衡芯平台应用软件开发手册

（版本1.0）

OpenTSN开源项目组

2021年1月

## 文档信息记录表

|  |  |  |
| --- | --- | --- |
| 版本号 | 版本内容 | 文档编写人 |
| 1.0 | 该版本为编程手册文档初始版本，主要包含衡芯软件框架所提供的关键数据结构及编程接口介绍、以及使用衡芯软件框架进行软件编程的示例 | 黄容、李军帅 |

目录

[衡芯平台应用软件开发手册 1](#_Toc61949979)

[文档信息记录表 2](#_Toc61949980)

[**1 引言** 4](#_Toc61949981)

[**2 总体概述** 4](#_Toc61949982)

[**3 核心数据结构** 6](#_Toc61949983)

[3.1 msg定义 6](#_Toc61949984)

[3.2 消息队列 8](#_Toc61949985)

[3.3上下文结构 10](#_Toc61949986)

[**4 软件接口** 12](#_Toc61949987)

[4.1 编程API 12](#_Toc61949988)

[4.1.1 资源操作API 12](#_Toc61949989)

[4.1.2 消息队列操作API 13](#_Toc61949990)

[4.1.3 空闲缓冲区管理API 14](#_Toc61949991)

[4.1.4 定时器API 15](#_Toc61949992)

[4.2 配置文本 16](#_Toc61949993)

[4.2.1 参数配置 16](#_Toc61949994)

[4.2.2 调度表配置 17](#_Toc61949995)

[**5 编程示例** 18](#_Toc61949996)

[5.1 功能开发 18](#_Toc61949997)

[5.1.1 创建目录和文件 18](#_Toc61949998)

[5.1.2 代码编写 19](#_Toc61949999)

[5.2 合入框架 21](#_Toc61950000)

[5.3 编译调试 22](#_Toc61950001)

**1 引言**

本文档为基于衡芯平台进行应用软件开发的用户提供编程指南，旨在为用户如何在衡芯平台进行应用软件开发提供帮助。

文档从用户编程视角概述如何在衡芯平台进行应用软件开发，主要内容包括衡芯应用软件框架概述、衡芯应用软件框架所提供的关键数据结构及编程接口详解、以及使用衡芯应用软件框架进行软件编程的示例等。

**2 总体概述**

为了方便用户基于衡芯平台进行应用软件的快速编程，OpenTSN开源项目组开发了一套衡芯应用软件框架。用户可基于这套应用软件框架进行相关应用功能的开发。

应用软件框架具有以下特点：

（1）框架提供基础的服务，如负责与外部控制CPU、与衡芯硬件平台进行通信，提供应用软件编程API等。用户不用关心数据报文如何获取，只需关心数据报文获取后如何处理。

（2）框架使用单进程多线程工作模式。每一个应用服务为一个线程，各个用户应用服务功能相互感知不到各自的存在。

（3）提供灵活的编排调度服务。各个线程间通信消息的编排由用户所提供的编排调度表进行控制。

应用软件框架与用户需要开发的应用软件之间的关系如下图所示。



应用软件框架与应用软件之间的关系示意图

衡芯应用软件框架总共包含三层：用户服务层、应用软件接口层、以及平台服务层。

用户服务层与用户应用相关，由用户按需进行定制。每一个应用服务为一个线程，各个应用服务功能相互感知不到各自的存在。

应用软件接口层包含编程API和配置文本两部分。编程API由软件框架提供，配置文本的格式软件框架已定义好，但具体内容与用户应用相关，由用户提供。

平台服务层包含编排调度服务和基础线程，由软件框架提供。编排调度服务根据用户所配置的编排调度表对各个线程间通信消息进行编排。基础线程负责与与外部控制CPU、与衡芯硬件平台进行通信，实现报文的收发。

**3 核心数据结构**

struct msg\_node、struct msg\_quene、struct msg\_quene\_node、struct hx\_context\_arg、struct hx\_context，这几个数据结构是用户基于衡芯软件框架进行软件开发必须使用的核心数据结构。

## 3.1 msg定义

各线程间传递的消息内容通过自定义的msg结构承载， msg似于skb\_buf结构。Msg的定义使用struct msg\_node结构体表示，结构体定义如下。

|  |
| --- |
| struct msg\_node  {  u64 msg\_type:8,//msg类型  src\_service\_id:8,//源服务ID  um\_type:4,//标识有无metedata和metedata长度  eth\_pkt\_len:16,//以太网报文长度  reserve:28;//保留位  u8\* pad\_head\_ptr;//填充报文头指针  u8\* eth\_head\_ptr;//以太网报文头指针  u64 ext\_domian[2];//扩展域  struct um\_metadata um;//数据报文metedata  }; |

