注意

您购买　　的产品、服务或特性等应受盛科网络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使用范围之内。除非合同另有约定，本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。

盛科网络(苏州) 有限公司

地址　　**江苏省苏州市工业园区恳汉街5号(腾飞新苏工业坊)B 幢4楼13/16单元**

电话　　86-512-

传真　　**86-512-**

网址　　**http://ww:**

邮箱

表格

图片

1 介绍

本手册假设您在本手册涉及的业务领域具有较好的工作经验和知识，并假设您对盛科芯片和基础知识有整体的了解，如果您没有使用过本系统，我建议您接受一次(或者更多)的有关芯片知识的培训。

在以太网交换芯片应用领域，SDK 的重要性十分显著,它是上层系统与芯片连接的桥梁，上层系统必须通过它才能配置芯片。通过本手册，你可以了解盛科 SDK 的接口设计原则、架构设计目标、系统架构、组织结构以及开发指导等方面阐述 Centec **SDK** 通用性的设计理念，旨在解决大家关心的两个个问题：

- **Centec SDK**可移植性，用户如何能能快速学习和使用 API 来开发系统?
- Centec SDK API接口如何做到前向兼容和后向兼容?

2 软件架构

2. 1接口设计原则

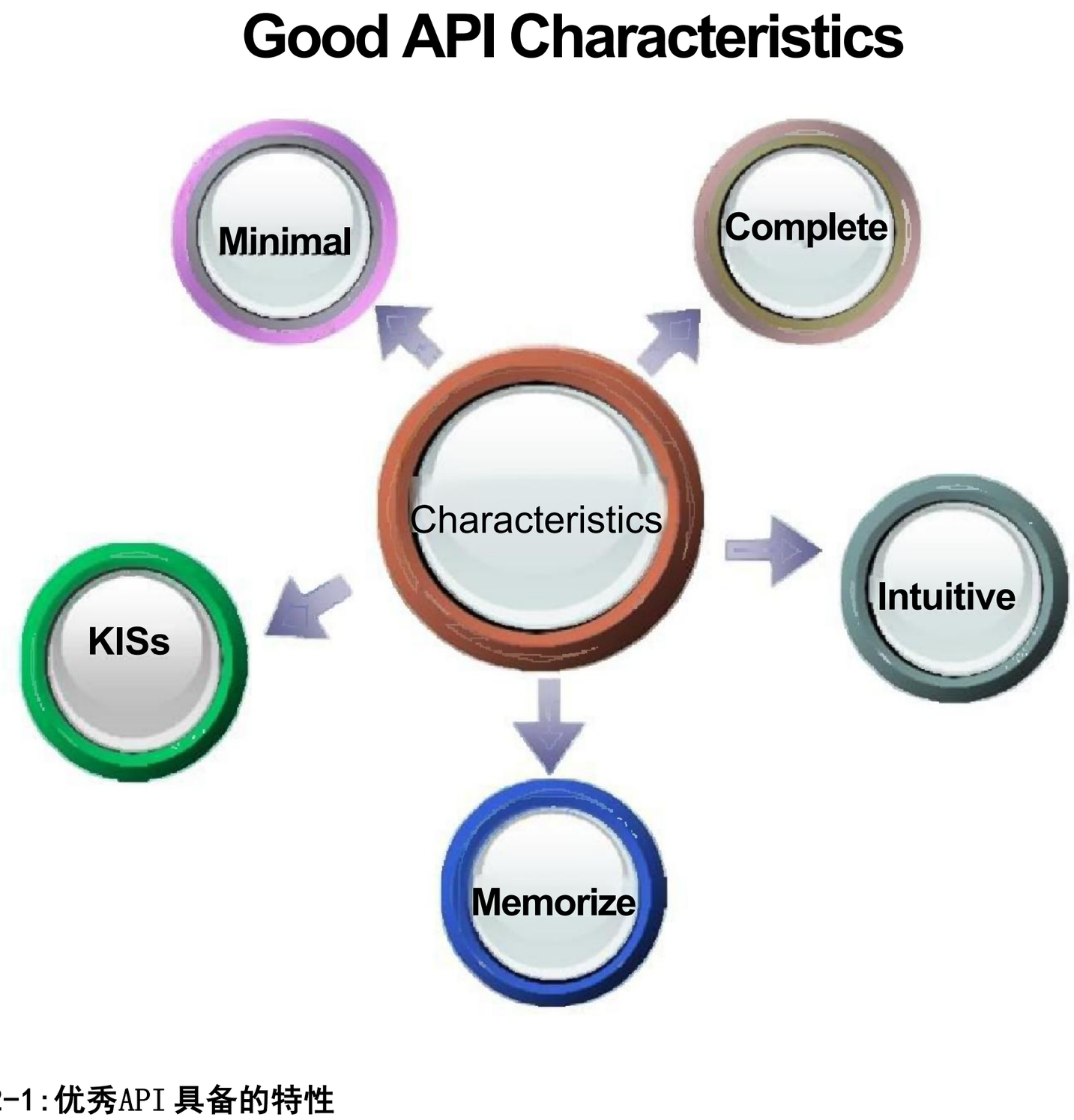

图2-1:优秀API 具备的特性

盛科**SDK**在设计上始终遵守 API 应该是最小化且完整的，拥有清晰且简单的语义，直观化，容易记忆，并且引导人写出易读的代码。

最小化(Minimal)

最小化的原则体现在使用较少的 API 接口和较少的数据结构满足功能需求，同时在定义新的接口时尽量考虑重用原有的 API 接口和数据结构。这个原则可以让 API 更简单易懂，更好记，更容易除错，且更容易改变，同时最小化的接口和数据结构也可以使接口扩展性更好。一个比较典型的例子：FDB 中 flush FDB或Query FDB, 现有的 SDK 仅仅提供一个 API函数就实现所有与之相关的功能，上层用户可以根据删除或查询条件实现相关的功能，如果现有的接口不能满足功能，只需要在提供的数据结构中增加删除或查询条件就可以扩展相应的功能，如下是 CTC SDK所 有 IPUC、IPMC、L2UC、L2MC 的所有 API 接口：

```
/*ipmc APIs*/
extern int32 ctc_ipmc_add_default_antry(uinte ip_version,cte_ipme_default_action_t type);
extern int32 ctc_ipmc_add_group(ctc_ipmc_group_info_t*P_group);
extern int32 ctc_ipmc_add_member(ctc_ipme_group_info_t*P_group);
extern int32 ctc_ipmc_get_mcast_force_route(ctc_ipme_force_route_t*p_data);
extern int32 ctc_ipme_get_statsTctc_ipmc_group_Info_E+p_giOup,ctc_stats_basic_t*p_stats);
extern int32 ctc_ipmc_init(void*ipmc_global_cFg);
extern int32 ctc_ipmc_remove_group(cte_ipmc_group_info_t*P_group);
extern int32 ctc_ipmc_remove_member(ctc_ipmc_gzoup_info_t*P_group);
extern int32 ctc_ipmc_reset_atata(ctc_ipmc_group_info_t*P_group);
extern int32 ctc_ipmc_set_mcast_force_route(ctc_ipmc_force_route_t*p_data);
extern int32 cte_ipme_update_rpf(cte_ipmc_gzoup_info_t*P_group);
rn
ex: int32 ctc_ipuc_add(ctc_ipuc_param_t*P_ipuc_info);
/* ip uc APIs */
extern int32 ctc_ipuc_add_default_entry(uintBip_ver,uint32 nh_id):
extern int32 ctc_ipuc_epu_rpf_check(bool enable);
extern int32 ctc_ipuc_init(void*ipuc_global_cfg);
extern int32 ctc_ipuc_ipv6_enable(bool enable);
extern int32 ctc_ipuc_remove(cte_ipuc_param_t*P_ipuc_info);
eatern int32 ctc_ipuc_set_lookup_ct1(cte_ipuc_1ookup_eti_t*p_lookup_ct1_info);
extern int32 ctc_ipuc_set_route_ct1(ctc_ipuc_route_ct1_t*p_route_ctI_info);
/*FDB APIs*/
extern int32 ctc_12_add_default_entry(ctc_12dflt_addz_t*12dflt_addr);
extern int32 ctc_12_add_fdb(cte_12_addr_t*12_addr);
extern int32 ctc_12_add_fdb_with_nexthop(ctc_12_addr_t*12_addr,uint32 nhp_id);
extern int32 ctc_12_add_port_to_default_entry(ctc_12dflt_addr_t*12dflt_addr);
extern int32 ctc_12_fdb_flush(ctc_12_fdb_flush_t*pFlush);
extern int32 ctc_12_fdb_init(void*12_fdb_glcba1_cfg);
extern int32 ctc_12_get_default_entry_features(ctc_12dflt_addr_t*12dflt_addr);
extern int32 ctc_12_get_fdb_by_index(uint32 index,ctc_12_addr_t*12_addr);
extern int32 ctc_12_get_fdb_count(ctc_12_fdb_query_t*pQuery);
extern int32 ctc_12_get_fdb_entry(cte_12_fdb_query_t*pQuery,ctc_12_fdb_query_rst_t*query_rst);
/*L2MC APIs*/
extern int32 ctc_12mcaat_add_addr(ctc_12_mcaat_addr_t*12mc_addr);
extern int32 ctc_12mcast_add_member(ctc_12_mcaat_addr_t*12mc_addr);
extern int32 ctc_12mcast_remove_addr(ctc_12_mcast_addr_t+12me_addr);
extern int32 ctc_12mcast_remove_member(ctc_i2_mcaat_addr_t*12mc_addr);
```

