体参数同请求content

5.8.4.2.6 响应扩展头

无。

5.8.4.2.7 参考样例

5.8.4.3 销毁 CCM/GCM 解密

Hi3516DV300平台不支持销毁CCM/GCM解密参数，返回错误码404

5.8.4.3.1 请求 Common Head 方法资源

方法	URL	VALUE
DELETE	CRYPTO_URI_CCM_GCM_DECRYPT	3

5.8.4.3.2 请求 Content

无

5.8.4.3.3 请求扩展头

无

5.8.4.3.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.4.3.5 响应 Content

无

5.8.4.3.6 响应扩展头

无。

5.8.4.3.7 参考样例

5.8.5 RSA 公钥加密私钥解密

5.8.5.1 RSA 公钥加密

5.8.5.1.1 创建 RSA 公钥加密

?.1. 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	CRYPTO_URI_RSA_PUBLIC_ENCRYPT	6

?.2. 请求 Content

```
struct crypto_hash_rsa_config{
uint32_t algorithm; //算法工作模式
uint32_t keyNum; //密钥个数
struct crypto_key_param keyParam[0]; //密钥结构体
};
```

参数名称	描述	取值
algorithm	数据加解密填充模式	0—no_padding 1—BLOCK_TYPE_0 2—BLOCK_TYPE_1 3—BLOCK_TYPE_2 4—RSAES_OAEP_SHA1 5—RSAES_OAEP_SHA224 6—RSAES_OAEP_SHA256 7—RSAES_OAEP_SHA384 8—RSAES_OAEP_SHA512 9—RSAES_PKCS1_V1_5 注：当数据与公钥长度相等无需填充时，选择no_padding模式。其余填充模式数据长度应满足算法要求。
keyNum	密钥个数	公钥加解密密钥个数为2，按顺序为N、E密钥。 私钥加解密包括两种形式，一种个数为2，按顺序为N、D密钥，另一种个数为6，按顺序为N、P、Q、DP、DQ、QP密钥
keyParam[0]	密钥结构体	密钥长度可以为128、256、374、512位，其中3516DV300平台不支持374位

struct crypto_key_param {

uint32_t keyLength; //密钥长度

unsigned char key[0]; //密钥

};

参数名称	描述	取值
keyLength	密钥长度	-
key[0]	密钥	-

?.3. 请求扩展头

无

?.4. 响应码

如果功能正常，响应码200，其他为错误码。

?.5. 响应 Content

无

?.6. 响应扩展头

无。

?.7. 参考样例

5.8.5.1.2 RSA 公钥加密计算

?.1. 请求 Common Head 方法资源

方法	URL	VALUE
UPDATE	CRYPTO_URI_RSA_PUBLIC_ENCRYPT	6

?.2. 请求 Content

struct crypto_config_data{

uint32_t dataStatus; //数据状态

uint32_t dataNum; //加解密数据个数

struct crypto_data_param data[0]; //加解密数据结构体

struct crypto_data_param tag[0]; //tag值

};

参数名称	描述	取值
dataStatus	数据状态	0—update。 1—final。 注：加解密采用流式加解密方式，内部16字节对齐，当状态为final时会把对齐剩余数据结果返回
dataNum	加解密数据个数	-
data[0]	加解密数据结构体	见下表
tag[0]	TAG值。CCM/GCM模式专用参数	当前接口无TAG值。

struct crypto_data_param{

uint32_t length; //数据长度

unsigned char data[0]; //数据

};

参数名称	描述	取值
length	数据长度	-
data[0]	数据	-

?.3. 请求扩展头

无

?.4. 响应码

如果功能正常，响应码200，其他为错误码。

?.5. 响应 Content

struct crypto_config_data；具体参数参照请求content

?.6. 响应扩展头

无。

?.7. 参考样例

5.8.5.1.3 销毁 RSA 公钥加密

?.1. 请求 Common Head 方法资源

方法	URL	VALUE
DELETE	CRYPTO_URI_RSA_PUBLIC_ENCRYPT	6

?.2. 请求 Content

无

?.3. 请求扩展头

无

?.4. 响应码

如果功能正常，响应码200，其他为错误码。

?.5. 响应 Content

无

?.6. 响应扩展头

无。

?.7. 参考样例

5.8.5.2 RSA 私钥解密

5.8.5.2.1 创建 RSA 私钥解密

?.1. 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	CRYPTO_URI_RSA_PRIVATE_DECRYPT	7

?.2. 请求 Content

```
struct crypto_hash_rsa_config{
uint32_t algorithm; //算法工作模式
uint32_t keyNum; //密钥个数
struct crypto_key_param keyParam[0]; //密钥结构体
};
```

参数名称	描述	取值
algorithm	数据加解密填充模式	0—no_padding 1—BLOCK_TYPE_0 2—BLOCK_TYPE_1 3—BLOCK_TYPE_2 4—RSAES_OAEP_SHA1 5—RSAES_OAEP_SHA224 6—RSAES_OAEP_SHA256 7—RSAES_OAEP_SHA384 8—RSAES_OAEP_SHA512 9—RSAES_PKCS1_V1_5 注：当数据与公钥长度相等无需填充时，选择no_padding模式。其余填充模式数据长度应满足算法要求。
keyNum	密钥个数	公钥加解密密钥个数为2，按顺序为N、E密钥。 私钥加解密包括两种形式，一种个数为2，按顺序为N、D密钥，另一种个数为6，按顺序为N、P、Q、DP、DQ、QP密钥
keyParam[0]	密钥结构体	密钥长度可以为128、256、374、512位，其中3516DV300平台不支持374位

struct crypto_key_param {

uint32_t keyLength; //密钥长度

unsigned char key[0]; //密钥

};

参数名称	描述	取值
keyLength	密钥长度	-
key[0]	密钥	-

?.3. 请求扩展头

无

?.4. 响应码

如果功能正常，响应码200，其他为错误码。

?.5. 响应 Content

无

?.6. 响应扩展头

无。

?.7. 参考样例

5.8.5.2.2 RSA 私钥解密计算

?.1. 请求 Common Head 方法资源

方法	URL	VALUE
UPDATE	CRYPTO_URI_RSA_PRIVATE_DECRYPT	7

?.2. 请求 Content

struct crypto_config_data{

uint32_t dataStatus; //数据状态

uint32_t dataNum; //加解密数据个数

struct crypto_data_param data[0]; //加解密数据结构体

struct crypto_data_param tag[0]; //tag值

};

参数名称	描述	取值
dataStatus	数据状态	0—update。 1—final。 注：加解密采用流式加解密方式，内部16字节对齐，当状态为final时会把对齐剩余数据结果返回
dataNum	加解密数据个数	-
data[0]	加解密数据结构体	见下表
tag[0]	TAG值。CCM/GCM模式专用参数	当前接口无TAG值。

struct crypto_data_param{

uint32_t length; //数据长度

unsigned char data[0]; //数据

};

参数名称	描述	取值
length	数据长度	-
data[0]	数据	-

?.3. 请求扩展头

无

?.4. 响应码

如果功能正常，响应码200，其他为错误码。

?.5. 响应 Content

struct crypto_config_data；具体参数参照请求content

?.6. 响应扩展头

无。

?.7. 参考样例

5.8.5.2.3 销毁 RSA 私钥解密

?.1. 请求 Common Head 方法资源

方法	URL	VALUE
DELETE	CRYPTO_URI_RSA_PRIVATE_DECRYPT	7

?.2. 请求 Content

无

?.3. 请求扩展头

无

?.4. 响应码

如果功能正常，响应码200，其他为错误码。

?.5. 响应 Content

无

?.6. 响应扩展头

无

?.7. 参考样例

5.8.6 RSA 私钥加密公钥解密

5.8.6.1 RSA 私钥加密

5.8.6.1.1 创建 RSA 私钥加密

?.1. 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	CRYPTO_URI_RSA_PRIVATE_ENCRYPT	8

?.2. 请求 Content

```
struct crypto_hash_rsa_config{
uint32_t algorithm; //算法工作模式
uint32_t keyNum; //密钥个数
struct crypto_key_param keyParam[0]; //密钥结构体
};
```

参数名称	描述	取值
algorithm	数据加解密填充模式	0—no_padding 1—BLOCK_TYPE_0 2—BLOCK_TYPE_1 3—BLOCK_TYPE_2 4—RSAES_OAEP_SHA1 5—RSAES_OAEP_SHA224 6—RSAES_OAEP_SHA256 7—RSAES_OAEP_SHA384 8—RSAES_OAEP_SHA512 9—RSAES_PKCS1_V1_5 注：当数据与公钥长度相等无需填充时，选择no_padding模式。其余填充模式数据长度应满足算法要求。
keyNum	密钥个数	公钥加解密密钥个数为2，按顺序为N、E密钥。 私钥加解密包括两种形式，一种个数为2，按顺序为N、D密钥，另一种个数为6，按顺序为N、P、Q、DP、DQ、QP密钥
keyParam[0]	密钥结构体	密钥长度可以为128、256、374、512位，其中3516DV300平台不支持374位

struct crypto_key_param {

uint32_t keyLength; //密钥长度

unsigned char key[0]; //密钥

};

参数名称	描述	取值
keyLength	密钥长度	-
key[0]	密钥	-

?.3. 请求扩展头

无

?.4. 响应码

如果功能正常，响应码200，其他为错误码。

?.5. 响应 Content

无

?.6. 响应扩展头

无。

?.7. 参考样例

5.8.6.1.2 RSA 私钥加密计算

?.1. 请求 Common Head 方法资源

方法	URL	VALUE
UPDATE	CRYPTO_URI_RSA_PRIVATE_ENCRYPT	8

?.2. 请求 Content

struct crypto_config_data{

uint32_t dataStatus; //数据状态

uint32_t dataNum; //加解密数据个数

struct crypto_data_param data[0]; //加解密数据结构体

struct crypto_data_param tag[0]; //tag值

};

参数名称	描述	取值
dataStatus	数据状态	0—update。 1—final。 注：加解密采用流式加解密方式，内部16字节对齐，当状态为final时会把对齐剩余数据结果返回
dataNum	加解密数据个数	-
data[0]	加解密数据结构体	见下表
tag[0]	TAG值。CCM/GCM模式专用参数	当前接口无TAG值。

struct crypto_data_param{

uint32_t length; //数据长度

unsigned char data[0]; //数据

};

参数名称	描述	取值
length	数据长度	-
data[0]	数据	-

?.3. 请求扩展头

无

?.4. 响应码

如果功能正常，响应码200，其他为错误码。

?.5. 响应 Content

struct crypto_config_data；具体参数参照请求content

?.6. 响应扩展头

无。

?.7. 参考样例

5.8.6.1.3 销毁 RSA 私钥加密

?.1. 请求 Common Head 方法资源

方法	URL	VALUE
DELETE	CRYPTO_URI_RSA_PRIVATE_ENCRYPT	8

?.2. 请求 Content

无

?.3. 请求扩展头

无

?.4. 响应码

如果功能正常，响应码200，其他为错误码。

?.5. 响应 Content

?.6. 响应扩展头

无。

?.7. 参考样例

5.8.6.2 RSA 公钥解密

5.8.6.2.1 创建 RSA 公钥解密

?.1. 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	CRYPTO_URI_RSA_PUBLIC_DECRYPT	9

?.2. 请求 Content

```
struct crypto_hash_rsa_config{
uint32_t algorithm; //算法工作模式
uint32_t keyNum; //密钥个数
struct crypto_key_param keyParam[0]; //密钥结构体
};
```

参数名称	描述	取值
algorithm	数据加解密填充模式	0—no_padding 1—BLOCK_TYPE_0 2—BLOCK_TYPE_1 3—BLOCK_TYPE_2 4—RSAES_OAEP_SHA1 5—RSAES_OAEP_SHA224 6—RSAES_OAEP_SHA256 7—RSAES_OAEP_SHA384 8—RSAES_OAEP_SHA512 9—RSAES_PKCS1_V1_5 注：当数据与公钥长度相等无需填充时，选择no_padding模式。其余填充模式数据长度应满足算法要求。
keyNum	密钥个数	公钥加解密密钥个数为2，按顺序为N、E密钥。 私钥加解密包括两种形式，一种个数为2，按顺序为N、D密钥，另一种个数为6，按顺序为N、P、Q、DP、DQ、QP密钥
keyParam[0]	密钥结构体	密钥长度可以为128、256、374、512位，其中3516DV300平台不支持374位

struct crypto_key_param {

uint32_t keyLength; //密钥长度

unsigned char key[0]; //密钥

};

参数名称	描述	取值
keyLength	密钥长度	-
key[0]	密钥	-

?.3. 请求扩展头

无

?.4. 响应码

如果功能正常，响应码200，其他为错误码。

?.5. 响应 Content

?.6. 响应扩展头

无。

?.7. 参考样例

5.8.6.2.2 RSA 公钥解密计算

?.1. 请求 Common Head 方法资源

方法	URL	VALUE
UPDATE	CRYPTO_URI_RSA_PUBLIC_DECRYPT	9

?.2. 请求 Content

struct crypto_config_data{

uint32_t dataStatus; //数据状态

uint32_t dataNum; //加解密数据个数

struct crypto_data_param data[0]; //加解密数据结构体

struct crypto_data_param tag[0]; //tag值

};

参数名称	描述	取值
dataStatus	数据状态	0—update。 1—final。 注：加解密采用流式加解密方式，内部16字节对齐，当状态为final时会把对齐剩余数据结果返回
dataNum	加解密数据个数	-
data[0]	加解密数据结构体	见下表
tag[0]	TAG值。CCM/GCM模式专用参数	当前接口无TAG值。

struct crypto_data_param{

uint32_t length; //数据长度

unsigned char data[0]; //数据

};

参数名称	描述	取值
length	数据长度	-
data[0]	数据	-

?.3. 请求扩展头

无

?.4. 响应码

如果功能正常，响应码200，其他为错误码。

?.5. 响应 Content

struct crypto_config_data；具体参数参照请求content

?.6. 响应扩展头

无。

?.7. 参考样例

5.8.6.2.3 销毁 RSA 公钥解密

?.1. 请求 Common Head 方法资源

方法	URL	VALUE
DELETE	CRYPTO_URI_RSA_PUBLIC_DECRYPT	9

?.2. 请求 Content

无

?.3. 请求扩展头

无

?.4. 响应码

如果功能正常，响应码200，其他为错误码。

?.5. 响应 Content

无

?.6. 响应扩展头

无。

?.7. 参考样例

5.8.7 RSA 数据签名

5.8.7.1 创建 RSA 签名参数

5.8.7.1.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	CRYPTO_URI_RSA_SIGN	10

5.8.7.1.2 请求 Content

```
struct crypto_hash_rsa_config {
uint32_t algorithm; //算法工作模式
uint32_t keyNum; //密钥个数
struct crypto_key_param keyParam[0]; //密钥结构体
};
```

参数名称	描述	取值
algorithm	数据加解密填充模式	0x100--RSASSA_PKCS1_V15_SHA1 0x101--RSASSA_PKCS1_V15_SHA224 0x102--RSASSA_PKCS1_V15_SHA256 0x103--RSASSA_PKCS1_V15_SHA384 0x104--RSASSA_PKCS1_V15_SHA512 0x105--RSASSA_PKCS1_PSS_SHA1 0x106--RSASSA_PKCS1_PSS_SHA224 0x107--RSASSA_PKCS1_PSS_SHA256 0x108--RSASSA_PKCS1_PSS_SHA384 0x109--RSASSA_PKCS1_PSS_SHA512
keyNum	密钥个数	签名密钥包括两种形式，一种个数为2，按顺序为N、D密钥，另一种个数为6，按顺序为N、P、Q、DP、DQ、QP密钥
keyParam[0]	密钥结构体	密钥长度可以为128、256、374、512位，其中3516DV300平台不支持374位

struct crypto_key_param {

uint32_t keyLength; //密钥长度

unsigned char key[0]; //密钥

};