各字段的含义：

（1）msg类型：msg类型字段用于标识线程间所传递的消息内容所属类别，便于编排调度拼接key。

msg类型根据用户需要可以不断扩充。目前定义的msg类型如下表。

|  |
| --- |
| //msg类型  typedef enum  {  NP\_NMAC\_RESPONSE = 0x00, //np NMAC响应  NP\_STATIC = 0x01, //np数据统计报文  NP\_SR = 0x02,//sr类型  }msg\_type; |

（2）源服务ID：标识msg消息上一跳处理线程的服务ID编号。

所有的服务线程都有一个线程编号，用于唯一标识线程的身份。msg的前两个字段，即msg类型和源服务ID，共同组成编排调度表需要的key。

服务ID编号根据用户需要也可以不断扩充。目前定义的服务ID类型如下。

|  |
| --- |
| //服务ID  typedef enum  {  DATA\_REQUEST\_ID= 0x00, //数据请求线程  DATA\_RESPONSE\_ID = 0x01, //数据响应线程  CONTROL\_REQUEST\_ID = 0x02, //控制请求线程  CONTROL\_RESPONSE\_ID = 0x03, //控制响应线程  SRV6\_SERVICE\_ID = 0x04, //SRV6服务线程  }service\_id; |

（3）um\_type：metedata的长度，主要用于标识数据报文发往硬件时metedata指针偏移量。

um\_type目前定义的类型如下。

|  |
| --- |
| //um类型  typedef enum  {  NON\_UM = 0x00, //不带metedata  NP\_UM = 0x01, //metedata长度32字节  TSN\_UM = 0x02, //metedata长度8字节  }um\_type; |

（4）eth\_pkt\_len：以太网报文长度。

（5）reserve：保留位；

（6）\*pad\_head\_ptr：填充报文头地址；

（7）\*eth\_head\_ptr：以太网报文头地址，即报文偏移metedata头之后的报文;



（8） ext\_domian：用户自定义字段，允许用户根据硬件平台和应用场景需要自定义字段以保存应用服务处理中间的结果；

（9） um\_ctrl：对于硬件送上来的数据报文有metedata头的情况，存放metedata信息。如果没有metedata头，则不用存放。

## 3.2 消息队列

为了避免访问内核态，线程间的通信使用自定义的用户态消息队列。即各线程间传递的msg结构，通过消息队列存储。

rxq为接收消息队列，txq为发送消息队列。

编排调度负责往rxq队列里写msg，往txq队列里读msg。

服务线程负责往rxq队列里读msg，往txq队列里写msg。



编排调度读写队列示意图

消息队列的定义使用struct msg\_quene结构体表示，结构体定义如下。

|  |
| --- |
| //消息队列  struct msg\_quene  {  u8 flag;//队列活跃标志.0:不活跃;1:活跃  u8 service\_id;//服务线程名称  u16 size;//队列大小  struct msg\_quene\_node \*front;//队列首指针  struct msg\_quene\_node \*rear;//队列尾指针  }; |

各字段的含义：

（1）flag：消息队列活跃标志，0为不活跃，1为活跃。由软件框架所提供的编程API进行设置；

（2）service\_id：表示这是哪个服务线程的消息队列，由用户设置；

（3）size：表示消息队列的深度，由用户设置；

（4）\*front：消息队列的头指针，由软件框架所提供的编程API进行设置；

（5）\*rear：消息队列的尾指针，由软件框架所提供的编程API进行设置。

\*front和\*rear用于判断消息队列是否满了或队列是否为空。\*front每读一个消息，值相应的进行更新，\*rear每写一个消息更新值，。

消息队列节点用于描述如何存储msg消息，使用struct msg\_quene\_node结构体表示，结构体定义如下。

|  |
| --- |
| //消息队列节点  struct msg\_quene\_node  {  struct msg\_node \*msg; //消息  struct msg\_quene\_node \*next;  }; |

各字段的含义：

（1）\*msg：msg指针，指向线程间通信的msg内容。

（2）\*next：next指针，指向链表的下一个节点。

## 3.3上下文结构

上下文结构包括上下文结构体参数和上下文结构体两个数据结构。上下文结构体参数使用struct hx\_context\_arg结构体表示，结构体定义如下。

|  |
| --- |
| //上下文结构体参数  struct hx\_context\_arg  {  u8 service\_id; //服务线程编号  u16 rxq\_size; //接收队列大小  u16 txq\_size; //发送队列大小  }; |

各字段的含义：

1. service\_id：服务线程所对应的服务ID编号；
2. rxq\_size：接收队列的深度；
3. txq\_size：发送队列的深度。

上下文结构体使用struct hx\_context结构体表示，结构体定义如下。

|  |
| --- |
| //上下文结构体  struct hx\_context  {  struct msg\_quene rxq; //接收队列  struct msg\_quene txq; //发送队列  struct buf\_list buffer\_list;//空闲缓冲区链表  }; |