完整性(Complete)

完整性是指一个 API 要提供所有期望的功能，而不能一个功能需要两个或两个以上的 API 接口来提供，一个比较典型的例子，在Vlan class,CTC API 提供一个 API 接口提供基于 IP/MAC/Protocol VLAN,同时为了更好的扩展性 ctc vlan class t数 据

结构提供了具有流的Vlan Classify行为。

```
extern int32 ctc_vlan_clasification_entry(cte_rlan_cla8_tp_rlan_clas);
ctc_vlan_remove_vlan_classification_entry(ctc_vlan_class_t*p_vlan_class);
extern int32
ctc_vlan_flush_vlan_classification_entry(ctc_vlan_class_type_t type);
extern int32
ctc_vlan_add_vlan_classification_default_entry(ctc_vlan_cla-typet typeaction);
extern int32
ctc_vlan_remove_vlan_classification_default_entry(ctc_vlan_class_type_t type);
```

KISS原则(keep it simple and stupid)

代码实现尽量简单，白痴。简单白痴的代码可读性强，易维护，并降低复杂度。代码一经写成，就会读多次。易读的代码可能会花点时间来写，但是可以节省产品周期中的其他时间。

直观化(Intuitive)

API接口和参数必须是直观化的，直观化体现为：使用者不需要知道 **API**的内部实现就能读懂并应用这个 **API;** 使用者不借助文档就知道如何调用 **SDK**接口；不需要理解芯片的具体实现就知道如何调用 **SDK** 接口。例：对上层用户而言，路由增加和删除只需要预先创建下一跳 (Nexthop)，然后根据 IPDa+(VrfId)，创建路由条目即可。

```
/*@brief Define ipuc parameter structure */
struct cta_ipuc_param_s
{
    uint32 nh_id;            /**<Nexthop ID*/
    uint16 vrf_id;             *<Vrf ID of the route*/
    uint1613_inf;            /**<13 interface,only valid when CTC_IPUC_FLAG_CONNECT is set /
    uint8 route_flag;          *<Flags of the route, values is defined by ctc_ipuc_flag_e,
                                one route can set one or more flags,or no flag *7
    uint0 masklen;           /**<Mask length of destination */
    uintB . ip_ver;                  * * < Destination ipaddress version,CTC_IP_VER_4 or
    uint8 is_ecmp_nh:                /** <Nexthop is a ECMP group */
    union
    {
        ip_addr_t ipv4;              /**<IPv4 destination address */
        ipv6_addr_t ipv6;            /**<IPv 6 destination address */
    }ip;
}
ty pedef struct ctc_ipuc_param_a ctc_ipuc_param_t;
— aww ×
extern int32 ctc-ctc-ip1uc-rdd(vtcitpu-paramrtpipue in_info);
extern int32 ctc_ipuc_add_defauit_entry(uint8 ip_ver,uint32 nh_id);
```

易于记忆(Easy to learn and memorize)

让 API易于记忆，使用统一且精确的命名方法。API 的接口和参数命名上，同一概念采用同样的名字，不同的概念采用不同的名字；同时API接口命名简洁、风格统一，让上层用户使用时易于记忆，同时一致的**API**能够联想推测。如下 nexthop提供 的 API 接口，用户在使用了创建**nexthop** 接口后，再创建 mpls 的 nexthop, 很 自 然就能联想到 MPLS的 nextho接口。

```
extern int32
ctc_nh_create_l2uc  (uint16 gport,  ctc_nh_param_brguc_sub_type_t nh type);
extern     int32
ctc_nh_destory_l2uc(uint16 gport);

extern int32
ctc_nh_create_ipuc(uint32 nhid, ctc_ip_nh_param_t*p nh param);
extern         int32
ctc_nh_destory_ipuc(uint32 nhid);
extern int32
ctc_nh_update_ipuc  (uint32 nhid, ctc_ip_nh_param_t*p_nh_param);

extern int32
ctc_nh_create_mpls(       uint32 _nhid, ctc_mpls_nexthop_param_t*  p__nh__param) ;
extern         int32
cte_nh destory_mpls(uins32-hid);
ctc_nh_update_mpls      (uint32 nhid , ctc_mpls_nexthop_param_t*p__nh__param);

extern int32
ctc_nh_create_ecmp(ctc_nh_ecmp_creat_data_t* pdata);
extern    int32
ctc_nh_destory_ecmp(uint32 nhid)

exter n     int3 2
ctc      _nh _u pdate_ecmp(ctc_nh_ecmp_update_data_t*pdata);
```

2. 2架构目标

通用性，从调用者的角度来进行设计，做到硬件透明

SDK的设计对硬件是透明，API 接口与具体的芯片无关，从系统应用的角度去设计API的接口，使接口更具通用性，上层用户不需要关心具体的芯片处理流程而从上层的业务应用找到对应的 API 函数，也不需要用户了解如何操作芯片中具体的表和寄存器的字段而达到用户能配置芯片。

易用性，使SDK　具有最小系统的功能

体现在：**SDK** 默认提供集成对所有的中断进行处理，包括 **mac learning、Mac Aging、**Oam、stats等；SDK 默认提供集成对 CPU 的收发报文进行处理；同时为了使用户能灵活处理这些应用，SDK 还提供回调函数给用户注册，可由用户控制处理行为。

后向兼容，保持接口的一致性

接口的一致性使用户同一代芯片不同版本或向新一代芯片**SDK** 切换时实现客户的代码平滑升级，能使客户在短时间内完成版本的升级，可以节省客户大量开发时间；同时一致性能大大降低用户的学习和使用成本，用户过去的努力学习，能持续的收效. **Greatbelt SDK**复用90%以上的 **Humber SDK**接口 .

前向兼容，使接口具有可扩展性

在接口设计时屏蔽一些芯片实现的具体细节，对芯片实现细节进行抽象化，不会因为不同代芯片间实现的不同而更改接口

模块化，根据不用的功能应用划分模块，用户可针对不用的应用场景选择不用的模块

SDK 在设计时根据业务应用划分不同的功能模块，在设计时遵循高内聚低耦合的设计思想，使各模块之间功能保持相对独立，并能实现模块的可卸载，用户可根据不用的应用场景选择不用的功能模块。

多平台兼容，使代码具有可移植性

SDK提供封装与操作系统无关的函数的模块 (KAL 模块), 使SDK 能在linux、Vworks等系统中运行，同时SDK的代码完全使用标准 ANSI‘C’, 提高客户的可移植性。

提供丰富的debug　**手 段**

SDK专门提供一个 Diag模块，该模块提供丰富 Debug 功能，用户可通过 SDK提供的友好的 Diag CLI界面进行操作，可有效的帮助用户诊断、定位和排除芯片转发出现的问题。

容易查找和学习

API通过一个统一的文档介绍 SDK, 并根据不同的业务应用提供尽可能多的示例和 copy-paste 的代码，指导用户开发，降低用户的使用门槛。

参考centec_sdk_guide.chm。

SDK CLI 界面友好，简单易用

SDK CLIs界面友好，简单易用，在设计时遵循一个 API 对应一个 CLI, 通过 CLI 就能完全反映出 API的特性，使用户能通过SDK提供的CLIs 就能了解如何使用 API 接口。