参数名称	描述	取值
keyLength	密钥长度	-
key[0]	密钥	-

5.8.7.1.3 请求扩展头

无

5.8.7.1.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.7.1.5 响应 Content

5.8.7.1.6 响应扩展头

无。

5.8.7.1.7 参考样例

5.8.7.2 RSA 数据签名

5.8.7.2.1 请求 Common Head 方法资源

方法	URL	VALUE
UPDATE	CRYPTO_URI_RSA_SIGN	10

5.8.7.2.2 请求 Content

struct crypto_config_data{

uint32_t dataStatus; //数据状态

uint32_t dataNum; //加解密数据个数

struct crypto_data_param data[0]; //加解密数据结构体

struct crypto_data_param tag[0]; //tag值

};

参数名称	描述	取值
dataStatus	数据状态	0—update。 1—final。 注：加解密采用流式加解密方式，内部16字节对齐，当状态为final时会把对齐剩余数据结果返回
dataNum	加解密数据个数	注：数据签名不支持多包签名
data[0]	加解密数据结构体	见下表
tag[0]	TAG值。CCM/GCM模式专用参数	当前接口无TAG值。

struct crypto_data_param{

uint32_t length; //数据长度

unsigned char data[0]; //数据

};

参数名称	描述	取值
length	数据长度	-
data[0]	数据	-

数据数量可以为1个或者2个，第一个数据为原始数据，第二个数据为数据的Hash摘要

5.8.7.2.3 请求扩展头

无

5.8.7.2.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.7.2.5 响应 Content

struct crypto_config_data;

5.8.7.2.6 响应扩展头

无。

5.8.7.2.7 参考样例

5.8.7.3 销毁 RSA 签名

5.8.7.3.1 请求 Common Head 方法资源

方法	URL	VALUE
DELETE	CRYPTO_URI_RSA_SIGN	10

5.8.7.3.2 请求 Content

5.8.7.3.3 请求扩展头

无

5.8.7.3.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.7.3.5 响应 Content

5.8.7.3.6 响应扩展头

无。

5.8.7.3.7 参考样例

5.8.8 RSA 数据验签

5.8.8.1 创建 RSA 数据验签

5.8.8.1.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	CRYPTO_URI_RSA_VERIFY	11

5.8.8.1.2 请求 Content

```
struct crypto_config_data{
uint32_t algorithm; //算法工作模式
uint32_t keyNum; //密钥个数
struct crypto_key_param keyParam[0]; //密钥结构体
};
```

参数名称	描述	取值
algorithm	数据加解密填充模式	0x100--RSASSA_PKCS1_V15_SHA1 0x101--RSASSA_PKCS1_V15_SHA224 0x102--RSASSA_PKCS1_V15_SHA256 0x103--RSASSA_PKCS1_V15_SHA384 0x104--RSASSA_PKCS1_V15_SHA512 0x105--RSASSA_PKCS1_PSS_SHA1 0x106--RSASSA_PKCS1_PSS_SHA224 0x107--RSASSA_PKCS1_PSS_SHA256 0x108--RSASSA_PKCS1_PSS_SHA384 0x109--RSASSA_PKCS1_PSS_SHA512
keyNum	密钥个数	公钥加解密密钥个数为2，按顺序为N、E密钥。
keyParam[0]	密钥结构体	密钥长度可以为128、256、374、512位，其中3516DV300平台不支持374位

struct crypto_key_param {

uint32_t keyLength; //密钥长度

unsigned char key[0]; //密钥

};

参数名称	描述	取值
keyLength	密钥长度	-
key[0]	密钥	-

5.8.8.1.3 请求扩展头

无

5.8.8.1.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.8.1.5 响应 Content

5.8.8.1.6 响应扩展头

无。

5.8.8.1.7 参考样例

5.8.8.2 RSA 数据验签

5.8.8.2.1 请求 Common Head 方法资源

方法	URL	VALUE
UPDATE	CRYPTO_URI_RSA_VERIFY	11

5.8.8.2.2 请求 Content

struct crypto_config_data{

uint32_t dataStatus; //数据状态

uint32_t dataNum; //加解密数据个数

struct crypto_data_param data[0]; //加解密数据结构体

struct crypto_data_param tag[0]; //tag值

};

参数名称	描述	取值
dataStatus	数据状态	0—update。 1—final。 注：加解密采用流式加解密方式，内部16字节对齐，当状态为final时会把对齐剩余数据结果返回
dataNum	加解密数据个数	注：数据签名不支持多包签名
data[0]	加解密数据结构体	见下表
tag[0]	TAG值。CCM/GCM模式专用参数	当前接口无TAG值。

```
struct crypto_data_param{
uint32_t length; //数据长度
unsigned char data[0]; //数据
};
```

参数名称	描述	取值
length	数据长度	-
data[0]	数据	-

数据个数为2个或者3个，按顺序为原始数据、签名数据、hash摘要。

5.8.8.2.3 请求扩展头

无

5.8.8.2.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.8.2.5 响应 Content

5.8.8.2.6 响应扩展头

无。

5.8.8.2.7 参考样例

5.8.8.3 销毁数据验签

5.8.8.3.1 请求 Common Head 方法资源

方法	URL	VALUE
DELETE	CRYPTO_URI_RSA_VERIFY	11

5.8.8.3.2 请求 Content

5.8.8.3.3 请求扩展头

无

5.8.8.3.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.8.3.5 响应 Content

5.8.8.3.6 响应扩展头

无。

5.8.8.3.7 参考样例

5.8.9 HASH 参数

5.8.9.1 创建 HASH 参数

5.8.9.1.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	CRYPTO_URI_HASH	4

5.8.9.1.2 请求 Content

struct crypto_hash_rsa_config{

uint32_t algorithm; //算法工作模式

uint32_t keyNum; //密钥个数

struct crypto_key_param keyParam[0]; //密钥结构体

}CRYPTO_HASH_RSA_CONFIG;

参数名称	描述	取值
algorithm	算法工作模式	0—SHA1 1—SHA224 2—SHA256 3—SHA384 4—SHA512
keyNum	密钥个数	Hash运算无需密钥，不需要填写
keyParam[0]	密钥结构体	Hash运算无需密钥

5.8.9.1.3 请求扩展头

无

5.8.9.1.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.9.1.5 响应 Content

无

5.8.9.1.6 响应扩展头

无。

5.8.9.1.7 参考样例

5.8.9.2 HASH 计算

5.8.9.2.1 请求 Common Head 方法资源

方法	URL	VALUE
UPDATE	CRYPTO_URI_HASH	4

5.8.9.2.2 请求 Content

struct crypto_config_data{

uint32_t dataStatus; //数据状态

uint32_t dataNum; //加解密数据个数

struct crypto_data_param data[0]; //加解密数据结构体

struct crypto_data_param tag[0]; //tag值

};

参数名称	描述	取值
dataStatus	数据状态	0—update。 1—final。 注：hash计算时状态为update更新数据，内部64字节对齐。状态为final时返回hash值
dataNum	加解密数据个数	-
data[0]	加解密数据结构体	见下表
tag[0]	TAG值。CCM/GCM模式专用参数	当前接口无TAG值。

struct crypto_data_param{

uint32_t length; //数据长度

unsigned char data[0]; //数据

};

参数名称	描述	取值
length	数据长度	-
data[0]	数据	-

5.8.9.2.3 请求扩展头

无

5.8.9.2.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.9.2.5 响应 Content

若数据状态是0—update时，不返回数据。若为1—final时返回hash值，格式为Struct CRYPTO_CONFIG_DATA。

Hash值对应长度见下表。

Hash类型	Hash值长度
SHA1	20
SHA224	28
SHA256	32
SHA384	48
SHA512	64

5.8.9.2.6 响应扩展头

无。

5.8.9.2.7 参考样例

5.8.9.3 销毁 HASH 参数

5.8.9.3.1 请求 Common Head 方法资源

方法	URL	VALUE
DELETE	CRYPTO_URI_HASH	4

5.8.9.3.2 请求 Content

无

5.8.9.3.3 请求扩展头

无

5.8.9.3.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.9.3.5 响应 Content

无

5.8.9.3.6 响应扩展头

无。

5.8.9.3.7 参考样例

5.8.10 HMAC 参数

5.8.10.1 创建 HMAC 参数

5.8.10.1.1 请求 Common Head 方法资源

方法	URL	VALUE
CREATE	CRYPTO_URI_HMAC	5

5.8.10.1.2 请求 Content

struct crypto_hash_rsa_config{

uint32_t algorithm; //算法工作模式

uint32_t keyNum; //密钥个数

struct crypto_key_param keyParam[0]; //密钥结构体

}CRYPTO_HASH_RSA_CONFIG;

参数名称	描述	取值
algorithm	算法工作模式	5—HMAC_SHA1 6—HMAC_SHA224 7—HMAC_SHA256 8—HMAC_SHA384 9—HMAC_SHA512
keyNum	密钥个数	Hmac运算需要密钥，数量为1
keyParam[0]	密钥结构体	Hmac运算需要密钥

struct crypto_key_param {

uint32_t keyLength; //密钥长度

unsigned char key[0]; //密钥

};

参数名称	描述	取值
keyLength	密钥长度	-
key[0]	密钥	-

5.8.10.1.3 请求扩展头

无

5.8.10.1.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.10.1.5 响应 Content

无

5.8.10.1.6 响应扩展头

无。

5.8.10.1.7 参考样例

5.8.10.2 HMAC 计算

5.8.10.2.1 请求 Common Head 方法资源

方法	URL	VALUE
UPDATE	CRYPTO_URI_HMAC	5

5.8.10.2.2 请求 Content

struct crypto_config_data{

uint32_t dataStatus; //数据状态

uint32_t dataNum; //加解密数据个数

struct crypto_data_param data[0]; //加解密数据结构体

struct crypto_data_param tag[0]; //tag值

};

参数名称	描述	取值
dataStatus	数据状态	0—update。 1—final。 注：hash计算时状态为update更新数据，内部64字节对齐。状态为final时返回hash值
dataNum	加解密数据个数	-
data[0]	加解密数据结构体	见下表
tag[0]	TAG值。CCM/GCM模式专用参数	当前接口无TAG值。

struct crypto_data_param{

uint32_t length; //数据长度

unsigned char data[0]; //数据

};

参数名称	描述	取值
length	数据长度	-
data[0]	数据	-

5.8.10.2.3 请求扩展头

无

5.8.10.2.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.10.2.5 响应 Content

若数据状态是0—update时，不返回数据。若为1—final时返回hash值，格式为Struct CRYPTO_CONFIG_DATA。

Hash值对应长度见下表。

Hash类型	Hash值长度
SHA1	20
SHA224	28
SHA256	32
SHA384	48
SHA512	64

5.8.10.2.6 响应扩展头

无。

5.8.10.2.7 参考样例

5.8.10.3 销毁 HMAC 参数

5.8.10.3.1 请求 Common Head 方法资源

方法	URL	VALUE
DELETE	CRYPTO_URI_HMAC	5

5.8.10.3.2 请求 Content

无

5.8.10.3.3 请求扩展头

无

5.8.10.3.4 响应码

如果功能正常，响应码200，其他为错误码。

5.8.10.3.5 响应 Content

无

5.8.10.3.6 响应扩展头

无。

5.8.10.3.7 参考样例

5.8.11 随机数

5.8.11.1 获取随机数

5.8.11.1.1 请求 Common Head 方法资源

方法	URL	VALUE
GET	CRYPTO_URI_RANDOM	12

5.8.11.1.2 请求 Content

uint32_t randomLen 随机数长度，最大支持128位。

5.8.11.1.3 请求扩展头

无

5.8.11.1.4 响应码

如果功能正常，响应码200，其他为错误码

5.8.11.1.5 响应 Content

char random[] 所需长度随机数

5.8.11.1.6 响应扩展头

无

5.8.11.1.7 参考样例

5.9 osd.iaas.sdc 服务化接口定义

5.9.1 功能定义

提供一种可供用户显示OSD的方法，通过这种方法用户可以显示任意信息，OSD采用key-value的方式，用户首先要注册一个环境变量，即key，然后要添加环境变量的描述，描述的类型由设备支持的语言种类决定，添加描述完成后，在页面上就可以展示出已经注册的环境变量的列表，点击插入即可，最后通过更新value值就可以控制OSD内容的显示。

5.9.2 注册环境变量

5.9.2.1 请求 Common Head 方法资源

METHOD	URI含义	URI取值
CREATE	SDC_URI_OSD_ENV_NAME	21

5.9.2.2 请求 Content

typedef struct _RegisterEnvName {

char name[128]; //注册的环境变量的名称，即key 值

}RegisterEnvName;

参数名称	描述	取值
name	注册的环境变量的名称，即key值	需满足如下要求： 1、utf-8编码 2、固定格式${xxx}，如${ptz}，如${123}

5.9.2.3 请求扩展头

无

5.9.2.4 响应码

如果功能正常，响应码200， 其他为错误码。

5.9.2.5 响应 Content

无

5.9.2.6 响应扩展头

无。

5.9.2.7 参考样例

无

5.9.3 添加环境变量描述

5.9.3.1 请求 Common Head 方法资源

METHOD	URI含义	URI取值
CREATE	SDC_URI_OSD_ENV_DESC	24

5.9.3.2 请求 Content

typedef enum _LangType {

LANG_TYPE_ZH = 0x5A48,//中文

LANG_TYPE_EN = 0x454E,//英文

LANG_TYPE_ES = 0x4553,//西班牙语

LANG_TYPE_FR = 0x4652,//法语

}LangType;

typedef struct _EnvDesc {

LangType descLangType;

char envName[128]; /*!< Detailed envName: 环境变量的名称*/

char envDesc[128]; /*!<Detailed envDesc:供给页面展示的语言类型>*/

} EnvDesc;/* optional variable list */;

参数名称	描述	取值
descLangType	描述的语言类型	LANG_TYPE_ZH = 0x5A48,//中文 LANG_TYPE_EN = 0x454E,//英文 LANG_TYPE_ES = 0x4553,//西班牙语 LANG_TYPE_FR = 0x4652,//法语
envName	已注册的环境变量的名称	需满足如下要求： 1、utf-8编码 2、固定格式${xxx}，如${ptz}，如${123}

envDesc	对应语言的描述	需要满足以下要求： 1、必须为utf-8编码 示例：云镜信息在中文、英文、西班牙语、法语对应的描述为： 云镜信息（中文） PTZ information（英文） Informations PTZ（法语） Información PTZ（西班牙语）

5.9.3.3 请求扩展头

无

5.9.3.4 响应码

如果功能正常，响应码200， 其他为错误码。

5.9.3.5 响应 Content

5.9.3.6 响应扩展头

无。

5.9.3.7 参考样例

无

5.9.4 环境变量对应的 value 值更新

5.9.4.1 请求 Common Head 方法资源

METHOD	URI含义	URI取值
UPDATE	SDC_URI_OSD_ENV_VALUE	23

5.9.4.2 请求 Content

```
typedef struct _EnvInfo {
char envName[128];
char envValue[128];
```


}EnvInfo;

参数名称	描述	取值
envName	已经注册的环境变量	需满足如下要求： 1、utf-8编码 2、固定格式如环境变量${ptz}，输入值为ptz
envValue	需要显示的OSD内容	需满足如下要求： 1、utf-8编码

5.9.4.3 请求扩展头

无

5.9.4.4 响应码

如果功能正常，响应码200， 其他为错误码。

5.9.4.5 响应 Content

无

5.9.4.6 响应扩展头

无。

5.9.4.7 参考样例

```
#include "sdc.h"
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <stddef.h>
#include <errno.h>
#include <sys/uio.h>

typedef struct _RegisterEnvName {
char name[128]; //注册的环境变量的名称，即key 值
}RegisterEnvName;
```



```
typedef enum _LangType {
LANG_TYPE_ZH = 0x5A48,//中文
LANG_TYPE_EN = 0x454E,//英文
LANG_TYPE_ES = 0x4553,//西班牙语
LANG_TYPE_FR = 0x4652,//法语
}LangType;

typedef struct _EnvDesc {
LangType descLangType;
char envName[128]; /*!< Detailed envName: 环境变量的名称*/
char envDesc[128]; /*!<Detailed envDesc:供给页面展示的语言类型>*/
} EnvDesc;/* optional variable list */;

typedef struct _EnvInfo {
char envName[128];
char envValue[128];
}EnvInfo;

#define SDC_URI_OSD_ENV_NAME (21)
#define SDC_URI_OSD_ENV_DESC (24)
#define SDC_URI_OSD_ENV_VALUE (23)

int RegisterOsdEnvName()
{
int fd = -1;
int nret= -1;
RegisterEnvName envName = {0} ;
memcpy(envName.name, "${ptX}", sizeof(envName.name));
struct sdc_common_head head = {
.version = SDC_VERSION,
.url = SDC_URI_OSD_ENV_NAME,
.method = SDC_METHOD_CREATE,
.head_length = sizeof(head),
.content_length = sizeof(envName),
};
```



```
struct iovec iov[] = {
{ (void*)&head, sizeof(head) },
{ &envName, sizeof(envName) },
};
fd = open("/mnt/srvfs/osd.iaas.sdc", O_RDWR);
if (fd < 0) {
printf("open /mnt/srvfs/osd.iaas.sdc fail.\n");
return -1;
}
nret = writev(fd, iov, sizeof(iov)/sizeof(iov[0]));
if(nret < 0) {
printf("write err,errno %d",errno);
close(fd);
return -1;
}
nret = read(fd,&head,sizeof(head));
if(nret < 0) {
printf("read ret %d,errno %d\n",nret,errno);
close(fd);
return -1;
}
if(head.code != 200) {
printf("rsp code %d\n",head.code);
close(fd);
return -1;
}
printf("set osd code %d \n",head.code);
close(fd);
return 0;
}

int AddDesOsdEnvName()
{
```



```
int fd = -1;
int nret= -1;
EnvDesc envDesc[4] = {0};
LangType langTypeArry[4] = {LANG_TYPE_ZH, LANG_TYPE_EN, LANG_TYPE_FR,
LANG_TYPE_ES};//四种语言类型，中文、英文、法语、西班牙语
char descArry[4][128] = {"云镜信息", "PTZ information","Informations
PTZ" ,"Información PTZ"};
for (uint32_t i = 0; i < 4; i++) {
envDesc[i].descLangType = langTypeArry[i];
(void)memcpy(envDesc[i].envName,"${ptX}", sizeof(envDesc[i].envName));
(void)memcpy(envDesc[i].envDesc,descArry[i], sizeof(envDesc[i].envDesc));
}
struct sdc_common_head head = {
.version = SDC_VERSION,
.url = SDC_URI_OSD_ENV_DESC,
.method = SDC_METHOD_CREATE,
.head_length = sizeof(head),
.content_length = sizeof(envDesc),
};

struct iovec iov[] = {
{ (void*)&head, sizeof(head) },
{ &envDesc, sizeof(envDesc) },
};
fd = open("/mnt/srvfs/osd.iaas.sdc", O_RDWR);
if (fd < 0) {
printf("open /mnt/srvfs/osd.iaas.sdc fail.\n");
return -1;
}
nret = writev(fd, iov, sizeof(iov)/sizeof(iov[0]));
if(nret < 0) {
printf("write err,errno %d",errno);
close(fd);
return -1;
```



```
}
nret = read(fd,&head,sizeof(head));
if(nret < 0) {
printf("read ret %d,errno %d\n",nret,errno);
close(fd);
return -1;
}
if(head.code != 200) {
printf("rsp code %d\n",head.code);
close(fd);
return -1;
}
printf("set osd code %d \n",head.code);
close(fd);
return 0;
}

int UpdateValueEnv()
{
int fd = -1;
int nret = -1;
EnvInfo envInfo = {0};
static int i = 3;
char buf[128] = {0};
snprintf(buf,128,"dasdasd %d",i++);
memcpy(envInfo.envName,"ptX", sizeof(envInfo.envName));
memcpy(envInfo.envValue,buf, sizeof(buf));
printf("value %s \n",buf);
struct sdc_common_head head = {
.version = SDC_VERSION,
.url = SDC_URI_OSD_ENV_VALUE,
.method = SDC_METHOD_UPDATE,
.head_length = sizeof(head),
.content_length = sizeof(envInfo),
```



```
};

struct iovec iov[] = {
{ (void*)&head, sizeof(head) },
{ &envInfo, sizeof(envInfo) },
};
fd = open("/mnt/srvfs/osd.iaas.sdc", O_RDWR);
if (fd < 0) {
printf("open /mnt/srvfs/osd.iaas.sdc fail.\n");
return -1;
}
nret = writev(fd, iov, sizeof(iov)/sizeof(iov[0]));
if(nret < 0) {
printf("write err,errno %d",errno);
close(fd);
return -1;
}
nret = read(fd,&head,sizeof(head));
if(nret < 0) {
printf("read ret %d,errno %d\n",nret,errno);
close(fd);
return -1;
}
if(head.code != 200) {
printf("rsp code %d\n",head.code);
close(fd);
return -1;
}
printf("set osd code %d \n",head.code);
close(fd);
return 0;
}

int main ()
```



```
{
RegisterOsdEnvName();
AddDesOsdEnvName();
while(1) {
UpdateValueEnv();
sleep(2);
}
}
```

5.10 audio.iaas.sdc 服务化接口定义

5.10.1 音频解码播放接口

播放音频数据（音频格式要求：wav文件、音频编码格式pcm、采样率8000HZ、文件大小不能超过75KB）。

5.10.1.1 请求 Common Head 方法资源

METHOD	URI含义	URI取值
UPDATE	SDC_**AUDIO_URI_ALARM_OUT**	1

5.10.1.2 请求 Content

typedef struct AudioAdecFrame

{

uint32_t enType;// 码流类型 赋值为:1004

uint32_t chnID;// 不需要关注

uint64_t uSize;// 音频大小

uint64_t ullPts; // 不需要关注

uint8_t data[0]; // 音频数据

}SDC_AUDIO_ADEC_FRAME_S;

特别说明1：码流类型赋值为:1004;

2：播放最新推送的音频数据；

3：如果语音对讲开启，则不能播放推送的音频数据;

函数原型：VOID AudioSrvAudioDataDec(struct hbtp_message *pstMsgReq, struct srvfs_connection *pstConn)

返回：无返回值

5.10.1.3 请求扩展头

无

5.10.1.4 响应码

如果功能正常，响应码200， 其他为错误码。

5.10.1.5 响应 Content

无

5.10.1.6 响应扩展头

无。

5.10.1.7 参考样例

```
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/uio.h>
#include <errno.h>

#define SDC_AUDIO_SERVER "/mnt/srvfs/audio.iaas.sdc"
#define SDC_VERSION 0x5331
#define SDC_URL_DEMO 0x01
#define MAXFD 100
#define SDC_METHOD_CREATE 1
#define SDC_METHOD_GET 2
#define SDC_METHOD_UPDATE 3
#define SDC_METHOD_DELETE 4
#define SDC_CODE_200 200
//实际路径需要写到app内部路径，切记！！！
#define AUDIO_ALARM_DEFAULT_AUDIO "/XX/XXX/audiofiles/5ZGK6K2m6Z
+zLEFsYXJtIFNvdW5kLndhdg=="

typedef unsigned char uint8_t;
typedef unsigned char uint8_t;
typedef unsigned short int uint16_t;
typedef unsigned int uint32_t;
```



```
typedef unsigned long long uint64_t;

typedef struct{
uint16_t version;
uint8_t url_ver;
uint8_t method: 7;
uint8_t response: 1;
uint16_t url;
uint16_t code;
uint16_t head_length;
uint16_t trans_id;
uint32_t content_length;
}sdc_common_head;

typedef struct AudioAdecFrame
{
uint32_t enType;// 码流类型 赋值为:1004
uint32_t chnID;// 不需要关注
uint64_t uSize;// 音频大小
uint64_t ullPts; // 不需要关注
uint8_t data[0]; // 音频数据
}SDC_AUDIO_ADEC_FRAME_S;

int main()
{
FILE *fp = fopen(AUDIO_ALARM_DEFAULT_AUDIO, "rb");
if (fp == NULL) {
printf("invalid param, filename[%s]open failed\r\n",
AUDIO_ALARM_DEFAULT_AUDIO);
return -1;
}

fseek(fp, 0, SEEK_END); // 文件指针移动到末尾
uint64_t fileLen = ftell(fp); // 获取文件长度

SDC_AUDIO_ADEC_FRAME_S *pAudioInfo = (SDC_AUDIO_ADEC_FRAME_S
*)calloc(1, sizeof(SDC_AUDIO_ADEC_FRAME_S) + sizeof(uint8_t) * fileLen);
```



```
rewind(fp); // 文件指针恢复到文件头位置
uint32_t nRet = fread(pAudioInfo->data, sizeof(uint8_t), fileLen, fp);
if (nRet <= 0) {
printf("fread Fail nRet is %d.", nRet);
fclose(fp);
free(pAudioInfo);
return -1;
}
fclose(fp);
pAudioInfo->enType = 1004;
pAudioInfo->uSize = fileLen;
//pAudioInfo.data = tmpAudioData;
sdc_common_head head = {
.version = SDC_VERSION, //0x5331
.url = SDC_URL_DEMO, //0x01
.method = SDC_METHOD_UPDATE,
.head_length = sizeof(head),
.content_length = sizeof(*pAudioInfo) + fileLen,
};

struct iovec iov[] = {
{.iov_base = &head, .iov_len = sizeof(head) },
{.iov_base = pAudioInfo, .iov_len = sizeof(*pAudioInfo) },
{.iov_base = pAudioInfo->data, .iov_len = fileLen },
};

int nFd = open(SDC_AUDIO_SERVER, O_RDWR);
printf("SDC_AUDIO_SERVER nFd is %d, %s.\n",nFd,strerror(errno));
printf("error num:%d.\n",errno);
if (nFd < 0) {
free(pAudioInfo);
return 0;
}
int nret = writev(nFd, iov, sizeof(iov)/sizeof(iov[0]));
```



```
printf("SDC_AUDIO_SERVER write nret is %d, %s.\n",nret,strerror(errno));
printf("error num:%d.\n",errno);
if(nret < 0) {
close(nFd);
free(pAudioInfo);
return 0;
}

nret = readv(nFd,iov,2);
printf("SDC_AUDIO_SERVER read nret is %d, %s.\n",nret,strerror(errno));
printf("error num:%d\n",errno);
printf("head code num is %d.\n",head.code);
if(head.code == SDC_CODE_200) { //.../** 200 */
}
else{ //...
}
close(nFd);
free(pAudioInfo);
return 0;
}
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/uio.h>
#include <errno.h>

#define SDC_AUDIO_SERVER "/mnt/srvfs/audio.iaas.sdc"
#define SDC_VERSION 0x01
#define SDC_URL_DEMO 0x01
#define MAXFD 100
#define SDC_METHOD_CREATE 1
#define SDC_METHOD_GET 2
#define SDC_METHOD_UPDATE 3
```



```
#define SDC_METHOD_DELETE 4

#define SDC_CODE_200 200

//文件路径为app内部实际文件打包且可读取的路径，参考实际情况修改！！！

#define AUDIO_ALARM_DEFAULT_AUDIO "/usr/alg_model/audiofiles/
5ZGK6K2m6Z+zLEFsYXJtIFNvdW5kLndhdg=="

typedef unsigned char uint8_t;

typedef unsigned char uint8_t;

typedef unsigned short int uint16_t;

typedef unsigned int uint32_t;

typedef unsigned long long uint64_t;

typedef struct{

uint16_t version;

uint8_t url_ver;

uint8_t method: 7;

uint8_t response: 1;

uint16_t url;

uint16_t code;

uint16_t head_length;

uint16_t trans_id;

uint32_t content_length;

}sdc_common_head;

typedef struct AudioAdecFrame

{

uint32_t enType;// 码流类型 赋值为:1004

uint32_t chnID;// 不需要关注

uint64_t uSize;// 音频大小

uint64_t ullPts; // 不需要关注

uint8_t data[0]; // 音频数据

}SDC_AUDIO_ADEC_FRAME_S;

int main()

{

FILE *fp = fopen(AUDIO_ALARM_DEFAULT_AUDIO, "rb");
```



```
if (fp == NULL) {
printf("invalid param, filename[%s]open failed\r\n",
AUDIO_ALARM_DEFAULT_AUDIO);
return -1;
}
fseek(fp, 0, SEEK_END); // 文件指针移动到末尾
uint64_t fileLen = ftell(fp); // 获取文件长度

SDC_AUDIO_ADEC_FRAME_S *pAudioInfo = (SDC_AUDIO_ADEC_FRAME_S
*)calloc(1, sizeof(SDC_AUDIO_ADEC_FRAME_S) + sizeof(uint8_t) * fileLen);

rewind(fp); // 文件指针恢复到文件头位置
uint32_t nRet = fread(pAudioInfo->data, sizeof(uint8_t), fileLen, fp);
if (nRet <= 0) {
printf("fread Fail nRet is %d.", nRet);
fclose(fp);
free(pAudioInfo);
return -1;
}
fclose(fp);
pAudioInfo->enType = 1004;
pAudioInfo->uSize = fileLen;
//pAudioInfo.data = tmpAudioData;
sdc_common_head head = {
.version = SDC_VERSION, //0x5331
.url = SDC_URL_DEMO, //0x01
.method = SDC_METHOD_UPDATE,
.head_length = sizeof(head),
.content_length = sizeof(*pAudioInfo) + fileLen,
};

struct iovec iov[] = {
{.iov_base = &head, .iov_len = sizeof(head) },
{.iov_base = pAudioInfo, .iov_len = sizeof(*pAudioInfo) },
{.iov_base = pAudioInfo->data, .iov_len = fileLen },
};
```



```
int nFd = open(SDC_AUDIO_SERVER, O_RDWR);
printf("SDC_AUDIO_SERVER nFd is %d, %s.\n",nFd,strerror(errno));
printf("error num:%d.\n",errno);
if (nFd < 0) {
free(pAudioInfo);
return 0;
}
int nret = writev(nFd, iov, sizeof(iov)/sizeof(iov[0]));
printf("SDC_AUDIO_SERVER write nret is %d, %s.\n",nret,strerror(errno));
printf("error num:%d.\n",errno);
if(nret < 0) {
close(nFd);
free(pAudioInfo);
return 0;
}

nret = readv(nFd,iov,2);
printf("SDC_AUDIO_SERVER read nret is %d, %s.\n",nret,strerror(errno));
printf("error num:%d\n",errno);
printf("head code num is %d.\n",head.code);
if(head.code == SDC_CODE_200) { //.../** 200 */
}else{ //...
}
close(nFd);
free(pAudioInfo);
return 0;
}
```