健壮性与可靠性

- **提供内存管理函数**

SDK不会直接调用系统的 mallodree函进行，而是用 mem_malloc/mem_free替换，mem_malloc/mem_free的实现时采用宏，所以 SDK 的使用者在把 SDK 代码集成系统中时，可把具体实现替换成系统中的内存管理模块。

```
#ifdef MEMMGR
   fe   mem-freletype,size)mem_mgr_malloe(size,type,_FLE_,_INE_)
   {\
     mem-mar free(ptr,_FHLE_,_LINE_);\
   }
#defin e mem_realloc(type,ptr,size) mem_mgr_realloc(type,ptr,size  ,__FILE_,_LINE_)
# else
#define mem_malloc(type,size)  malloc(size)
#define mem_free(ptr)     —
    {\
      free (ptr);\
       ptr  =  NULL;\
     }
#de  fine mem_realloc(type,ptr,size) realloc(ptr,size )
#endif
```

- **提供 Debug 管理函数**

为了控制各个模块/子模块的调试信息的输出，调试信息的输出使用宏

CTC_DEBUG_OUT,由各个模块/子模块调用，输出调试信息. 当**SDK**的使用者在把 SDK 代码集成系统中时，可把具体实现替换成用户系统中的 **Debug**模块。

```
#define CTC_DEBUG_OUT(mod, sub, typeenum,level, fmt, args...) \
{                                                              \
   if (ctc_debug_check_flag(ctc_##mod##_##sub##_debug_handle, typeenum,level)) \
   {                                                           \
       kal_printf(fmt, ##args); \
   }                                                           \
}
```

- **丰富的 Error Code机制**

对不同功能模块每一个细节的操作失败都会返回一个错误码，并且提供对错误码进行详细的解释，上层用户能借助错误码很好的定位配置出现的问题.

- **错误时对先前操作进行回滚删除**

SDK 中的功能模块在操作失败时会对先前的操作过程进行回滚删除操作。

2.3在系统中的位置

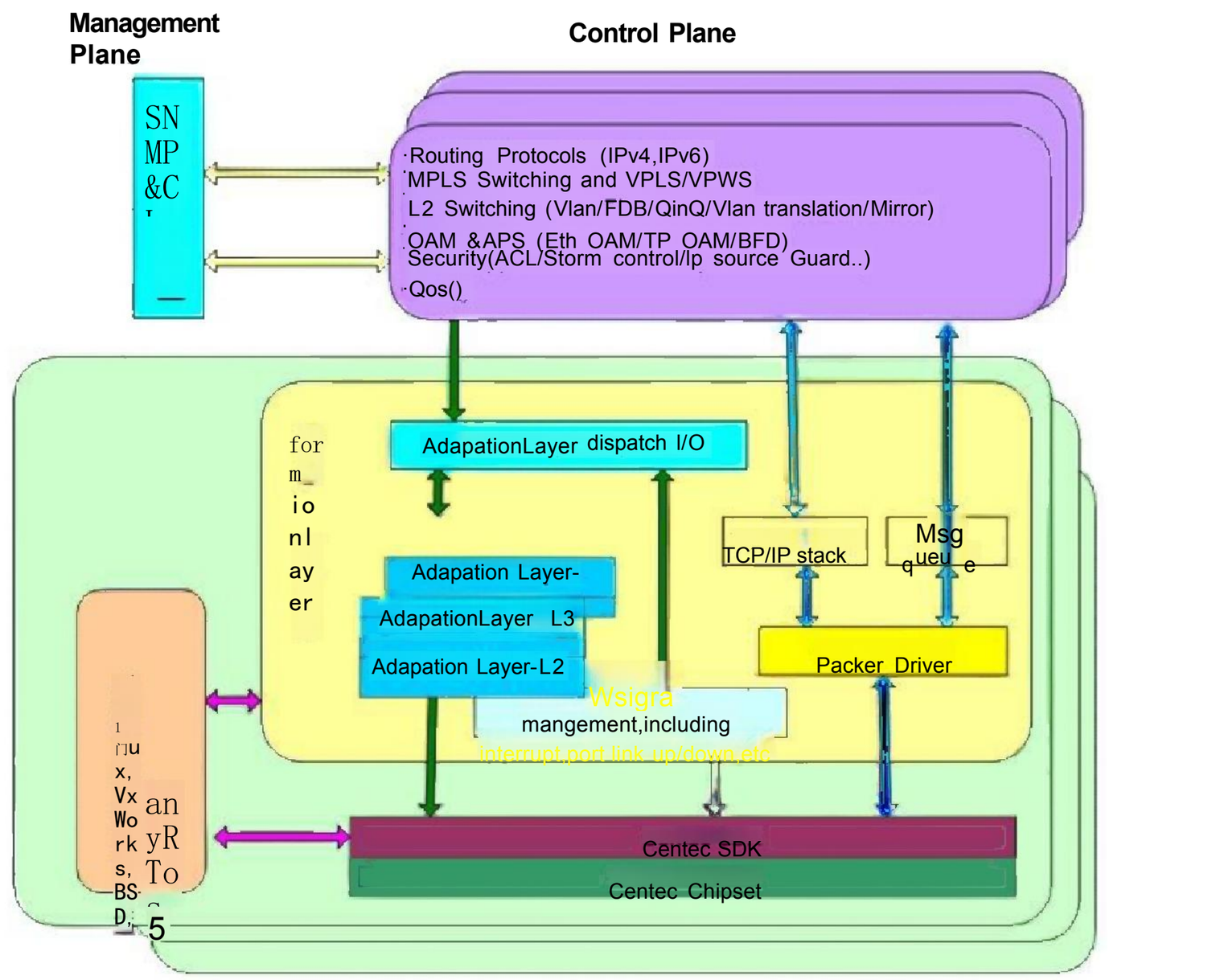

图2-2:Centec SDK 在系统中的位置

从上图可以看出，系统对心片的配置芯片、报文收发和芯片中断的处理都是通过适配层转化后进行处理的，故 Centec SDK再 做 API 设计时：

1. 从系统应用和适配层角度方面进行深入研究和分析：遵循**SDK** 架构目标中的通用性设计原则，对上层业务应用模块的接口进行抽象，封装并提供与芯片无关的接口。从系统应用的角度划分 SDK 的模块，并保持模块间相对独立，用户可根据不用的应用场景选择不用的功能模块。目前 **SDK**的功能模块划分为：12 (fdb/l2mc)、ipuc、ipmc、mpls、oam、aps、ptp、port、vlan、linkagg、13if、cpu traffic、acl、qos、queue、pdu、parser 、mirror、security、stp、sync_ether、stacking 等模块。

2. 基于盒式与分布式架构兼容性的设计使 API 接口完全屏蔽不同的应用场景，使 SDK在应用于盒式、堆叠及分布式系统的场景时接口一致，而只需要在 **SDK** 初始化时指定本芯片在全局中使用的 gchip, 之后对心片配置port 和 chip 都包含 gchip 的信息就能完成配置. int32 ctc_set_gchip_id(uint8 lchip,uint8 gchip)。

3. Centec SDK基于前向兼容和后向兼容架构设计目标，使提供的 API 接口在不同版本或不同代芯片间保持一致性。对于系统而言，不同的产品序列可能使用不同版本的 SDK 或不同代的芯片，系统软件的上层协议栈不需要更改或做较小的改动，如果 SDK 的 API 接口改动较少，则能降低适配层代码开发的工作量，使客户在短时间内完成版本的升级，节省客户大量开发时间；同时一致性也大大降低用户的学习和使用成本。