6 公共软件能力服务化接口参考

公共软件能力按照需要持续扩充，当前各服务功能描述和版本状态如下。

服务名	功能介绍	当前版本支持情况
appmgr.paas.sdc	提供app生命周期管理能力，提供app看门狗能力。	已支持
event.paas.sdc	通用事件订阅和分发的服务，提供灵活的事件订阅机制。具备事件数据基于零拷贝机制的发布和订阅能力。	已支持
gateway.paas.sdc	提供SDC所有APP以及OS自身北向SDK的统一接入入口。包括人机和机机界面	已支持
alarm.paas.sdc	提供业务告警联动功能。业务层告警可以触发EMAIL，录像等应急处理动作	已支持
tproxy.paas.sdc	提供TCP/UDP代理服务功能	已支持

所有服务的接口都基于SDC服务化接口规范HBTP来定义，具体请参考SDC 服务化接口总体概述。

6.1 appmgr.paas.sdc 服务化接口定义

6.1.1 app 看门狗

app看门狗接口为app提供看门狗服务，用于app进程死锁等异常场景下能够尽快停止并重启app。

6.1.1.1 请求 Common Head 方法资源

方法	URL	VALUE
UPDATE	SDC_URL_APP_WATCHDOG	0x01

6.1.1.2 请求 Content

struct appdog_op_req{

int32_t watchdog_time;

};

watchdog_time<0，表示不启动看门狗。默认关闭看门狗。

watchdog_time==0，表示立刻停止app。app被停止后，会在最迟15秒之后重新启动。

watchdog_time>0，表示看门狗时长，单位是毫秒，当app在watchdog_time时长内没有重置看门狗时长，则发送SIGKILL信号直接停止app的主进程，同时app的所有子进程也会被立刻停止。建议由app内部的业务进程定时重置看门狗的时长。app被停止后，会在最迟15秒之后重新启动。

6.1.1.3 请求扩展头

无。

6.1.1.4 响应码

该接口无响应。

6.1.1.5 响应 Content

无。

6.1.1.6 响应扩展头

无。

6.1.1.7 参考样例

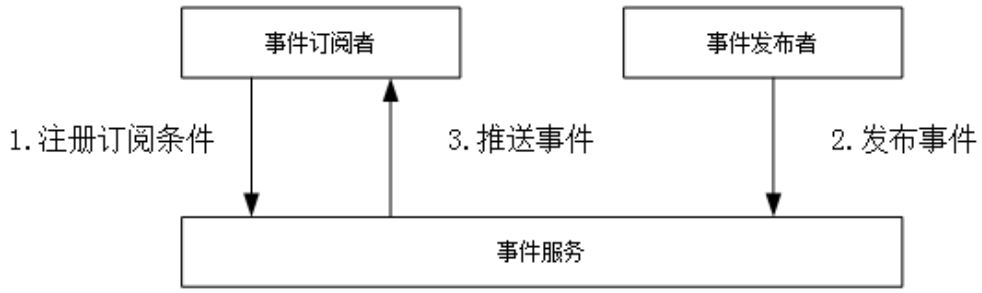
```
#include "sdc.h"
#include <errno.h>
```



```
#include <sys/uio.h>

int feed_watchdog(int fd, int32_t timer)
{
    struct appdog_op_req param = { .watchdog_time = timer };
        struct sdc_common_head head = {
        .version = SDC_VERSION, //0x5331
        .url = SDC_URL_APP_WATCHDOG, //0x01
        .method = SDC_METHOD_UPDATE,
        .head_length = sizeof(head),
        .content_length = sizeof(param)
    };
    struct iovec iov[] = {
        {.iov_base = &head, .iov_len = sizeof(head) },
        {.iov_base = &param, .iov_len = sizeof(param) }
    };
    if (writev(fd,iov,sizeof(iov)/sizeof(iov[0]))) return errno;
    return 0;
}
```

6.2 event.paas.sdc 服务化接口定义

6.2.1 功能定义



提供事件的发布和订阅机制，实现事件发布者和订阅者之间松耦合。

有以下约束：

1. 事件不具有持久化功能；
2. 订阅通道缓冲区满时则会丢弃最新事件。

6.2.2 事件发布接口

一次只发布一个事件。

6.2.2.1 请求 Common Head 方法资源

METHOD	URI含义	URI取值
CREATE	SDC_URL_PAAS_EVENTD_EVENT	0x00

6.2.2.2 请求 Content

如果未定义扩展头SDC_HEAD_ SHM_ CACHED_EVENT，则内容数据结构定义如下：

```
struct paas_event
{
char publisher[16]; //发送事件的服务标识，调测使用
char name[16]; //事件唯一标识，建议同域名定义避免冲突
uint64_t src_timestamp; //发生时的时间，单位毫秒（CLOCK_MONOTONIC时间）
uint64_t tran_timestamp; //服务转发的时间，单位毫秒（CLOCK_MONOTONIC时间）
uint32_t id; //建议同IP地址一样管理，不同前缀对应事件分类，方便分类订阅。
uint32_t length; //事件内容的长度.
char data[0];
};
```

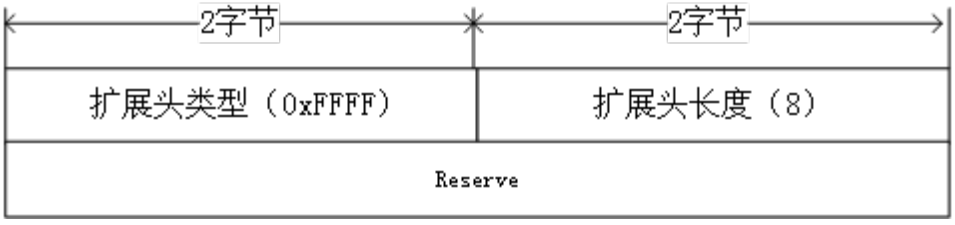
如果定义了扩展头SDC_HEAD_ SHM_ CACHED_EVENT，内容数据结构定义如下：

```
struct pass_shm_cached_event
{
uint64_t addr_phy;
uint32_t size;
uint32_t cookie;
};
```

其物理地址指向的空间存储的数据结构为struct paas_event，事件订阅者需要将物理地址映射为本进程的虚拟地址之后才可以使用，并在不需要访问数据的时候释放虚拟地址资源。参见2.2节 “SDC基于共享缓存机制”一节的介绍。

6.2.2.3 请求扩展头

6.2.2.3.1 SDC_HEAD_SHM_CACHED _EVENT）



忽略此扩展头Reserve字段的取值，只要定义了，就说明消息体内容数据格式为struct pass_shm_cached_event。

6.2.2.4 响应码

事件发布没有响应消息。

6.2.2.5 响应 Content

无。

6.2.2.6 响应扩展头

无。

6.2.2.7 参考样例

使用方法：publisher <name_prefix> <id_base> <event_cnt>

./publisher sample 100 3

publish event: name= sample_0, id = 100

publish event: name= sample_1, id = 101

publish event: name= sample_2, id = 102

```
#include "sdc.h"
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/uio.h>
#include <sys/mman.h>

struct sample_event
{
struct paas_event base;
int data[5000];
};

int main(int argc,char* argv[])
{
struct paas_shm_cached_event shm_event;
struct sdc_extend_head shm_head = {
.type = SDC_HEAD_SHM_CACHED_EVENT,
.length = 8,
};
struct sdc_common_head head = {
.version = SDC_VERSION,
.url = SDC_URL_PAAS_EVENTD_EVENT,
.method = SDC_METHOD_CREATE,
.head_length = sizeof(head) + sizeof(shm_head),
.content_length = sizeof(shm_event),
};
struct iovec iov[] = {
{.iov_base = &head, .iov_len = sizeof(head) },
{.iov_base = &shm_head, .iov_len = sizeof(shm_head) },
{.iov_base = &shm_event, .iov_len = sizeof(shm_event) },
};

struct sample_event * event ;

struct sdc_shm_cache shm_cache;

const char* name_prefix;
int fd,cache_fd,i,j,nret,id_base,event_cnt,event_len;

if(argc < 4) goto fail;

name_prefix = argv[1];
id_base = atoi(argv[2]);
```



```
event_cnt = atoi(argv[3]);
event_len = sizeof(event) + sizeof(head) + sizeof(shm_head);
fd = open("/mnt/srvfs/event.paas.sdc", O_RDWR);
if(fd < 0) goto fail;
cache_fd = open("/dev/cache", O_RDWR);
if(cache_fd < 0) goto fail;
for(;;) {
for(i = 0; i < event_cnt; ++i) {
shm_cache.size = sizeof(*event);
nret = ioctl(cache_fd, SDC_CACHE_ALLOC, &shm_cache);
if(nret) {
printf("cache alloc fail: %m\n");
goto fail;
}
event = shm_cache.addr_virt;
snprintf(event->base.name, sizeof(event->base.name), "%s_%d", name_prefix,i);
event->base.id = id_base + i;
for(j = 0; j < 10; ++j) {
event->data[j] = event->base.id + j;
event->data[5000 - j - 1] = event->base.id + j;
}
event->base.length = sizeof(event->data);
shm_event.addr_phy = shm_cache.addr_phy;
shm_event.size = shm_cache.size;
shm_event.cookie = shm_cache.cookie;
nret = writev(fd, iov, 3);
if(nret == -1) {
printf("writev fail: %m\n");
goto fail;
}
printf("publish event: name= %s, id = %d\n", event->base.name,event->base.id);
munmap(event, sizeof(*event));
usleep(1000 * 1000);
}
}
return 0;
fail:
return -1;
```



```
}
```

6.2.3 事件订阅接口

6.2.3.1 请求 Common Head 方法资源

METHOD	URI含义	URI取值
GET	SDC_URL_PAAS_EVENTD_EVENT	0x00

6.2.3.2 请求 Content

```
/**
* filter是订阅条件是一个布尔值表达式，只要表达式返回true（非0）就表示事件匹配成功
* 1. 订阅所有事件: filter = "1";
* 2. 基于掩码订阅事件: "event.id & 0xFFFF0000";
* 3. 订阅特定值的时间: "event.id in [2,3,9-30]";
* 4. 更复杂的订阅条件: "(event.id & 0x0000FFFF) in [20-30] && event.length > 0";
* 5. 根据事件的内容订阅: "event.body.u32[4] == 5";
*/
struct paas_event_filter
{
char subscriber[16]; //订阅者的标识, 调测使用
char name[16];
char filter[256];
};
```

支持一次订阅多个条件，条件数量 = hbtp.content_length / sizeof(struct paas_event_filter)。

订阅条件详细说明：

订阅条件是一个布尔值表达式，如果为空字符串等同于"1"，即订阅匹配了事件名的全部事件。

订阅条件支持以下几种方式访问事件数据的内容:

事件字段	含义	说明
event.id	获取事件的ID值进行运算	
event.length	获取事件的内容长度值进行运算	
event.body.u8[idx]	将body作为uint8_t的数组进行访问，获取指定位置的值进行运算	idx只支持常量，比如是10进制或者16进制数据，16机制必须以0x开头。
event.body.u16[idx]	将body作为uint16_t的数组进行访问，获取指定位置的值进行运算	idx只支持常量，比如是10进制或者16进制数据，16机制必须以0x开头。
event.body.u32[idx]	将body作为uint32_t的数组进行访问，获取指定位置的值进行运算	idx只支持常量，比如是10进制或者16进制数据，16机制必须以0x开头。
event.body.u64[idx]	将body作为uint64_t的数组进行访问，获取指定位置的值进行运算	idx只支持常量，比如是10进制或者16进制数据，16机制必须以0x开头。

支持如下操作符，其中双目运算符的优先级同C语言中的定义，下面的运算符按照优先级顺序定义。你可以使用括号()来改变计算的优先级。

运算符	说明	优先级
~	按位取反	最高优先级。同一优先级，从右至左运算。
!	非操作	
-	负数运算符	
*	乘号	同一优先级，从左至右运算
/	除号	
%	取余	
+	加法	同一优先级，从左至右运算
-	减法	
>>	右移	同一优先级，从左至右运算
<<	左移	
>	大于	同一优先级，从左至右运算
>=	大于等于	
<	小于	
<=	小于等于	

运算符	说明	优先级
==	等于	同一优先级，从左至右运算
！=	不等于	
&	与	同一优先级，从左至右运算
^	异或	同一优先级，从左至右运算
|	异或	同一优先级，从左至右运算
&&	逻辑与	同一优先级，从左至右运算
||	逻辑或	同一优先级，从左至右运算
IN	集合操作。集合必须在[] 中定义，不能为空。集合中的取值只能是常量，常量之间用","间隔，也支持通过"-"指定一个连续范围，是一个闭区间。比如[n-m]，表示>=n && <= m	备注： 1. IN不区分大小写 2. 前后必须有空格分离。

6.2.3.3 请求扩展头

无

6.2.3.4 响应码

如果订阅条件语法错误，则返回400；如果服务端错误（比如内存不足），则返回500；正常情况下响应码为200，和事件一起返回。

6.2.3.5 响应 Content

struct paas_event。

6.2.3.6 响应扩展头

无。

6.2.3.7 参考样例

使用方法： subscribe <event_name> <event_filter>

./subscriber "sample_1" "event.id > 1 && event.id <= 101 && event.body.u32[1] == 102"

name = sample_1, id = 101, length=40,
data=[101,102,103,104,105,106,107,108,109,110,]

subscribe stat: sample_1, cnt=1, fail_cnt=0

publis stat: sample_1, cnt=1, subscriber_cnt = 1, trans_cnt = 1, fail_cnt=0


```
#include "sdc.h"
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/uio.h>
#include <sys/mman.h>

struct sample_event
{
struct paas_event base;
int data[5000];
};

int main(int argc,char* argv[])
{
char buf[256];
struct paas_event_filter filter = { };
struct sdc_common_head* head = (void*)buf;
struct iovec iov[] = {
{.iov_base = head, .iov_len = sizeof(*head) },
{.iov_base = &filter, .iov_len = sizeof(filter) }
};

struct sdc_shm_cache shm_cache;
struct paas_shm_cached_event* shm_event ;
struct sample_event* event ;
struct paas_event_subscribe_stat* s_stat ;
struct paas_event_publish_stat* p_stat ;
int fd,cache_fd,n,i,nret;

memset(head, 0, sizeof(*head));
head->version = SDC_VERSION;
head->url = SDC_URL_PAAS_EVENTD_EVENT;
head->method = SDC_METHOD_GET;
head->head_length = sizeof(*head);
head->content_length = sizeof(filter);

if(argc < 3) {
printf("argc should be 3\n");
goto fail;
}

snprintf(filter.name,sizeof(filter.name),"%s", argv[1]);
```



```
snprintf(filter.filter,sizeof(filter.filter),"%s", argv[2]);

fd = open("/mnt/srvfs/event.paas.sdc", O_RDWR);
if(fd < 0) {
printf("open fail: %m\n");
goto fail;
}
cache_fd = open("/mnt/srvfs/proc/cache", O_RDWR);
if(cache_fd < 0) goto fail;

nret = writev(fd, iov, 2);
if(nret == -1) {
printf("write fail: %m\n");
goto fail;
}

for(;;) {
nret = read(fd,buf,sizeof(buf));
if(nret == -1) {
printf("read fail: %m\n");
goto fail;
}
if(head->code != SDC_CODE_200){
printf("code == %d\n", head->code);
goto fail;
}

switch(head->url) {
case SDC_URL_PAAS_EVENTD_EVENT:
shm_event = (void*)&buf[head->head_length];
shm_cache.addr_phy = shm_event->addr_phy;
shm_cache.size = shm_event->size;
shm_cache.cookie = shm_event->cookie;
nret = ioctl(cache_fd, SDC_CACHE_MMAP, &shm_cache);
if(nret) {
printf("mmap fail: %m\n");
continue;
}
event = shm_cache.addr_virt;

printf("\nname = %s, id = %d, length=%d, data=[", event->base.name,event-
>base.id,event->base.length);
for(i = 0; i < 10; ++i) { printf("%d,", event->data[5000 - i - 1]); }
```



```
printf("]\n");
memset(head,0,sizeof(*head));
head->version = SDC_VERSION;
head->url = SDC_URL_PAAS_EVENTD_SUBSCRIBE_STAT;
head->method = SDC_METHOD_GET,
head->head_length = sizeof(*head);
nret = write(fd,head,sizeof(*head));
if(nret == -1) {
printf("write 1 fail: %m\n");
goto fail;
}
head->url = SDC_URL_PAAS_EVENTD_PUBLISH_STAT;
head->content_length = sprintf(&buf[head->head_length],"%s",event->base.name);
nret = write(fd,head, head->head_length + head->content_length);
if(nret == -1) {
printf("write 2 fail: %m\n");
goto fail;
}
munmap(event,shm_cache.size);
break;
case SDC_URL_PAAS_EVENTD_SUBSCRIBE_STAT:
s_stat = (void*)&buf[head->head_length];
n = head->content_length / sizeof(*s_stat);
for(i = 0; i < n; ++i, ++s_stat) {
printf("subscribe stat: %s, cnt=%d, fail_cnt=%d\n", s_stat->name,(int) s_stat->cnt, (int)s_stat->fail_cnt);
}
break;
case SDC_URL_PAAS_EVENTD_PUBLISH_STAT:
p_stat = (void*)&buf[head->head_length];
printf("publis stat: %s, cnt=%d, subscriber_cnt = %d, trans_cnt = %d, fail_cnt= %d\n",
p_stat->name,(int) p_stat->cnt, p_stat->subscriber_cnt,(int) p_stat->trans_cnt, (int) p_stat->fail_cnt);
break;
default:
printf("unknown response\n");
goto fail;
```



```
}
}
return 0;
fail:
return -1;
}
```

6.2.4 订阅通道统计数据查询接口

6.2.4.1 请求 Common Head 方法资源

METHOD	URI含义	URI取值
GET	SDC_URL_PAAS_EVENTD_SUBSCRIBE_STAT	0x01

6.2.4.2 请求 Content

无

6.2.4.3 请求扩展头

无

6.2.4.4 响应码

如果服务端错误（比如内存不足），则返回500；正常情况下响应码为200。

6.2.4.5 响应 Content

```
struct paas_event_subscribe_stat
{
char name[16];
uint64_t cnt;
uint64_t fail_cnt;
};
```

每个订阅条件对应一个统计数据，数量= hbtp.content_length / sizeof(struct paas_event_subscribe_stat)。

6.2.4.6 响应扩展头

无。

6.2.4.7 参考样例

参见订阅接口

6.2.5 发布事件统计数据查询接口

6.2.5.1 请求 Common Head 方法资源

METHOD	URI含义	URI取值
GET	SDC_URL_PAAS_EVENTD_PUBLISH_STAT	0x02

6.2.5.2 请求 Content

char name[16];

基于事件标识查询统计数据。

6.2.5.3 请求扩展头

无

6.2.5.4 响应码

如果服务端错误（比如内存不足），则返回500；正常情况下响应码为200。

6.2.5.5 响应 Content

```
struct paas_event_publish_stat
{
char name[16];
uint32_t subscriber_cnt;
uint64_t cnt;
uint64_t trans_cnt; //成功转发的次数
uint64_t fail_cnt; //转发失败的次数
};
```


6.2.5.6 响应扩展头

无。

6.2.5.7 参考样例

参见订阅接口

6.2.6 智能元数据事件定义

智能业务的元数据是SDC中重要的一种事件，SDC中的北向接入协议以及流媒体会根据各自的需要订阅智能元数据的事件，或通过RESTful接口，或通过流媒体订阅接口将智能的元数据发送给最终的数据订阅者。

6.2.6.1 事件头定义

事件头数据结构为**struct paas_event**，其中：

name: 必须是"itgt.saas.sdc"

id: 必须是 0.

其他字段无约束。

6.2.6.2 元数据内容定义

内容为单层TLV（Type-Length-Value）格式。

参见《全网智能接口对接TLV数据详解》中“智能通用事件”章节的描述。

通用TLV数据量大小根据设备款型的最大分辨率大小进行区分限制。200W款型支持最大不超过1M，400W和500W款型支持最大不超过1.5M，800W以上款型支持最大不超过4M。

6.2.6.3 发布样例

参见事件发布接口的样例。

如果智能元数据包括了图片等大数据，建议用共享内存的方式传递数据，提升效率。通过共享内存传递事件的样例参考事件发布接口的样例代码。

6.3 gateway.paas.sdc 服务化接口定义

6.3.1 功能定义

APP安装到SDC上之后，APP可以通过注册，将自身的路由信息注册到网关服务gateway.paas.sdc：

1. APP注册后，可以在Web 首页APP桌面双击APP图标，进入APP的配置界面，对APP业务进行配置，一般为静态html资源；
2. APP注册后，APP自身可以提供RestAPI接口，gateway.paas.sdc可以将对应的请求消息路由到APP。

6.3.2 APP 注册接口

gateway.paas.sdc作为服务端，接受所有APP的注册请求。

6.3.2.1 请求 Common Head 方法资源

METHOD	URI含义	URI取值
CREATE	SDC_URL_PAAS_GATEWAY_REGISTER	0x00

6.3.2.2 请求 Content

struct paas_gateway

{

char register_path[256];

};

register_path：APP注册信息路径，路径需要空字符串结束，最长256字节，路径中必须包含网关配置文件portal.conf名称，且portal.conf配置文件必须被打包到APP包根目录下，portal.conf文件内容以及填写方法，见以下详解：

;

;Copyright (C) huawei

;Copyright (C) huawei, Inc.

;

;网关配置文件: 用于配置外部http请求到用户APP之间的消息路由

;

;表示注释

;文件编码必须为utf8,

;以下value字段不区分大小写

;所有行前、行尾不能存在空格，

;配置文件所有字段key、value区分大小写

;键值对符号分隔符=前后不能存在空格

;请求URI最长1023字节

;APP向网关服务注册的配置文件portal.conf路径长度不超过511字节

;配置文件的一行不超过1023字节

;业务自己的用户名和密码长度不超过63字节

;配置文件格式由华为定义，任何第三方开发机构不允许更改配置文件格式，以及配置项键值对中的key

;注释项建议用户也不要更改，但是用户可以自己添加注释，注释只能单独一行，不能追加到行尾

;portal configuration file version，目前只能为1

[common]

version=1

;app名称，主要用于Web界面显示app名称,app_name_en必须与rpm包中的appname相同，以在APP界面显示, 最长255字节,app_name目前没有使用

app_name=default

app_name_en=default

;可见性，可以为：on、off, 表示是否在界面上呈现该APP

visible=on

;app静态资源（比如图片、html）路径, 是所有静态资源的根路径

resource_path=./res/

;app配置入口页面,最终路径：./{resource_path}/{index}

index=./html/index.html

;正常情况下，用于APP界面显示的图片,最终路径：./{resource_path}/{normal_image}

normal_image=./image/normal.png

;鼠标上浮在APP图标上的APP图片,最终路径：./{resource_path}/{hover_image}，当前没有使用

hover_image=./image/hover.png

;app业务特性描述信息

desp=app is used to dectect.

;Web界面上，app的搜索关键字，当前没有使用

Key_word=车辆

[router_info]

;APP、服务对应的rest api消息路由目的地，

;APP和服务填写域套接字即可，长度不超过100字节

;一些特殊的业务对网关开放了端口号，则该字段需要为ip地址+端口号，一般为127.0.0.1即可

;地址类型：0表示域套接字，1表示IP地址，即AF_INET

app_dst_addr_type=0

app_dst_unix_addr=./httprest/sdc_rest_api

app_dst_ip_addr=127.0.0.1

;当app_dst_addr_type为ip地址时，该字段有效

business_dst_port=8080

;URI匹配关键字，满足URL命名规则，

;对于SDC API接口，配置示例如下：

;路由模块收到URI：/SDCAPI/V1.0/its/xxx, 则该字段为:/SDCAPI/V1.0/its/

;该字段用于后续的网关消息路由，与认证鉴权，非常关键

;该字段最长64字节

;realm表示认证域

router_url_key_word=/SDCAPI/V1.0/app_key_url/

;router_url_key_word所属的认证域

sdc_auth_realm=realm

;协议类型可以取值为:http、https、http\https,分别对应0、1、2，其它值非法,其中协议类型http\https表示会同时开两个端口80、443，

;protocol_type=http

protocol_type=https

;是否需要校验协议类型, 0为不需要，1为需要,目前只能填0

is_need_check_protocol_type=0

;是否启用SDC自带的认证模块, 0为不需要，1为需要

is_need_sdc_auth_module=1，配置时，需要注意，不存在只进行鉴权，而不认证的APP

;是否启用SDC自带的鉴权模块, 0为不需要，1为需要

is_need_sdc_resoure_auth_module=1

;APP与网关之间是否为长连接，只有一些特殊要求的APP采用短连接，如标准协议onvif，其它均为长连接

keep_alive=1

6.3.2.3 响应码

如果功能正常，响应码200，输入错误为400，服务端错误为500。

6.3.2.4 响应 Content

无

6.3.2.5 响应扩展头

无

6.3.2.6 参考样例

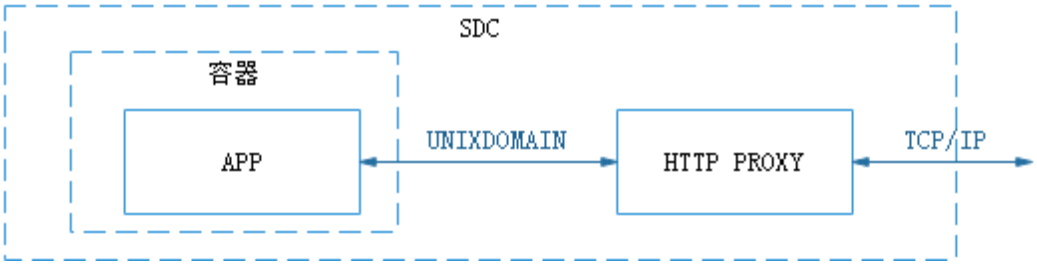
无

6.3.3 APP 注销接口

当APP需要在gateway.paas.sdc中注销时，只需要关闭APP与网关服务之间的连接即可。

6.3.4 基于 http proxy 建立网络连接

6.3.4.1 网络模型



APP运行环境中无法直接访问SDC的网络，需要通过SDC提供的HTTP PROXY建立到外网的连接。步骤如下：

1、创建AF_UNIX的SOCKET

2、连接到HTTP PROXY

3、发送HTTP CONNECT方法

4、接收HTTP响应

如果HTTP响应码是200说明建链成功。建链成功之后就可以利用此fd进行任何消息的收发处理。

6.3.4.2 参考样例

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
/**
* 创建一个连接外部IP:PORT地址的fd，此fd支持TCP传输
* RETURN 可读写的fd，基于此fd可传输任何私有协议数据。-1表示失败。
*/
static int http_proxy_connect(const char* ip, unsigned short port)
{
static const char* const connect_format = "CONNECT %s:%d HTTP/1.1\r\nHost:%s:
%d\r\n\r\n";
char buf[256];
int fd = -1, nret , len;
struct sockaddr_un addr = {};
addr.sun_family = AF_UNIX;
/** 后续应该是通过getenv("http.proxy")获取地址，当前可以先硬编码 */
(void)snprintf(addr.sun_path, sizeof(addr.sun_path), "%s", "/tmp/
http_proxy_connect.socket");

fd = socket(AF_UNIX, SOCK_STREAM, 0);
if(fd < 0) goto fail;
nret = connect(fd, (struct sockaddr*)&addr, sizeof(addr));
if(nret == -1) goto fail;
len = snprintf(buf, sizeof(buf), connect_format, ip, port, ip, port);
if(len > sizeof(buf)) {
printf("domain is too long\n");
goto fail;
}
nret = write(fd, buf, len);
if(nret < len) {
printf("write connect request fail\n");
```

```
goto fail;
}
/** 判断是否返回200 OK，如果是说明建连成功 */
nret = read(fd, buf, sizeof(buf));
if(nret == -1) goto fail;
if (!strstr(buf, "200")) {
printf("proxy connect fail: %s\n", buf);
goto fail;
}
return fd;
fail:
if(fd > -1) close(fd);
printf("fail: %m\n");
return -1;
}
```

6.4 alarm.paas.sdc 服务化接口定义

6.4.1 App 注册

所有app注册的告警最大数量为20个，当历史注册的告警数量超过20个时，无法注册新增告警，返回错误码403。

6.4.1.1 请求 Common Head 方法资源

METHOD	URI含义	URI取值
CREATE	SDC_URI_ALARM_SOURCE	1

6.4.1.2 请求 Content

/*
//切记注册文件路径为App内部路径！！！！
char alarmInputFile[]，告警app注册文件，当服务或设备重启时，需要重新注册，形式如下
app注册文件：
sourcelist=/xx/xx/xx/mysource.ini 告警列表，文件路径小于64字节。
appname=thirdapp app名称，app名称小于32字节。
language.ZH=/xx/xx/xxx/mylangZH.ini 资源文件中文，文件路径小于64字节。
language.EN=/xx/xx/xx/mylangEN.ini 资源文件英文，文件路径小于64字节。
language.FR=/xx/xx/xx/mylangFR.ini 资源文件法语，文件路径小于64字节。
language.ES=/xx/xx/xx/mylangES.ini 资源文件西班牙语，文件路径小于64字节。
*/
告警列表：
name=network 告警名称
desc=NET 告警描述
type=0 告警类型，0--故障告警，1--事件告警
alarmTime=1 取值范围0~256，若取值为0，则表示改时间不生效。故障告警对应自动恢复时间，事件告警对应告警间隔时间
资源文件：
“network” = “网络”

6.4.1.3 请求扩展头

无

6.4.1.4 响应码

如果功能正常，响应码200， 其他为错误码

6.4.1.5 响应 Content

无

6.4.1.6 响应扩展头

无。

6.4.1.7 参考样例

无

6.4.2 告警事件发布

告警事件发布接口。通过此接口发布的告警事件可以通过透明通道订阅接收，订阅方法与支持的告警类型参考文档《华为SDC9.0.0 API协议说明》。对外呈现的告警类型以各联动模块（28181、onvif、sdk、手机app等）实际支持的为准。

6.4.2.1 请求 Common Head 方法资源

METHOD	URI含义	URI取值
CREATE	SDC_URI_ALARM	**3**

6.4.2.2 请求 Content

typedef struct {

char alarmName[64];

char alarmSource[32];

char metaData[0];

}ALARM_REPORT_PARAM;

参数名称	参数描述	取值
alarmName	告警名称	
alarmSource	App名称	
metaData	元数据	用户数据，建议遵循《 全网智能接口对接TLV数据详解 》中“智能通用事件”的单层TLV数据格式的定义标准。

6.4.2.3 请求扩展头

无

6.4.2.4 响应码

如果功能正常，响应码200， 其他为错误码。

6.4.2.5 响应 Content

无

6.4.2.6 响应扩展头

无。

6.4.2.7 参考样例

无

6.4.3 告警事件消除

告警事件消除接口。通过此接口发布的告警消除事件可以通过透明通道订阅接收，订阅方法与支持的告警类型参考文档《华为SDC 8.2.0 SDCAPI协议说明》。对外呈现的告警类型以各联动模块（28181、onvif、sdk、手机app等）实际支持的为准。

6.4.3.1 请求 Common Head 方法资源

METHOD	URI含义	URI取值
DELETE	SDC_URI_ALARM	**3**

6.4.3.2 请求 Content

只有故障告警才可调用此接口

typedef struct {

char alarmName[64];

char alarmSource[32];

char metaData[0];

}ALARM_REPORT_PARAM;

参数名称	参数描述	取值
alarmName	告警名称	
alarmSource	App名称	
metaData	元数据	后续业务需要，暂时未使用

6.4.3.3 请求扩展头

无

6.4.3.4 响应码

如果功能正常，响应码200， 其他为错误码。

6.4.3.5 响应 Content

无

6.4.3.6 响应扩展头

无。

6.4.3.7 参考样例

无

6.4.4 告警参考样例