2.4软件架构

Centec SDK 主要由 SDK Core/SDK Applications/OS Services三部份组成，如下图所示：

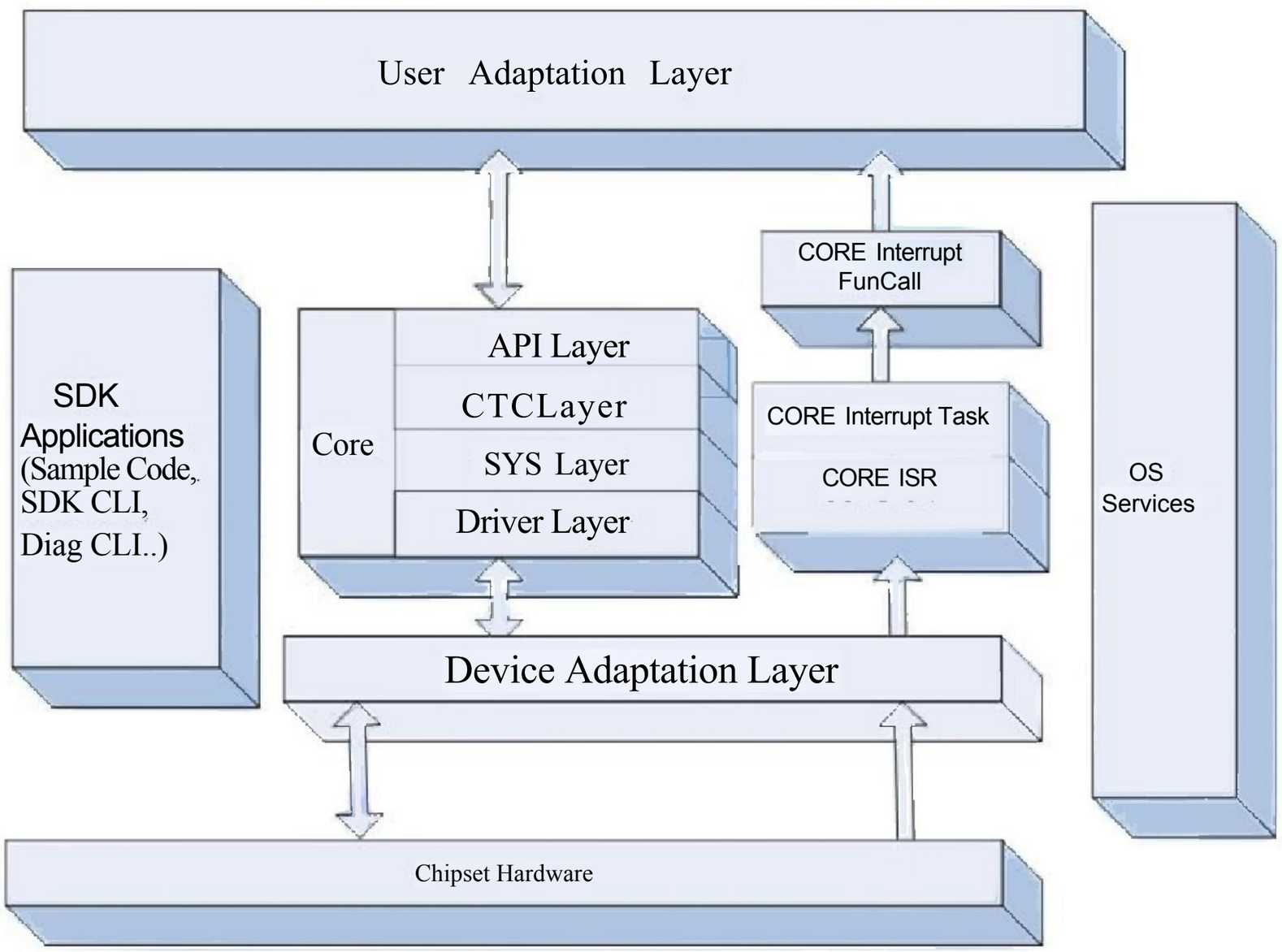

图2-3：Centec SDK在系统中的位置

2.4.2 SDK Core

SDK core是 SDK 的核心逻辑的处理代码，主要完成：

- 对上层提供友好的接口，让产品的开发周期更短；
- 对芯片提供的功能进行抽象；
- 封装与芯片无关的功能接口；
- 对芯片中表项索引的管理和维护；
- 对芯片中的表项及寄存器进行管理和维护；

2.4.3 SDK Applications

SDK Applications是 SDK 为了使用户更好的理解和使用 SDK, 为 SDK 的使用者提供以下应用：

- 配置 SDK 的 CLI 界面；
- 调试和诊断芯片的CLI 界面；

- 应用方案的示例代码，如 MAC Learning/Aging 处理的软件代码、保护切换、FTM Profile的使用、CPU 收发报文等。

2.4.4 OS Services

OS Service的主要功能是封装与操作系统无关的函数，包括 C 库函数、内存管理模块、Debug 管理模块等。

2.5 SDK组件组成

如下图所示，SDK 共有以下组件组成： sal、Core、Driver、app、libctccli、ctccli等，各组件的功能作用分别是：

- sal:主要功能是封装与操作系统无关的函数，包括C 库函数和内存管理函数等；
- dal:主要功能是封装芯片驱动相关的 API,包 括 PCIe、chip I/O、Serdes、Interupt、datapath 等；
- Core: 实现SDK 的核心逻辑的代码，芯片 function的 APIs;
- driver ：提供I/O接口操作芯片中表和寄存器；
- libctccli:主要功能为提供 Centec SDK CLIs实现的公共的 CLIs库；
- ctccli:主要功能为向 SDK 使用者提供CLIs命令及芯片 debug使用的命令；
- app: 初始化 SDK 的示例代码，用户可根据实际情况进行修改或替换。
- dkits:Dkits(Debug&Diagnosis)工具集是 SDK 的一个可选组件，Dkits提供了各种命令简洁、界面友好的CLI, 用于调试盛科系列芯片。通过 Dkits提供的一系列命令，用户可以快速的定位芯片问题(如丢包、转发错误等),大大提高了对芯片的开发速度。

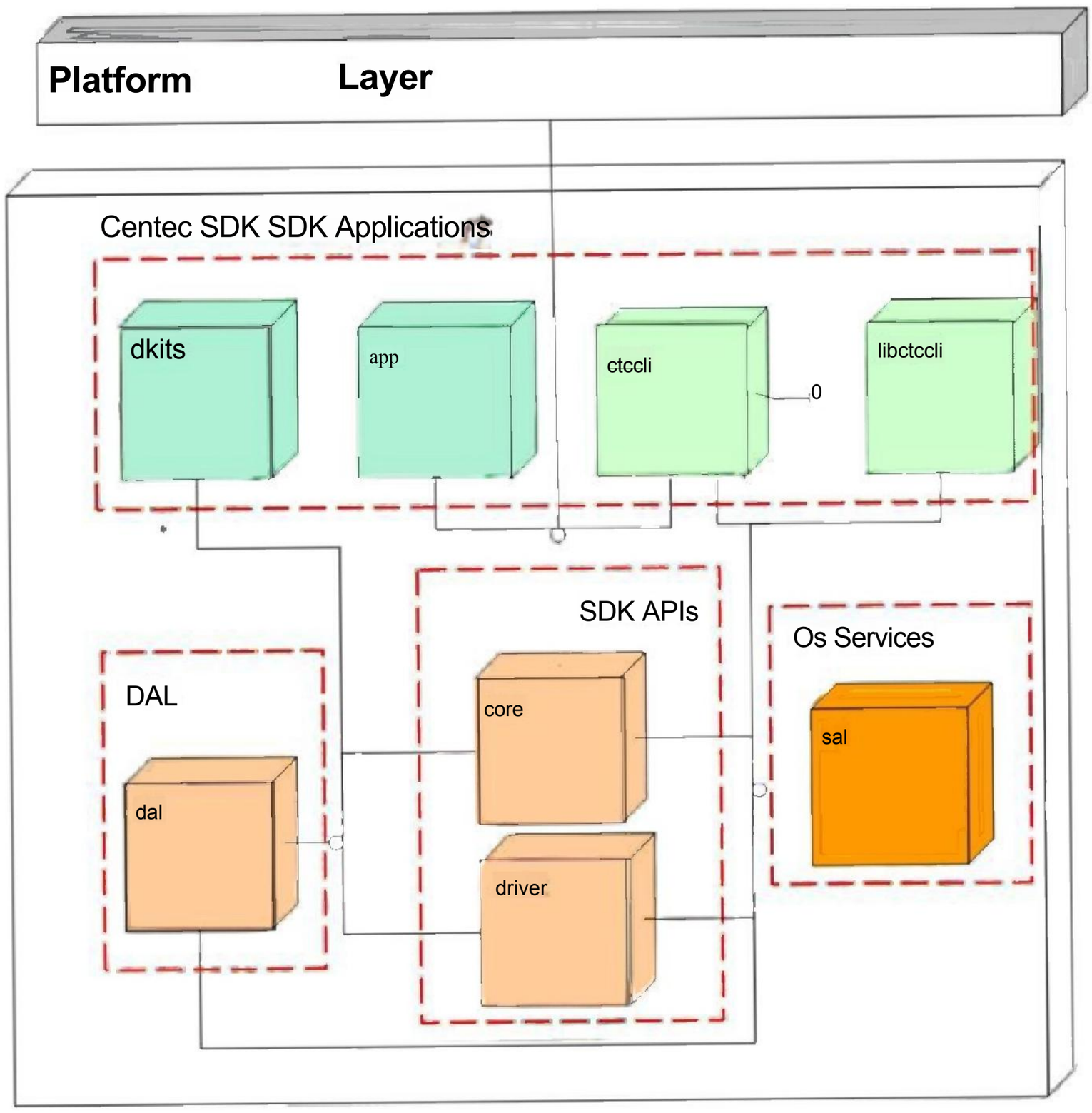
Platform
Layer
Centec SDK SDK Applications
dkits
app
ctccli
0
libctccli
SDK APIs
Os Services
DAL
core
sal
dal
driver

图2-4: SDK 组件

3 SDK 中的一些重要概念

Centec SDK中有几个重要概念，例如 gport,FID,Nexthop 等，这些概念在 SDK 使用广泛，是使用 SDK 的基础，本章节的目的在于对 SDK 中的相关重要概念进行阐述。

3.1 Port

端口是交换机和网络之间传递数据包的网络接口，在盛科的交换机芯片中把网络芯片端口区分为三种类型。

3.1.1本地端口(lport)

是指单芯片所管理的本地端口，在盛科中，本地端口不仅指物理端口，还包括中芯片的保留端口和内部端口，盛科的每一款芯片都有一定数量的本地端口，如Humber 最大支持256个端口，Greatbelt 支持128个端口，这些端口号可以作为索引可以得到 DsPhyPort,DsSrcPort 和 DsDestPort 等以端口为索引的表项属性，通常情况下，物理端口的范围由芯片能出的最大MAC 数目而决定，而保留端口对于每一款保留的端口数量也是不一样，除去物理端口和保留端口，剩下的端口全部都可以当做内部端口来使用，以 Greatbelt 为例，支持的端口情况如下图所示：

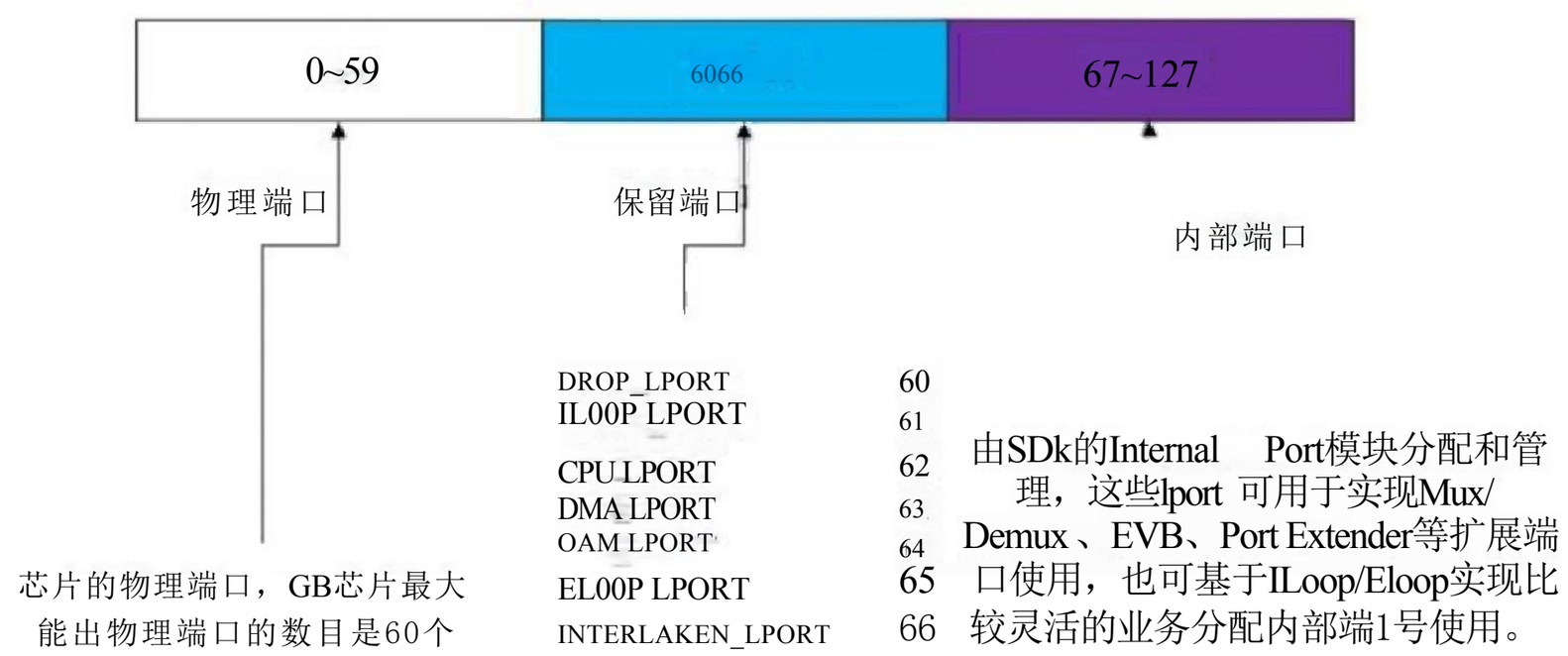

图3-1：**Greatbelt lport**

3.1.2全局端口(gport)

如果一个系统由多芯片组成(分布式系统或堆叠系统)时，使用lport是不能区分端口在不同的心片上，同时对于 Linkagg, 也不能使用lport 表示，此时就引入了全局端口即 global port, 简称gport, 如 下 ：

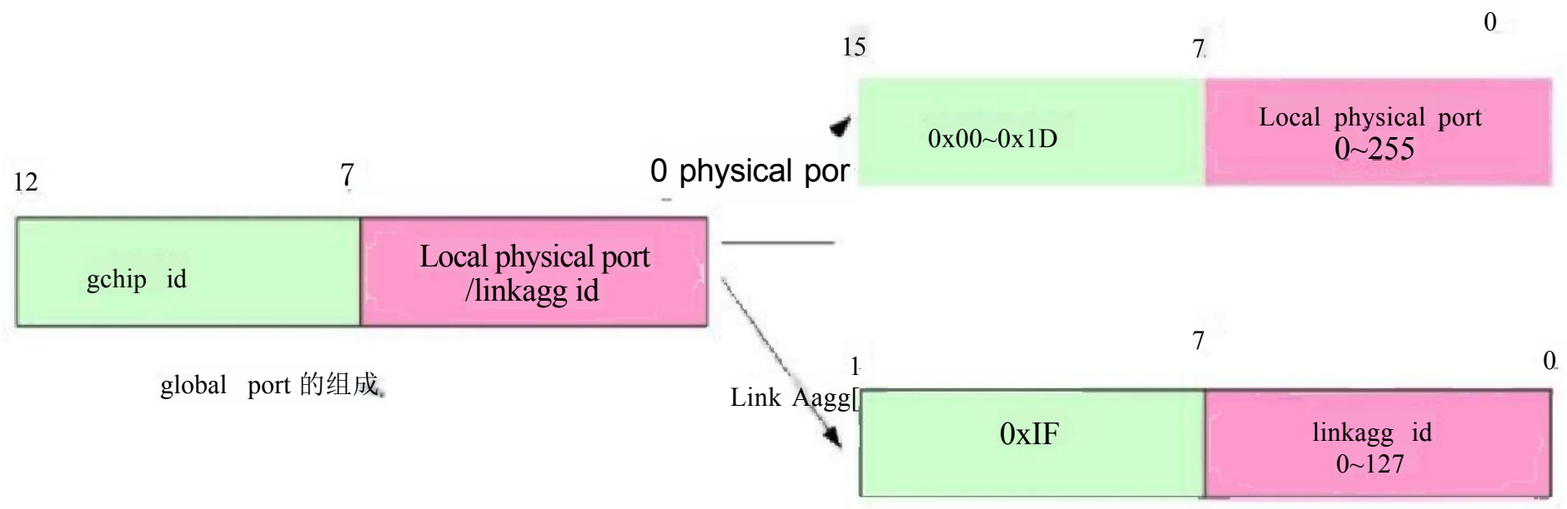

图3-2：gport

从图上可以看出，,gport 由两部份组成，其中高5 bit 表示 gchip, 指 的 是 全 局 指 定 芯片 ID, 在 SDK 初始化时调用 ctc_set_gchip_id(uint8 lchip,uint8 gchip)指定所在板卡所在的全局芯片ID, 预 留 0x1e给芯片内部使用；低8 bit 表示 lport 或 linkagg id.