```
typedef struct {
char alarmName[64];
char alarmSource[32];
char metaData[0];
} ALARM_REPORT_PARAM;
typedef enum {
ALARM_URI_CONFIG_PARAM = 0,
ALARM_URI_INPUT_REGISTER = 1,
ALARM_URI_OUTPUT_REGISTER = 2,
ALARM_URI_EVENT_PUBLISH = 3,
ALARM_URI_SOURCE_PARAM = 4,
ALARM_URI_ACTION_PARAM = 5,
ALARM_URI_APP_PARAM = 6,
ALARM_URI_APPLANG_PARAM = 7,
ALARM_URI_ACTIONLANG_PARAM = 8,
ALARM_URI_RESTORE_DEFAULT_DATA = 9,
} ALARM_URI_RES;

AlarmDemo::AlarmDemo(void) : m_fd(-1)
{
LOG_DEBUG("Alarm Demo init");
}

AlarmDemo::~AlarmDemo(void)
{
if (m_fd != -1) {
close(m_fd);
m_fd = -1;
}
}
```



```
int32_t AlarmDemo::InitFd(void)
{
m_fd = open("/mnt/srvfs/alarm.paas.sdc", O_RDWR);
if (m_fd < 0) {
LOG_DEBUG("open alarm.paas.sdc srv fail\n");
return -1;
}
return 0;
}

/* 注册App */
int32_t AlarmDemo::CreateSource(void)
{
//当前仅支持绝对路径，请在App内部给出绝对路径
char file[32] = "/usr/app/cfg/alarmlist.ini"; // origin file
struct sdc_common_head_stru head;
head.version = SDC_VERSION;
head.url = ALARM_URI_INPUT_REGISTER;
head.method = HBTP_METHOD_CREATE;
head.content_length = sizeof(file);
head.head_length = sizeof(head);

struct iovec iov[2] = {
{.iov_base = &head, .iov_len = sizeof(head)},
{.iov_base = &file, .iov_len = sizeof(file)}
};

int32_t nret = writev(m_fd, iov, 2);
if (nret < 0) {
printf("alarm list writev failed[%d]\n", nret);
return -1;
}

nret = read(m_fd,&head, sizeof(head));
LOG_DEBUG("CreateAlarmSource nret:%d, code:%d\n", nret, head.code);
```



```
if (nret < 0 || head.code != SDC_CODE_200) {
printf("alarm list writev read[%d]\n", nret);
return -1;
}
return 0;
}

//
int32_t AlarmDemo::EventHandle(HBTP_METHOD_E method)
{
ALARM_REPORT_PARAM* name =
(ALARM_REPORT_PARAM*)malloc(sizeof(ALARM_REPORT_PARAM) + 32);
if (name == NULL) {
LOG_DEBUG("malloc failed");
return -1;
}
memset(name, 0 ,sizeof(ALARM_REPORT_PARAM) + 32) ;
memcpy(name->alarmName, "face", sizeof("face"));
memcpy(name->alarmSource, "Yolov3", sizeof("Yolov3"));
memcpy(name->metaData, "hahaha", 32); // tlv

struct sdc_common_head_stru head;
head.version = SDC_VERSION;
head.url = ALARM_URI_EVENT_PUBLISH;
head.method = method;
head.content_length = sizeof(ALARM_REPORT_PARAM) + 32;
head.head_length = sizeof(head);

struct iovec iov[2] = {
{.iov_base = &head, .iov_len = sizeof(head)},
{.iov_base = name, .iov_len = sizeof(ALARM_REPORT_PARAM) + 32}
};

int32_t nret = writev(m_fd, iov, 2);
if (nret < 0) {
free(name);
```



```
return -1;
}
nret = read(m_fd, &head, sizeof(head));
LOG_DEBUG("event , nret:%d, code:%d\n", nret, head.code);
if (nret < 0 || head.code != SDC_CODE_200) {
free(name);
return -1;
}
free(name);
return 0;
}

int32_t AlarmDemo::example(void)
{
if (InitFd() != 0) {
return -1;
}
if (CreateSource() != 0) {
return -1;
}
while (1) {
if (EventHandle(HBTP_METHOD_CREATE) != 0) {
return -1;
}
if (EventHandle(HBTP_METHOD_DELETE) != 0) {
return -1;
}
sleep(10);
}

return -1;
}
/*
参考文件配置文件：
```


alarmlist.ini

sourcelist=/usr/app/cfg/myalarm.ini

appname=Yolov3

language.ZH=/usr/app/cfg/alarmYoloZH.ini

alarmYoloZH.ini

Yolov3=智能告警

face=狗检测

myalarm.ini

name=face

desc=dogdet

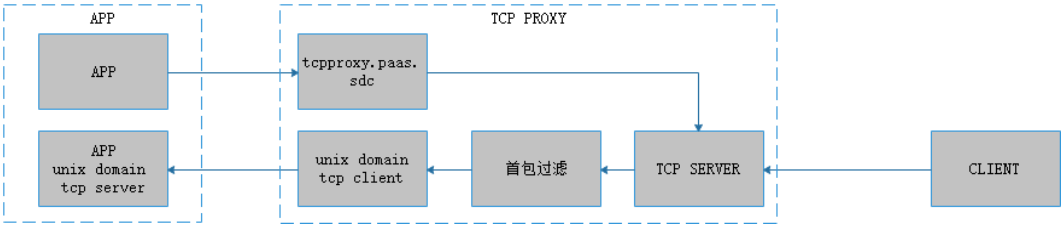
type=0

alarmTime=0

*/

6.5 tproxy.paas.sdc 服务化接口定义

6.5.1 注册 PF_UNIX 服务



APP可以注册一个PF_UNIX地址的SERVER和首包过滤条件，注册之后SDC会绑定3128端口，等待外部连接。在外部建连之后还会等待首包数据，如果首包数据匹配了APP注册的过滤条件，则TCP会建立到APP的PF_UNIX的连接，并转发消息包。

关闭发送注册命令的fd，相当于取消注册。

6.5.1.1 请求 Common Head 方法

METHOD	URI含义	URI取值
CREATE	SDC_URL_TPROXY_SERVER	0

6.5.1.2 请求 Content

/**

* domain: 当前仅支持AF_UNIX

```
* type: SOCK_STREAM | SOCK_DGRAM
* addr: unixdomain socket文件路径,当前仅支持创建在/tmp目录下
* filter：首包过滤条件的表达式。
*/
struct sdc_tproxy_server
{
int32_t domain;
int32_t type;
char addr[128];
char filter[0];
};
```

过滤条件详细说明：过滤条件是一个布尔值表达式，匹配从网络上收到的首包数据，匹配成功后，tproxy将数据转发给注册的AF-UNX地址。

过滤条件支持以下几种方式访问首包数据，支持的操作符参见事件订阅接口中的说明。

事件字段	含义	说明
data.u8[idx]	将整个首包作为uint8_t的数组进行访问，获取指定位置的值进行运算	idx只支持常量，比如是10进制或者16进制数据，16机制必须以0x开头。注意字节序，SDC都是小端序。
data.u16[idx]	将整个首包作为uint16_t的数组进行访问，获取指定位置的值进行运算	idx只支持常量，比如是10进制或者16进制数据，16机制必须以0x开头。注意字节序，SDC都是小端序。
data.u32[idx]	将整个首包作为uint32_t的数组进行访问，获取指定位置的值进行运算	idx只支持常量，比如是10进制或者16进制数据，16机制必须以0x开头。注意字节序，SDC都是小端序。
data.u64[idx]	将整个首包作为uint64_t的数组进行访问，获取指定位置的值进行运算	idx只支持常量，比如是10进制或者16进制数据，16机制必须以0x开头。注意字节序，SDC都是小端序。

6.5.1.3 请求扩展头

无

6.5.1.4 响应码

无

6.5.1.5 响应 Content

如果功能正常，响应码200， 其他为错误码。

6.5.1.6 响应扩展头

无

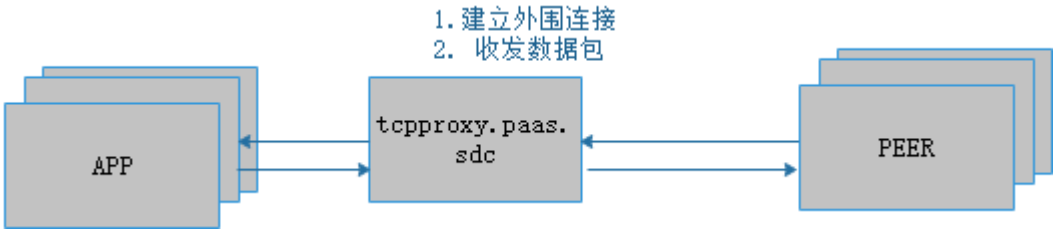
6.5.1.7 参考样例

```
/**
* type: SOCK_STREAM | SOCK_DGRAM
* RETURN: 返回到proxy服务的通信句柄，不要关闭，一直保留，进程退出会自动回收
* 返回之后有客户端连接到tproxy端口，根据首包匹配filter成功后会将消息转发给path对应的socket
* 用户应该先调用unix_server,成功后再调用tproxy_server_register
* 假设：首包数据的特征是前2个字节是0x5331,则filter可以是："data.u16[0] == 0x5331" 或者 "data.u8[0] == 0x31 && data.u8[1] == 0x53"（小端序）
*/
#include "sdc.h"
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/uio.h>
static int tproxy_server_register(int domain, int type, const char* path, const char* filter)
{
struct sdc_tproxy_server info = {
.domain = domain,
.type = type,
};
struct sdc_common_head hdr = {
.version = SDC_VERSION,
.url = SDC_URL_TPROXY_SERVER,
.method = SDC_METHOD_CREATE,
```



```
.head_length = sizeof(hdr),
.content_length = sizeof(info) + strlen(filter) +1,
};
struct iovec iov[] = {
{.iov_base = &hdr, .iov_len = sizeof(hdr) },
{.iov_base = &info, .iov_len = sizeof(info) },
{.iov_base = (char*)filter, .iov_len = strlen(filter) + 1 },
};
int fd = open("/mnt/srvfs/tproxy.paas.sdc", O_RDWR);
int nret;
snprintf(info.addr, sizeof(info.addr), "%s", path);
if( fd == -1) goto fail;
nret = writev(fd, iov, sizeof(iov) / sizeof(iov[0]));
if(nret == -1) goto fail;
nret = read(fd, &hdr, sizeof(hdr));
if(nret == -1 || hdr.code != SDC_CODE_200) goto fail;
return fd;
fail:
if(fd > -1) close(fd);
return -1;
}
```

6.5.2 建立网络连接



APP作为客户端，主动建立到外部网络的连接，收发数据。

这和HTTP PROXY的区别在于：tproxy提供的是服务文件通信链路和外部TCP/IP网络的转发；而http proxy提供的是unixdomain套接字和外部TCP/IP网络的转发。

6.5.2.1 请求 Common Head 方法

METHOD	URI含义	URI取值
CREATE	SDC_URL_TPROXY_CONNECTION	1

6.5.2.2 请求 Content

/**

* 提供TCP/UDP正向代理功能，创建成功之后，服务句柄将可以直接和外部实现网络通信，读写协议和SDC服务通信协议无关

* domain: 当前仅支持AF_INET

* type：SOCK_STREAM | SOCK_DGRAM

* addr: <ip>:<port>这种格式的字符串

*/

struct sdc_tproxy_connection

{

int domain;

int type;

char addr[128];

};

6.5.2.3 请求扩展头

无

6.5.2.4 响应码

如果功能正常，响应码200， 其他为错误码。

6.5.2.5 响应 Content

无

6.5.2.6 响应扩展头

无

6.5.2.7 参考样例

/**

* RETURN 可读写的fd，-1表示失败

* 返回的fd是服务文件的读写fd，不是socket的fd，不能支持send/sendto/recv/recvfrom，只支持write/writev/read/readv操作

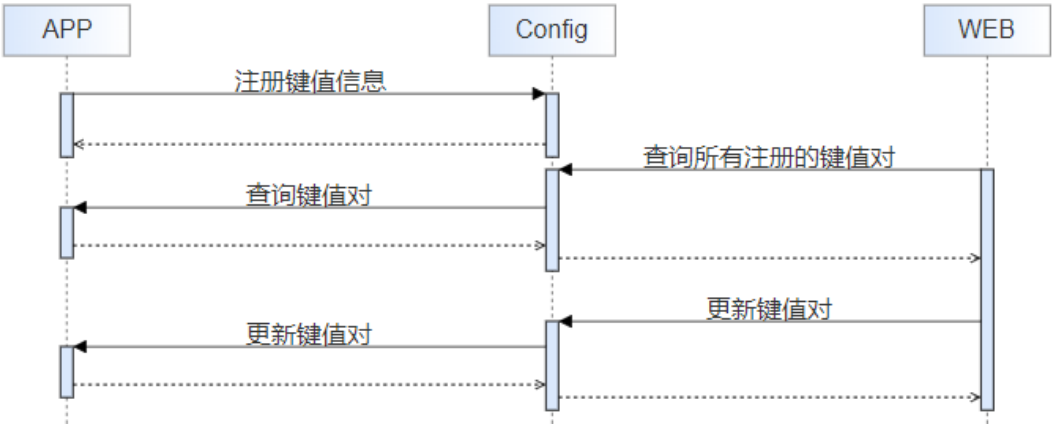