3.1.3 逻辑端口 (logic port)

在芯片中逻辑端口并不直接对应一个交换机中的物理端口，它是指一种业务(或实例)在一个物理端口上的抽象念，在实际应用中一般以下两种情况可涉及到使用logic port。

VPLS 网络

在VPLS 网络中表示是一个 PW 端口或 AC 端口，当为 AC 端口时logic port可以通过 gport+vlan 或 gport 映射得到，但未PW 端口时可通过VC Label映射得到。

两层保护切换(APS) 网 络

在两层保护切换(APS) 网络时一对保护组的出口同属于一个logic port,可通过 Port 模块的接口把两个端口配置属于同一个logic port。

3.2 FID

FID全称为 Forwarding Instance ID,在二层网络中表示转发实例号，在盛科的芯片中所有的二层转发都是基于 MAC+FID 进行的，FID 可从以下映射得到：

- 基 于Vlan 的 L2 Bridge转发

  -IVL:vlan :fid 是1:1 关系，此种应用可配置为：Fid=vlan id

  -SVL:Vlan :fid 是n:1 关系，可通过配置 DsVlan.fid 对应VLAN 在 SVL 中的 FID。

以上都可以通过 SDK API:ctc_vlan_set_fid(uint16 vlan_id,uint16 fid)配置对应的 FID。

- 基 于C+S VLAN的转发

上层用户可通过Vlan mapping的接口ctc_vlan_add_vlan_mapping()把C+S 映射到一个 FID 中进行转发。

- 基于VSI的VPLS 的转发

在VPLS应用中，虚拟交换实例即VSI可映射为心片中的 FID, 在配置 AC 端口或 PW 端口时指定。

芯片最大支持 **FID**的范围可由用户在**FDB**模块初始化时来指定，不同的芯片，支持范围可能不一样，以 **Greatbelt**芯片序列为例，最大支持16k 个 FID。

3.3 Macst Group ID

在芯片中，**L2MC**、**IPMC**、基于**VLAN** 或基于**VSI**的广播都采用组播复制的方式来实现，每一个组播组都有一个标示来索引它，这个标示就是组播组 ID。同时在芯片中组播的实现是存放每一条组播条目的表项为**DsMetEntry,**并以链表结构来存储的，链表第一个条目的索引即为组播组 ID，然后接下来的条目由 DsMetEntry.nextMetPtr 串联得到，所以在 **SDK**中会为 **DSMetEntry** 预留一部分表项出来给 **Mcast Group ID** 来使用，这部分称作 Global DsMetEntry,剩余的给其他组播成员来分配使用称作 Local DsMetEntry,一般情况下Mcast Group ID由上层来分配和使用，下面以 CTC516x的默认 FTM Profile为例给出Mcast Group ID分配的示例。

表3-1 : Mcast Group ID分配

Mcast Type	Mcast Number	Mcast Group ID
Vlan based Broadcast	4k	0~4095
VSI based Broadcast	1k	4096-5119
L2 Multicast	1k	5120~6143
IP Multicast	1k	6144~7167

3.4 vrfId

虚拟路由转发实例，属于 L3 interface的属性。

3.5 L3 interface

L3 interface 是三层属性的接口，在Humber/Greatbelt 中L3 interfac区分为三种类型：Physical interface、Vlan interface和 Sub interface,全局支持1k interrace。

- Physical interface:也为 Routed port,仅仅能配置三层属性，只有路由功能，相当于路由器上的接口， Physical interface基于 Port 的，可直接连接路由器。
- *Vlan interface:两层交换机的Vlan 配置了路由接口，Vlan interface是基于Vlan,* ylan 接口集成了交换和路由的功能。
- Sub interface: 也为 Routed port,仅仅能配置三层属性，只有路由功能，但是 sub interface 是基于 Port+vlan 来区分的，在一个 Port 上可以通过 Vlan 来划分不同的 Inerace。

在Greatbel 中 ，L3 interrace 全局支持1k interrace。

3.6 Nexthop

Nexthop 是一个广义的概念，是对应芯片中一系列表项的总称，这些表项可能包含：DsFwd,,DsNexthop,,DsL2Edit,,DsL3Edit. 这些表项决定报文会被发往哪些端口，在出端口出去时对报文做哪些编辑动作，在不用的应用中Nexthop包含不同的动态表，如下：

- L2UC:DsFwd->DsNexthopReg(Internal Register)
- L2MC:DsMet->DsNexthopReg(Internal Register)
- IPUC:DsFwd(opt)->DsNexthop
- IPMC:DsMet->DsNexthop
- MPLS Push Nexthop:DsFwd->DsNexthop->DsL2EditEth->(DsL3EditMpls,opt)
- MPLS PHP Nexthop:DsFwd->DsNexthop->DsL2EditEth
- IP tunnel:DsFwd->DsNexthop->DsL2EditEth&DsL3EditTunnel
- ILoop:DsFwd
- ToCpu:DsFwd
- Drop:DsFwd
- BypassAll: DsFwd->DsNexthop
- STP tunnel nexthop:DsFwd->DsNexthop->DsL2EditEth

同时Nexthop区分为两种不同类型的Nexthp, 一个是外部 Nexthop, 由 SDK 的

使用者自己创建和删除，并以 NHID为索引分配和进行管理，上述的 IPUC、MPLS、IP Tunnel等都属于这一类，更多的接口参考 SDK的 Nexthop模块的API; 而另外 一种是内部 Nexthop, 是不需要用户参与分配和管理的，由SDK 代码自信来完成，也不需要指定 NHID, 上述的 L2UC/L2MC/IPMC/ILoop/ToCPI/Drop/BypassAll/STP tunnel 等 Nexthop都属于这一类。

4 配置开发环境

4.1 结构组织

4.1.1顶层

Centec SDK 根据 SDK 的组件划分代码 ，如下表所示。

表4-1：SDK 代码

top-level directory	
cfg/	SDK的配置文件
core/	包含SDK的核心代码，包含API、CTC及SYS层代码
ctccli/	录存放的是SDK的CLIs命令的代码
dal/	设备驱动层，位于Driver和底层OS之间的中间层。
driver/	存放上层访问芯片表项驱动的代码
sal/	存放封装与操作系统无关的代码
libctccli/	存放Centec CLI命令公共函数的代码。
Makefile	最上层的Makefile文件
mem_model/	存放盛科芯片软件仿真平台下的内存仿真模型
mk/	存放Makefile的
release_notes/	存放release notes文件

app/	存放初始化SDK的示例代码，用户可根据实际情况进行修改或替换
dkits/	存放调试心片工具的代码
ctc_shell/	存放的是CLIs命令shell的代码

4.1.2 core/ 子

$sdk/core/子　包含 SDK 的核心代码，包含 SDK 分层中的三层代码，分别是 API 层，CTC 层，SYS 层。API 层是对外封装与芯片无关的接口函数和数据结构，是所有芯片对外提供API接口的集合；CTC 层是封装芯片独立的API 函数，每个序列芯片都会提供 API 函 数 ； SYS 层操作软表，分配硬件表项的index, 维护不同表之间的关联，是 SDK 中核心处理代码。Ssdk/core/子　在芯片中主要完成以下功能：

- 对心片提供的功能进行抽象；
- 封装与芯片无关的功能接口；
- 对上层提供友好的接口，让产品的开发周期更短；
- 对芯片中表项索引的管理和维护；
- 对芯片中的表项及寄存器进行管理和维护。

表4-2:$sdk/core/子

top-level core/subdirectories	
common/	存放SDK公共的数据结构、宏、常量、错误码及公共算法库的相关文件.
ctc/	存放API层的头文件和源代码
init	SDK core初始化入口函数
greatbelt/	存放greatbelt序列芯片的CTC层及SYS层的源代码
humber/	存放humber序列芯片的CTC层及SYS层的源代码

Makefile	存放core 下Makefile文件
Makefile_libsdkcore.vx	存放core 在Vxworks环境下的Makefile文件

4.1.3 ctccli/ 子

$sdk/ctccli/ 子 存放的是 SDK 的 CLIs 命令的代码，完成以下几个功能：

- 为 API 函数提供完整的 CLIs命令，遵循一个 API 对应一个 CLI，用户可通过 CLI 的使用，熟悉和学习如何调用 API 函数；
- 提供丰富的调试芯片的的命令，用户可在线通过 SDK 提供的CLIs命令查看芯片中的表项，丢包原因，转发信息等等。

表4-3: Ssdk/ctccli/子

top-level ctccli/subdirectories	
diagcli/	包含调试 的 令的源代码。.
sdkcli/	API函数提供完整的CLIs命令的源代码
ctc_master_cli.c	引导SDK CLIs的入口
Makefile	存放ctccli 下Makefile文件
Makefile_libctccli.vx	存放ctccli 在Vxworks环境下Makefile文件
Makefile.vx	存放ctccli 在Vxworks环境下Makefile文件