```
* 如果是UDP，接收的数据包大小不能超过4000字节。
*/
#include "sdc.h"
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/uio.h>
static int tproxy_connect(int type, const char* ip, unsigned short port)
{
struct sdc_tproxy_connection tproxy_addr = {
.domain = AF_INET,
.type = type,
};
struct sdc_common_head hdr = {
.version = SDC_VERSION,
.url = SDC_URL_TPROXY_CONNECTION,
.method = SDC_METHOD_CREATE,
.head_length = sizeof(hdr),
.content_length = sizeof(tproxy_addr),
};
struct iovec iov[] = {
{.iov_base = &hdr, .iov_len = sizeof(hdr) },
{.iov_base = &tproxy_addr, .iov_len = sizeof(tproxy_addr) },
};
int nret;
int fd = open("/mnt/srvfs/tproxy.paas.sdc", O_RDWR);
if( fd == -1) goto fail;
/** AF_INET的地址格式: <IP>:<PORT> */
(void)snprintf(tproxy_addr.addr, sizeof(tproxy_addr.addr), "%s:%d", ip, port);
nret = writev(fd, iov, sizeof(iov) / sizeof(iov[0]));
if(nret == -1) goto fail;
nret = read(fd, &hdr, sizeof(hdr));
if(nret == -1 || hdr.code != SDC_CODE_200) goto fail;
```



```
return fd;
fail:
if(fd > -1) close(fd);
return -1;
}
```

6.6 config.paas.sdc 服务化接口定义

通用配置服务提供为APP提供键值对的WEB配置页面，其工作流程如下所示：



其中查询键值对和更新键值对对于APP而言，是接收请求，回送响应，工作于服务端模式。

6.6.1 注册键值

一个APP的键值数量不确定，此外还应该支持多语言环境，所以APP注册的是本APP所有键值的描述信息。描述文件的内容包括：

app名称、app描述、键值、多语言显示文件。

app注册文件格式如下：

```
name=TXProcess
desc=Video_Vehicle_Detector_Platform
param=./TXscheme.json

[language]
language.ZH=./TXweblangZH.ini
language.EN=./TXweblangEN.ini
language.FR=./TXweblangFR.ini
language.ES=./TXweblangES.ini
```

name字段为app名称，desc字段为app描述，param为键值描述的schema文件，是选填字段，若param字段为空，则键值以参数获取接口返回的为准。language.*为多语言

显示文件，当前web支持四种语言的显示，包括中文(ZH)，英文(EN)，法语(FR)和西班牙语(ES)。

schema文件参考格式如下，具体格式以app实际键值为准：

```
{
"type": "object",
"properties": {
"Enable": {
"type": "boolean",
"default": "0"
},
"IPAddress": {
"type": "string",
"format": "ipv4"
},
"Port": {
"type": "integer",
"default": 0,
"maximum": 65535,
"minimum": 1
},
"LaneIdBase": {
"type": "integer",
"maximum": 99,
"minimum": 0
},
"IntersectionID": {
"type": "string",
"maxLength": 32,
"minLength": 0
},
"IntersectionDir": {
"type": "string",
"maxLength": 8,
```


"minLength": 0

}

},

"additionalProperties": false

}

语言文件参考格式如下，这里以中文为例：

group1=北向接入协议 // SDC 820新增字段，可以前向兼容老版本

TXProcess=视频车检器平台

Enable=使能

IPAddress=平台IP

Port=端口号

LaneIdBase=车道起始编号

IntersectionID=道路编号

IntersectionDir=道路方向

6.6.1.1 请求 Common Head 方法

METHOD	URI含义	URI取值
CREATE	SDC_URL_CONFIG_APP	1

6.6.1.2 请求 Content

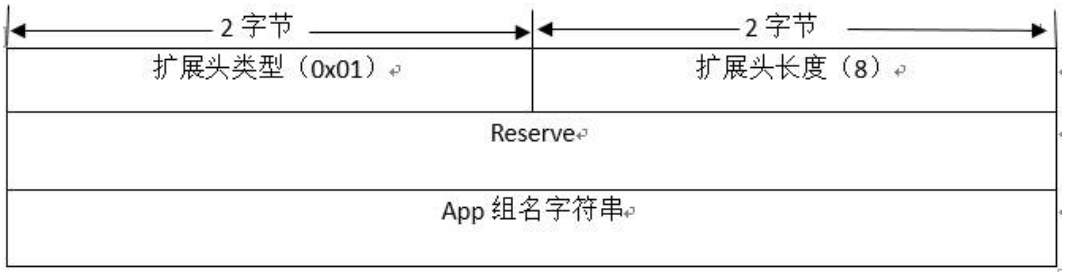
struct sdc_config_path {

const char* filename;

};

APP键值配置文件路径。

6.6.1.3 请求扩展头



注册app时需要app组名字符串(小于256字节)，缺省为“other”；对应每个app的所有语言资源包都要增加(如扩展头内容为“group1”)

对应.ini文件中需要增加APP组名描述，例如“group1=NorthboundAccessProtocol”字段，缺省则不需要。

6.6.1.4 响应码

如果注册成功，响应码200， 文件路径错误无法读取，响应码400，资源不足等为内部错误码500。

6.6.1.5 响应 Content

无。

6.6.1.6 响应扩展头

无。

6.6.1.7 参考样例

6.6.2 查询键值

注意：是APP接收请求消息，回送响应。

6.6.2.1 请求 Common Head 方法

METHOD	URI含义	URI取值
GET	SDC_URL_CONFIG_PARAM	0

6.6.2.2 请求 Content

无。

表示查询APP注册的所有键值对信息。

6.6.2.3 请求扩展头

无。

6.6.2.4 响应码

成功查询返回200，其他为APP端报告的错误码。

6.6.2.5 响应 Content

```
struct sdc_config_param {
char* key;
char* value;
};
```

```
struct sdc_config_param_list {
uint32_t cnt;
struct sdc_config_param* params;
};
```

6.6.2.6 响应扩展头

无。

6.6.2.7 参考样例

6.6.3 更新键值

注意：是APP接收请求消息，回送响应。

6.6.3.1 请求 Common Head 方法

METHOD	URI含义	URI取值
UPDATE	SDC_URL_CONFIG_PARAM	0

6.6.3.2 请求 Content

```
struct sdc_config_param {
char* key;
char* value;
};

struct sdc_config_param_list {
uint32_t cnt;
struct sdc_config_param* params;
};
```

6.6.3.3 请求扩展头

无。

6.6.3.4 响应码

更新成功返回200，其他为错误码。

6.6.3.5 响应 Content

无。

6.6.3.6 响应扩展头

无。

6.6.3.7 参考样例

7 附录

7.1 技术 FAQ

Q1. 在SDC中进行深度学习的AI算法开发与GPU上的开发又什么差别？GPU上开发的算法模型如何移植到SDC上？

A1. SDC中集成了AI专用芯片NNIE(NPU)，因为嵌入式芯片的特定约束，其功能不完全等同于GPU。NNIE仅支持Caffe的标准层和部分扩展层（具体内容参考《HiSVP 开发指南》，请直接联系合作经理获取），如果模型只包含NNIE支持的网络层次，完全可以在GPU上进行训练和调测，算法结果达到设计目的之后即可将Caffe模型文件通过RuyiStudio转换为NNIE支持的WK模型文件格式，在SDC上加载进行运算。如果使用其他深度学习的框架需要利用开源工具将其模型文件转换为Caffe格式后再转换为WK模型文件格式。

如果业务必须使用NNIE不能支持的网络层次，需要将这些网络层的运算改为CPU完成，因此需要将模型文件分拆为多个。CPU和NNIE之间的数据传递使用SDC提供的共享物理内存（MMZ）实现零拷贝的传输。

Q2. APP集成和算法集成的方案有什么差异?

A2. APP即应用，提供最终用户所需、直观可见的功能，任何APP和SDC的边界都由SDC所定义，即SDC提供的服务化接口，它属于SDC定义的可扩展机制。这些APP提供的可以是智能业务也可以是非智能业务。

而算法集成一定是由某个智能业务APP所定义，其具体接口由智能APP定义，属于智能业务APP内部可扩展机制的一种设计。比如智能交通APP有自己的多种算法集成接口，水污染检测APP又有另外一套算法集成接口。

简单总结：APP集成是SDC OS提供的能力，所有APP集成机制遵循标准的/统一的机制;算法集成是某个具体APP提供的一种能力，不同智能APP可能提供不同的算法集成机制。

Q3. APP和服务的区别是什么?

A3. 他们有相同点，都是独立的进程或者容器，可以独立发布/升级。他们的定位层次不同，APP提供自己能力给最终用户所用，而服务开放自己能力供APP所用。SDC OS中的最关键的服务是提供底层硬件能力的集中管理和调度功能，使得多个APP可以共享同一个硬件。此外APP也可以提供一个或多个SDC OS服务化接口，以服务的形式将自身的能力发布出去，供其他APP所用。

Q4. 如何防止APP及其算法模型文件不会被盗用?

A4. SDC提供服务化接口获取唯一硬件标识，APP基于此硬件标识生成License文件，APP启动或运行时验证当前环境的硬件标识和License文件的一致性，由此保证APP不会被盗用。算法模型文件可以由APP加密后安装到SDC上，运行时由APP解密后再调用SDC的模型文件的加载运算接口，由此保证模型文件的安全性。

Q5. 如何获取APP开发所需要的SDK软件包和编译、调测工具链?

A5. APP运行时不依赖SDC的任何SDK软件包，也不需要依赖海思的SDK。SDC的服务化接口是一种通过Linux文件传递的消息接口，仅需要基于本文档中的定义，自主生成APP所需要的消息数据结构的定义即可。SDC提供了C语言定义数据结构的头文件供APP参考/选用。

APP开发也不需要任何特定的编译器，使用开源的ARM编译器即可，推荐linaro。注意在3559A环境上需要ARM64的编译器，而在3519A和3516D上是ARM32位的编译器。SDC对APP开发编译环境以及编程语言都没有约束。

Q6. APP开发如何获取正确的海思SDK版本?

A6. 如A5的答复，APP不需要依赖海思SDK，也不能使用海思SDK。海思SDK提供底层硬件的访问能力，大部分接口需要特定权限的用户才能访问以及只能有一个访问者的约束。APP应该通过SDC的服务化接口和底层硬件交互，比如获取原始视频帧/加载运算深度学习的算法模型等。

在SDC上APP在独立的容器中运行，此运行环境中不会有任何海思SDK可用。依赖海思SDK的APP将不能在SDC上安装和运行。

Q7. SDC是基于哪个Linux发行版？APP能使用哪些Linux工具可用?

A7. SDC是一个嵌入式Linux运行环境，为了减少尺寸，未使用Ubuntu/centos这些多用在服务端的linux发行版，而使用busybox提供linux的命令和工具集，具体可以参考busybox官网的使用说明.

Q8. APP如何安装到SDC？如何升级和打补丁？是否保证升级时数据不丢失?

A8. SDC仅支持rpm软件包的安装，有关rpm软件包的规范参考本文档的“**APP软件发布、安装、运行环境规范**”章节和附录中“**APP打包及安装指南**”章节。所谓升级就是安装同名但不同版本的rpm软件包。

SDC支持升级时原版本数据不丢失，需要将这些数据放到自定义的数据盘中，典型的如License文件。也支持仅更新部分文件的rpm补丁软件包。

Q9. APP有自己的WEB Server和SDC OS的WEB SERVER都抢占TCP 80端口，如何解决?

A9. 所有APP都不能直接访问外部网络，只能绑定unixdomain socket地址，将其地址和app name注册到SDC OS提供的portal服务中，tcp 80端口由portal服务占用。portal服务提供类似手机APP桌面功能，运维人员打开portal的页面，点击APP图标后，运维人员的访问请求将通过portal服务转发到APP的WEB SERVER中。

portal服务终结https，提供统一的安全和帐号管理功能，各个APP的WEB SERVER可以至关注于自身的业务展现。

Q10. APP如何主动访问自己的云端设备?

A10. 如A9所答，APP不能直接访问外部网络，只能使用unixdomain socket地址。SDC提供unixdomain地址的http proxy。SDC在启动APP的时候会注入环境变量$http.proxy，APP基于此环境变量获取http proxy地址，建立到外部的通信通道。此地址的格式为：unix:<unixdomain socket地址>.

Q11. APP的云端设备如何主动访问APP?

A11. 同Portal机制，SDC提供API GATEWAY服务，APP将自己的unixdomain socket地址和app name注册道API GATEWAY，APP的云端设备即可通过API Gateway地址访问APP，其URL格式为https://<ip>:<port>/app name/...。

Q12. APP能使用多少计算和存储资源?

A12. APP的RPM软件包总大小不超过150M，其所需要的内存和数据盘空间大小可以在RPM包中的sdc.conf中定义，如果sdc.conf中申请的内存和磁盘大小超过了系统限制，安装或者运行将会失败。NNIE算力由所有APP共享，没有针对单个APP的约束。

Q13. APP是否必须通过SDC的算法商城才能安装到SDC上?

A13. 没有此限制。SDC的本地WEB也支持APP的安装和升级。我们鼓励APP发布到算法商场供更多潜在用户选用，也可以帮助APP开发者获取更大的实际收益。

Q14. APP发布到SDC算法商城之后，如何保证APP开发者权益不受侵犯?

A14. 如A4所答。APP开发者完全可以通过License控制，用户从算法商场购买APP之后还会向APP开发者购买License才可以运行，License的购买方法由APP开发者自主定义。

Q15. SDC服务化接口较多，对于一个智能业务APP需要哪些最基本的服务来开发智能业务?

A15. 对于一个智能业务，其处理过程一般包括三部分：获取原始视频帧，以视频帧作为输入加载运行算法模型文件，获取运算结果进行业务呈现。智能业务需要从video.iaas.sdc订阅视频帧，然后通过algorithm.iaas.sdc加载运行算法模型，最后的业务呈现可能有多种形式：在APP自身的WEB页面呈现，则需要将自己的WEB地址注册道portal.paas.sdc;或者上报到自己的云端服务，根据和云端通道建立方式的不同需要依赖gateway.paas.sdc或者利用http proxy;或者由SDC OS负责业务呈现，则需要智能业务将运算结果转换为SDC定义的元数据标准，发布给event.paas.sdc。

Q16. 最新版本的SDC OS需要从哪里获取？如何获取？

A16.最新版本的SDC OS发布路径会在如下路径公告：**https://bbs.huaweicloud.com/forum/thread-16617-1-1.html**，如需使用最新版本SDC OS，或联系合作经理获取对应区域销售经理联系方式，求助一线销售渠道负责人下载。

Q17.开发调测中有哪些求助渠道？

A17. 目前有3种求助渠道，1）登录**华为云工单系统**提单；2）登录**华为云开发者社区**搜寻或发帖交流；3）直接联系合作经理获取对应技术支持人员联系方式。

Q18.ISV开发SDC APP开发需要具备什么样的经验，预计多久可以完成开发。

A18.ISV开发人员需要具备嵌入式平台（Linux+busybox+makefile)的开发经验，开发语言不限，具备C/C++语言开发能力的开发者能快速的完成对接；开发人员还需要具有初步的深度学习网络模型设计、模型转换、模型裁剪经验；具备上述能力的开发者独立开发相应的APP，工作量约为3天~30天周期。

Q19.SDC 对算法模型有无依赖关系、 哪些训练平台所训练的算法可以直接使用？

A19.目前算法模型无法直接使用在摄像机上，使用caffe框架设计的算法模型，需要通过华为提供的Ruyistudio工具转换成摄像机识别的wk格式文件方可使用，Darknet、Pytorch、Tensorflow等深度学习框架设计的模型需要先使用开源工具转换成caffe格式。摄像机支持的算法模型和训练平台没有直接关系，只和模型文件格式有关系。

Q20.当前SDC OS支持哪些型号的摄像机？

A20.当前SDC OS支持的摄像机型号较多，主要有使用Hi3559A的芯片的X系类摄像机和部分使用Hi3519A的芯片M系类（M-Q）摄像机。

Q21.ISV开发SDC APP需要准备哪些环境？

Q21.首先安装ubuntu 64位操作系统，并针对摄像机所使用额芯片型号安装对应的编译工具链，具体参见**编译工具链**章节；安装海思的Ruyistudio模型转换工具、安装**RPM打包工具**。当前未提供Ruyistudio的下载链接，请直接联系合作经理李军提供。

Q22.模型加密处理需要ISV完成还是SDC OS上有对应的服务接口？加密的安全等级是否有相关的说明文件？

A22.为更好的保护ISV的知识产权，模型加解密由ISV独立完成，SDC OS不提供对模型进行加解密的服务，加密的安全等级由用户控制。

Q23.ISV APP上线算法商城审查与测试的方法、工具及流程？

A23.APP上线算法商城审查要求满足APP**软件包规范**，用户在上传算法商城时会对算法APP进行自检、测试，符合规范要求的APP才会被审查通过，不符合要求的APP将会显示发布失败，发布失败原因会呈现在审核意见栏。

7.2 APP 打包及安装指南

7.2.1 app 打包

app需要打包为rpm格式才能通过用户界面安装到SDC。rpm格式是redhat主导的linux世界的app主流打包格式，SDC直接采用该格式作为app的打包格式。

rpm(RPM Package Manager)设计初衷是源码/二进制打包、app安装/卸载/升级、app信息封装、包校验等。当前SDC主要支持rpm安装/卸载功能。rpm工具包包含的rpmbuild工具，可以将app打包为rpm格式。

7.2.1.1 安装 rpm 工具包

1）二进制方式：

rpm可以直接在各大主流Linux发行版本中安装。如Ubuntu可以通过如下命令安装：

sudo apt install rpm

Redhat系列发行版本及其衍生版本可以通过yum安装：

sudo yum install rpm

SuSE系列发行版本可以通过zypper安装：

sudo zypper install rpm

2）源码方式安装：

也可以通过源码安装最新稳定版本的rpm。可以到如下链接找到最新稳定版本及下载链接：

https://rpm.org/download.html

通过./configure && make && make install完成rpm的构建和安装。

7.2.1.2 使用 rpmbuild 构建 rpm 包

安装完rpm之后便可以使用rpmbuild。使用rpmbuild打包需要先编写SPEC文件，SPEC文件详细描述参考rpm.org的相关描述。

SPEC文件样例：

name: hello

version: v1.0.0

release: 1

summary: description in one line

license: -

vendor: hello

%define __strip %{cross}-strip

%prep

%build

%install

mkdir -p %{buildroot}