4.1.4 sal/ 子

$sdk/kal/ 子 存放封装与操作系统无关的代码。Centec SDK 提供的 SAL 支持 Vxworks 和Linux 操作系统，如果用户使用的是其他操作系统，用户需要移植相关的代码到 SAL 中。

表4-4：$sdk/sal/子

top-level sal/subdirectories	
include	存放sal的头文件
src	存放sal源代码
Makefile	存放ctccli 下Makefile文件
Makefile_libctccli.vx	存放sal 在Vxworks环境下Makefile文件
Makefile.vx	存放sal 在Vxworks环境下Makefile文件

4.1.5 Driver/ 子

$sdk/driver/子 存放访问芯片表项驱动的代码，它提供的芯片中 table/register 读写操作接口。

表4-5：$sdk/driver/子

top-level driver/subdirectories	
humber	包含Humber序列芯片表项驱动的源代码
greatbelt	包含Greatbelt序列芯片表项驱动的源代码

4.1.6 Dal/ 子

Ssdk/dal/子 设备驱动层，位于 Driver 和底层 OS 之间的中间层，它封装访问芯片驱动相关的 API, 包 括 PCIe、Register、Table I/O、Serdes、中断处理等，它属于Driver的一部分。

表4-6：$sdk/dal/子

top-level dal/subdirectories	
include	包含dal的头文件
src	包含dal的源代码
Makefile	存放dal 下Makefile文件
Makefile_libctccli.vx	存放dal 在Vxworks环境下Makefile文件

4.1.7 libctccli/ 子

$sdk/libctccli/ 子 存放 Centec CLI命令公共函数的代码。

表4-7：$sdk/libctccli/子

top-level libctccli/subdirectories	
include	包含libctccli的头文件
src	包含libctccli的源代码
Makefile	存放libctccli 下Makefile文件
Makefile_libctccli.vx	存放libctccli 在Vxworks环境下Makefile文件

4.1.8 app/ 子

Ssdk/app/ 子 存放上层代码如何初始化 SDK。

表4-8:$sdk/app/ 子

top-level app/subdirectories	
config	SDK初始化各模块初始化的一些全局配置
usr	用户可以调用sdk的提供的接口实现一些功能并可把它作为SDK的一部分来管理和维护，该 由用户自己管理和维护。
sample	软件学习和老化FDB的示例代码以及通过Eth端口报文收发报文的示例代码，上层代码可根据实际应用需求参考示例代码进行修改。
ctc_app.c	引导SDK初始化的入口
Makefile	Linux下的Makefile文件
Makefile.vx	Vxworks下的Makefile文件

4.1.9 ctc_shell/ 子

$sdk/ctc_shell/子 存放 CLIs shell的代码，用于和CTC CLI处理函数通信。

表4-9:$sdk/ctc_shell/子

top-level ctc_shell/subdirectories	
ctc_shcll.c	ctc_shell的处理代码
Makefile	Linux下的Makefile文件

4.2如何编译 SDK?

4.2.1 Makefile 组织结构

SDK中 的Makefiles 组织如下图所示，在 Linux 环境下，$SDK 及所属子 都有一个 makefile 文件；而在 Vxworks 环境中每一个子 都 有 makefile。但只有 $SDK/makefile及$SDK/mk/sys.mk,$SDK/mk/sys_vx.mk 下的文件时可以修改的。

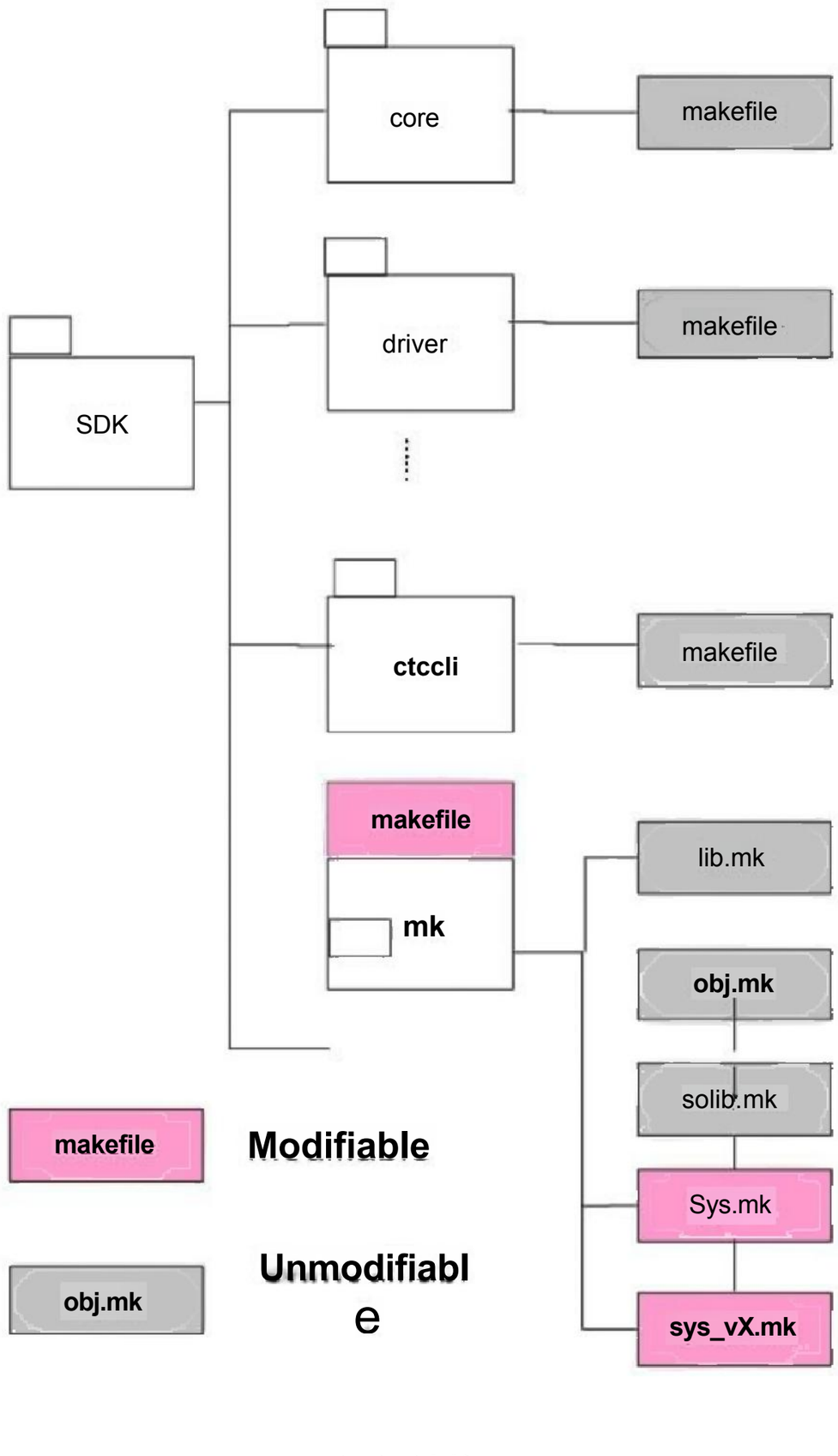

图4-1:SDK Makefiles组织结构

1. 通过$SDK/makefile 修改编译环境变量：

$CHIPNAME -- 指定编译的芯片名，目前可选humber/greatbelt

Stargetbase --指定编译所在的操作系统，目前可选 linux/vxworks

$BOARD-- 指定编译后所运行的平台类型，日前可选 linux-sim/linux-board

$ARCH --指 定CPU 的类型，目前支持powerpc/mips/freescale/loongson等架构的CPU。

$VER - 指定编译的版本时release版本还是 debug 版本，目前可选 r/d

$ PILE --指定编译 SDK 的交叉编译器。默认：=ppc-linux-/mips64-octeon-linux-gnu-/mipsel-linux-/powerpc-linux-gnu-

2. 配置交叉编译器存放路径

在bashrc文件中，对PATH这个环境变量指定交叉编译器的路径；

3. 通过$SDK/mk/sys.mk 或 SDK/mk/sys_vx.mk修改编译选项

其中$SDK/mk/sys.mk 用于 linux 平台下，而$SDK/mk/sys_vx.mk 用于 vxworks 环 境下，在这个文件中主要是根据编译环境设置 CPPFLAGS 和CFLAGS 两个编译选项，以下字段可能需要调整:

CPPFLAGS=-DHOST_IS_LE=0 //CPU是否是小端字节序

其他字段在使用时根据实际情况进行调整。

4. 2. 2在linux 环境下编译SDK

在 Linux 环境下要编译 SDK, 进 入SDK , 根据实际的编译环境，通过上一节章节的描述修改$SDK/makefile和$SDK/mk/sys.mk, 本文以编译 Cavium 5010为例修改这两个文件：

- 修改$/sdk/etc/bashrc文件，在PATH 这个环境变量指定交叉编译器的路径
- 修改$/sdk/Makefile

ifndefARCH

export ARCH=

endif

修改为

ifndef ARCH

export ARCH=mips

endif

Step1 找到

ifndef targetbase

export targetbase=

endif

Step2 修改为

ifndef targetbase

export targetbase=linux

endif

Step3 找到

ifndef BOARD

export BOARD=

endif

Step 4 修改为

ifndef BOARD

export BOARD=linux-board

endif

Step5 找到

ifndef CHIPNAME

export CHIPNAME=greatbelt

endif

Step6 修改为(目前可选为 greatbelt/humber,请正确选择对应芯片序列)

ifndef CHIPNAME

export CHIPNAME=greatbelt

endif

Step7 找到

ifndef SUBCHIPNAME

export SUBCHIPNAME=

endif

Step 8 修改为(请正确选择具体对应芯片型号)

ifndef SUBCHIPNAME

export SUBCHIPNAME=greatbelt

endif

备注：greatbelt芯片序列包含：greatbelt(ctc516x)/rialto(ctc316x)/rama(ctc512x)

humber芯片序列包含：humber(ctc6048)

Step 9 找到

ifeq($(ARCH),mips)

CPU=octeon

PILE =mips64-octeon-linux-gnu-

endif

Step 10 修改为(请正确选择具体对应芯片型号)

ifeq ($(ARCH),mips)

CPU=octeon

PILE =mips64-octeon-linux-gnu-

Endif

- 修改$/sdk/mk/sys.mk

Step11 找到

ifeq($(ARCH),mips)

CPPFLAGS=

endif

Step12 修改为

ifeq ($(ARCH),mips)

CPPFLAGS=-DHOST_IS_LE=0

CPPFLAGS+=-mabi=n32

endif

备注：根据 CPU大小端字节序进行修改

Step13 进入$/sdk/ ,执行以下命令就可以编译出最终的 Image和 SDK相关的库文件。

make

编译成功之后，你将在$ sdk/build.octeon.d/lib.linux-board找到5个库文件，上层应用代码可以包含ctc_api.h 和对应的5个库文件，然后再和上层应用代码一起编译生成一个应用程序，默认情况下，$SDK/app 充当了上层的应用代码，故执行完make命令以后你能在$sdk/build.octeon.d/bin.linux-board 找到一个可执行的SDK Image.

默认生成的 SDK Image在相应目标板上运行可以正常初始化芯片，并且芯片能在Vlan1中完成基本的二层转发功能，同时你也可以通过 SDK提供的CLI 配置芯片，并完成 API提供的所有功能。

表4-10:库文件

Library name	Description
Liblibctccli	Lib for SDK CLI,optional to integrate.

Libctccli	SDK CLI,optional to integrate.
Libdal	DAL.Must be integrated.
Libdrv	SDK Driver.Must be integrated.
Libsal	SAL.Must be integrated.
libsdkcore	SDK core.Must be integrated.

4.3如何集成SDK CLI?

SDK 的 CLI 工作在C/S 模式下。SDK 的中ctccli 和 libctccli 负责CLI 的解析，并且调用core 的处理逻辑，这部分处于server 端；而 shell 部分处于client 端，负责将输入的 CLI 命令传递给 server 端 。

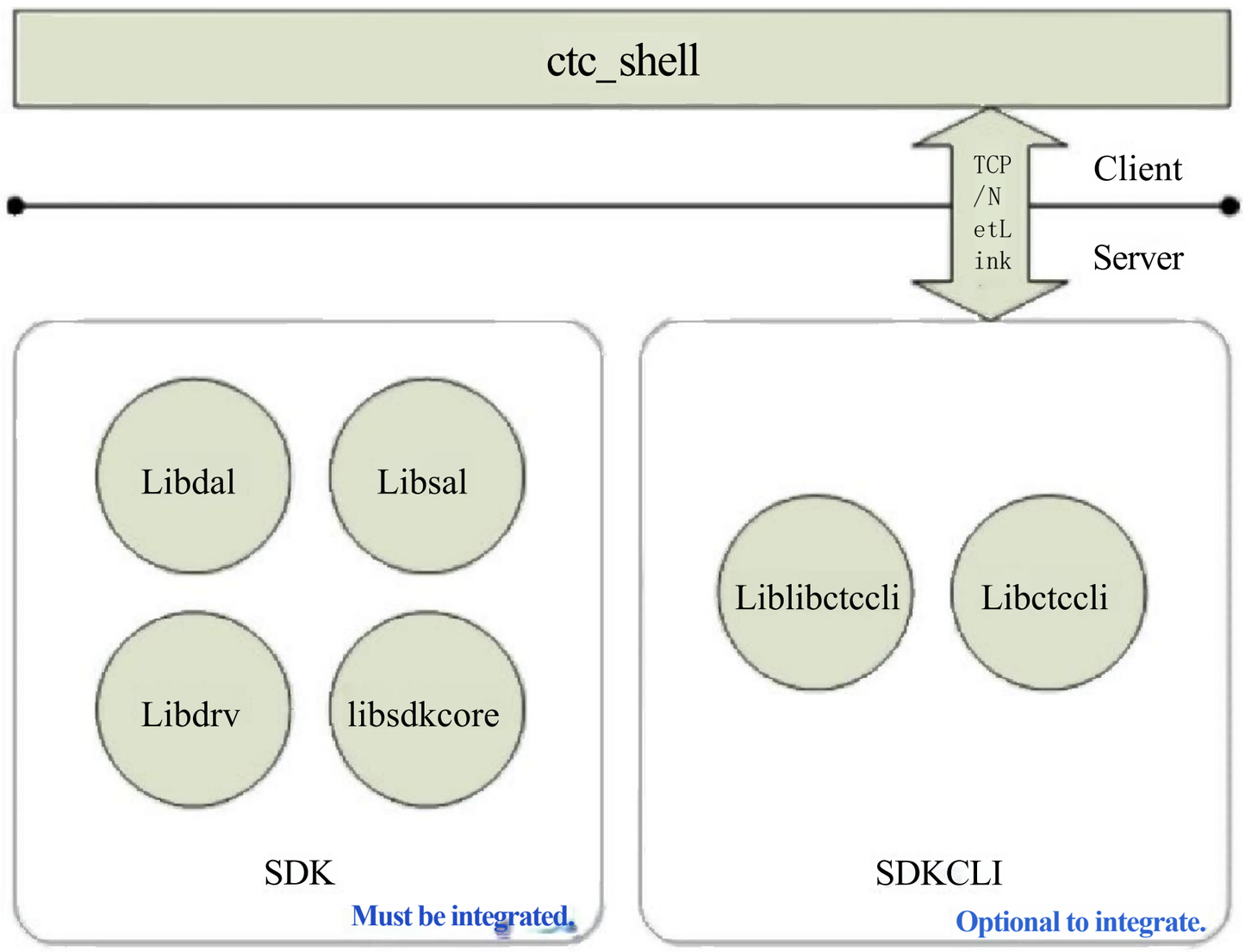

图4-2:SDK Makefiles组织结构

CTC CLI有两种工作模式， 一是Client和 Server在一个进程中；另一个是分开为 server 和 client 两个进程。可以通过 ctc_master_cli(is_ctc_shell);的参数来选择。 is_ctc_shell=1 表示两个进程模式。

对于C/S分开的模式通过下面两件步骤可以将 CLI集成到系统的命令行下：

1. 首先把server端生成的库libctccli,liblibctccli,libsdkcore 等与上层系统代码编译到一起，这样 SDK 就会同系统软件一同启动，包括 CLI 处理逻辑

2.ctc-shell 可单独编译成一个进程，单独启动与server 连接；也可以通过系统 shell 下的命令进行启动

备注：

当CLI server工作在kernel时，cli 主要通过netlink的方式通信，其通信netlink protocol 为20;可通过 ctc_shell.h 中的CTC_SDK_SIIELL_NETLINK进行配置修改

当CLI server工作在 user 时，cli主要通过tcp的方式通信，其L4 port 为8100;可通过 ctc_shell.h的 CTC_SDK_SHELL_TCP_PORT进行配置修改

ctc_shell可同时工作在 netlink/tcp这两者模式下，当启动ctc_shell时，通过指定 kernel 即可采用netlink方式，否则使用tcp 方式