```
mkdir -p %{buildroot}/bin

mkdir -p %{buildroot}/lib

install -m 755 /path/to/app %{buildroot}/bin/main

install -m 666 /path/to/sdc.conf %{buildroot}/sdc.conf

install -m 755 /path/to/libxxx.so %{buildroot}/lib/libxxx.so

%clean

rm -rf %{buildroot}

%files

/*

/bin/*

/lib/*

%changelog

%description

description in multi-lines
```

上面的SPEC文件可以用于将构建后的二进制可执行程序打包为rpm。示例中/path/to/app是主程序，/path/to/libxxx.so是主程序依赖的动态链接库。SPEC文件指示rpmbuild在打包时将app打包到bin目录，并重命名为main；将libxxx.so打包到lib目录。

另外sdc.conf文件是可选的app自定义配置文件，主要用于定义app的资源配置如最大内存限制、最大磁盘限制等，也可以用于定义app自身的业务配置。

执行rpmbuild

以root账号为例，rpmbuild的默认构建路径是/root/rpmbuild。首先将上一章节的SPEC文件内容保存为/root/rpmbuild/SPEC/hello.spec，并在/root/rpmbuild目录执行rpmbuild命令，命令示例如下：

```
rpmbuild -bb --target=aarch64-himix100-linux --define "cross \"aarch64-himix100-

linux\"" SPECS/hello.spec
```

target是目标架构，cross是用于指定交叉工具链的前缀。根据CPU类型不同有不同的target和cross取值，对于3559A芯片，target/cross取值为aarch64-himix100-linux；对于3519A或3516D芯片，target取值为armv7l-himix200-linux，cross取值为arm-himix200-linux。

最终在目录/root/rpmbuild/RPMS/aarch64生成hello-v1.0.0-1.aarch64.rpm文件。该文件即为打包后的app安装包。

7.2.2 App 安装

当前SDC仅支持rpm解压安装等。

可以通过web页面实现app的安装，所有支持第三方应用的款型都可以通过维护->第三方应用标签页完成app的安装动作。web界面如下：

理论上3559A和3519A芯片款型都可以支持第三方app应用。后续会逐步增加支持其他芯片款型。

web页面新增应用按钮可以执行安装app，安装后的app会以表格的形式呈现。每个app都有名称、版本号、运行状态信息，和管理状态切换、删除按钮。可以根据需要执行启用app、停用app、删除app等操作。

注意事项：

网页方式安装app，后端CGI会检测文件类型，对于后缀为rpm的文件，如果网页所在系统安装了某些影音类软件，会导致系统自动认为rpm后缀的文件为音视频文件，导致上传操作失败。可以采用如下两种方式解决：

1）删除rpm后缀文件与音视频软件关联关系。

2）修改rpm后缀为非rpm。

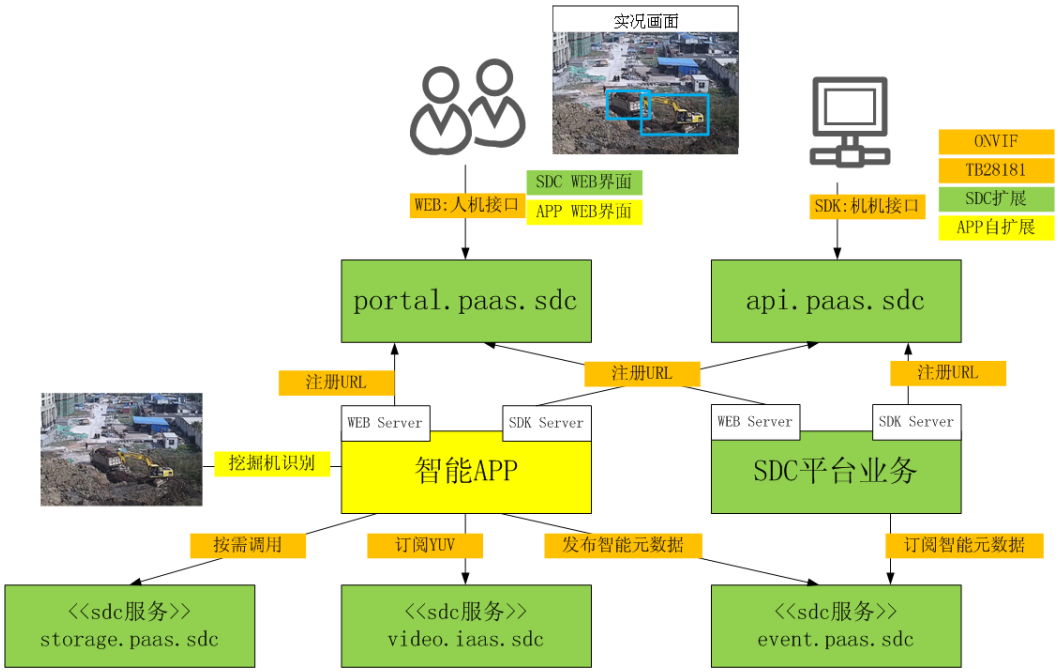
后续发布SiMS工具，用于对批量摄像机进行app生命周期管理操作，则不存在上述问题。

7.2.3 app 调测

app以前述章节方式安装后，将以容器形态运行，包括CPU、内存、磁盘等在内的资源都会与host/其他容器隔离，因此可能带来调测的不便。考虑到容器内运行与容器外运行最大的差异在于权限，所以功能调测/性能调测等可以直接在host上完成。容器形态只需要确认能够正确执行即可。

7.3 第三方 APP 算法的工作流程

SDC OS和摄像机硬件一起提供摄像机的基本功能，并开放摄像机的软、硬件资源供APP调用，第三方APP就是一个可以安装到SDC OS上的应用程序，作为一个独立的业务进程运行在SDC OS上，APP 可以通过SDC OS接口获取YUV数据流，通过SDC OS提供的服务化接口调用摄像机上的CPU/NNIE等硬件资源做智能识别及进一步处理，实现用户需要的功能。SDC OS支持多APP并行，通过华为SiMS管理工具在线管理APP程序，如安装、删除、更新、启用、停用APP，客户可以根据所需定制实现复杂场景下的多种智能应用，也可以通过更换智能算法应用变更用途。以下是第三方APP在摄像机中工作的流程图。



1、智能算法APP启动后，按需注册自身北向的WEB URL和北向SDK URL，运维人员可以通过APP自身的WEB界面设置智能算法的APP的调优参数。

2、智能算法APP向video.iaas.sdc服务订阅YUV数据，作为算法输入；算法输出按需发布到event.pass.sdc。

3、SDC平台提供的实时视频流等服务的输出界面中可以呈现智能算法APP发布的智能元数据；第三方平台也可以通过SDC北向接口获取。

备注：SDC平台的PORTAL和API服务提供北向的统一入口（端口号）、统一安全机制、HTTPS终结和URL路由功能，基于注册信息分发URL到各个APP。

7.4 算法开发步骤

步骤1 算法模型适配。

1. 已有训练好算法模型的算法厂家，可以使用华为的Ruyistudio工具把已经训练好的caffe框架的AI算法模型编译成海思NNIE支持的WK文件格式，具体转换方法参见如下链接：https://bbs.huaweicloud.com/forum/thread-16777-1-1.html。

   说明

   由于当前支持SDC OS的摄像机使用了海思Hi3559A/Hi3519A芯片智能异构芯片，其使用的AI加速引擎芯片NNIE采用Caffe框架，对于非caffe深度学习框架（如Pytorch、tensorflow）设计的AI模型需要采用开源工具转换成Caffe格式。

2. 无训练资源或没有算法经验的厂家，可以使用华为企业云的ModelArts服务进行AI开发，ModelArts是面向AI开发者的一站式开发平台，提供海量数据预处理及半自动化标注、大规模分布式训练、自动化模型生成等服务，帮助用户快速创建AI模型，ModelArts链接如下：**https://www.huaweicloud.com/product/modelarts.html**。算法模型训练好之后，需要转换成适合华为SDC的Caffe模型，参考**上一步**转换成NNIE支持的WK文件。

步骤2 开发者参照SDC服务化接口开发摄像机应用，并进行调试。用户可以在自己开发的APP中，使用SDC OS提供的服务化接口加载自己的AI算法模型（NNIE 模型创建，具体参

见**algorithm.iaas.sdc服务化接口定义**章节），根据需要预定需要的YUV数据流送入AI网络，调用文档中NNIE forward和NNIE forwardWithBbox服务的NNIE推理功能接口对各网络节点进行智能推理处理。用户可根据本文档进行代码开发，具体编码可以参考文档中的示例代码，也可以参考开发者论坛中的帖子：**https://bbs.huaweicloud.com/forum/thread-17561-1-1.html**。

步骤3 用户开发完APP后，使用**4.1 编译工具链**中的编译器编译并打包成RPM文件安装包，用户安装RPM安装包到对应的安装了SDC OS软件包的华为摄像机，在摄像机上进行调试验证（用户在初期调试阶段可直接编译成elf文件，在摄像机上直接进行调试）。详细步骤参见**APP打包及安装指南**，或者参考论坛中的附件：**https://bbs.huaweicloud.com/forum/thread-17512-1-1.html**。

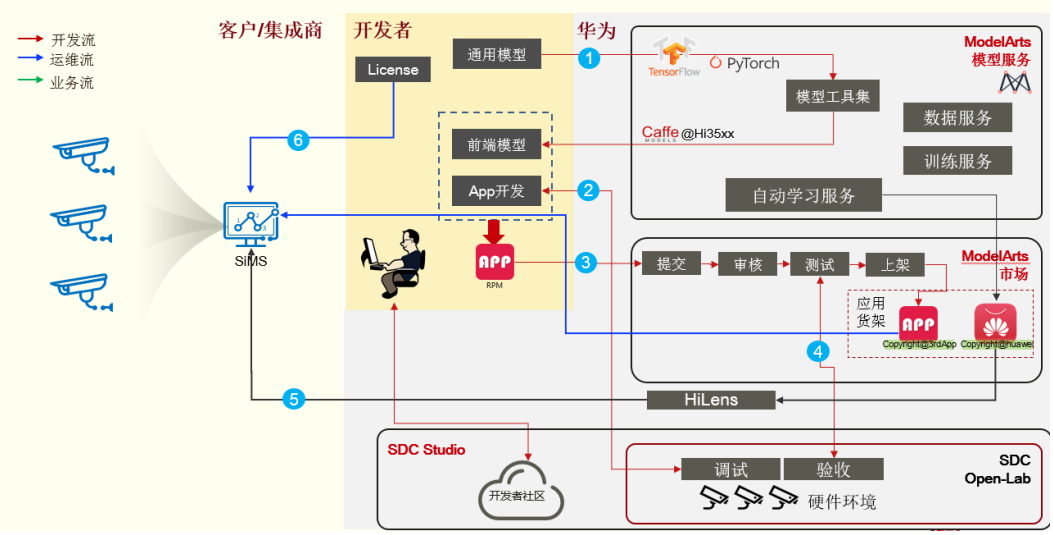
说明

用户可以通过购买、借用的方式获得摄像机，对算法程序进行调试，也可以申请使用华为的OpenLab实验室远程登录到对应的华为摄像机环境进行调试，网址参见**7.8 开发调试云网址**，使用需要找合作经理开放账号。

步骤4 第三方AI公司可以将算法发布到华为AI市场，ModelArts市场对软件包进行审核、验收并上架。

步骤5 客户通过iClient连接到市场下载App包并安装到SDC中。客户向开发者获取License并通过iClient安装到SDC。

----结束



7.5 SDC OS 生态组件介绍

围绕SDC OS，华为提供算法商城、开发云、训练云、SiMS(SDC Controller)、开发者社区论坛、工具链支撑生态发展。

7.6 当前支持 SDC OS 的摄像机款型

华为规划X2221-CL、X2281-HL、M2221-QL、M2241-QL、M1221-Q、M2391-T等多款摄像机供用户选择，后续规划了更多款型，不支持SDC OS的款型，暂不能升级SDC OS版本，如有特殊要求，需要提需求。要了解摄像机详细信息或更多其他款型，需要到华为产品官网了解，也可以联系合作经理李军（057113588038981）推荐款型，更多款型信息**参见官网SDC产品介绍**。

7.7 SDC OS 开发者论坛

摄像机开发者论坛：https://bbs.huaweicloud.com/forum/forum-799-1.html

有任何疑问可以去论坛发帖互动交流，后续有工单系统**参见官网宣传**上线。

7.8 开发调试云网址

服务器网址**http://222.90.69.7:65336**

目前可以远程使用的数量有限，请登录网址注册账号后，联系合作经理李军申请激活账号。