各字段的含义：

（1）rxq：接收队列；

（2）txq：发送队列。

（3）buffer\_list：空闲缓冲区链表。软件框架提供统一的空闲缓冲区管理机制。

**4 软件接口**

软件接口包含编程API和配置文本两部分。编程API由软件框架提供，配置文本的格式软件框架已定义好，但具体内容与用户应用相关，由用户提供。

## 4.1 编程API

衡芯软件框架所提供的编程API分为基础API和应用API。

基础API指与衡芯平台相关的，如资源操作API、消息队列操作API、空闲缓冲区管理API等。

应用API指与用户应用服务相关的，如查表匹配API、定时器设置相关API等。

### 4.1.1 资源操作API

（1）资源初始化API

定义：int hx\_init(struct hx\_context \*tmp\_context,struct hx\_context\_arg \*arg);

输入参数：上下文结构体指针、上下文结构体参数指针

输出参数：成功返回0，失败返回-1

功能描述：完成消息队列的初始化、应用服务空闲缓冲区的申请。

首先根据上下文结构体参数判断是否需要初始化rxq队列和txq队列，然后完成消息队列的初始化和注册，最后再完成从空闲缓冲区资源池中获取一组空闲缓冲区地址的申请。

（2）资源释放API

定义：void hx\_destroy(struct hx\_context\* context);

输入参数：上下文结构体指针

输出参数：无

功能描述：完成消息队列的销毁、应用服务空闲缓冲区的释放。

首先根据上下文结构体队列参数判断是否rxq队列和txq队列存在，然后完成队列的销毁和注销，最后再完成组空闲缓冲区地址归还到空闲缓冲区资源池中。

### 4.1.2 消息队列操作API

（1）消息队列读消息API（阻塞式）

定义：struct msg\_node \* hx\_read\_msg\_quene\_node\_blocking (struct msg\_quene\* queen);

输入参数：消息队列指针

输出参数：成功返回消息节点，失败返回NULL

功能描述：若消息队列中有msg消息，则从消息队列中获取msg指针，并作为输出参数返回；否则等待，直到有msg消息再取出输出参数返回。

（2）消息队列写消息API（阻塞式）

定义：Int hx\_write\_msg\_quene\_node\_blocking(struct

msg\_quene\* queen, struct msg\_node \*node);

输入参数：消息队列指针，消息节点

输出参数：成功返回0，失败返回-1

功能描述：若消息队列未满，则将输入参数node写入消息队列中；否则等待，直到消息队列有空闲节点，将输入参数node写入消息队列中。

（3）消息队列读消息API（非阻塞式）

定义：struct msg\_node\* hx\_read\_msg\_quene\_node (struct msg\_quene\* queen);

输入参数：消息队列指针

输出参数：成功返回msg指针，失败返回NULL

功能描述：若消息队列中有msg消息，则从消息队列中获取msg指针，并作为输出参数返回；否则返回NULL。

（4）消息队列写消息API（非阻塞式）

定义：Int hx\_write\_msg\_quene\_node (struct msg\_quene\* queen, struct msg\_node \*node);

输入参数：消息队列指针，消息节点

输出参数：成功返回0，失败返回-1

功能描述：若消息队列未满，则将输入参数node写入消息队列中；否则返回-1。

### 4.1.3 空闲缓冲区管理API

（1）申请以太网报文buf地址API

定义：u8\* hx\_malloc\_eth\_pkt\_buf(struct buf\_list \*buf\_head);

输入参数：buf\_list指针

输出参数：成功返回eth报文buf地址，失败返回NULL

功能描述：从应用服务空闲缓冲区中申请一个空闲缓冲区块，并将eth\_pkt地址作为输出参数返回给用户。

（2）申请metedata报文buf地址API

定义：u8\* hx\_malloc\_metedata\_buf(struct buf\_list \*buf\_head);

输入参数：buf\_list指针

输出参数：成功返回metedata报文buf地址，失败返回NULL

功能描述：从应用服务空闲缓冲区中申请一个空闲缓冲区块，并将metedata报文地址作为输出参数返回给用户。

（3）释放buf地址API

定义：int hx\_free\_buf(struct buf\_list \*buf\_head,struct msg\_node \*msg);

输入参数：buf\_list指针，msg的指针

输出参数：成功返回0，失败返回-1

功能描述：将msg所对应的缓冲区块归还到应用服务空闲缓冲区中。

### 4.1.4 定时器API

（1）注册定时器API

定义：int hx\_register\_timer(int total\_time,int service\_id,int ismult);

输入参数：定时器周期、服务ID、是否循环定时

输出参数：成功返回0，失败返回-1

功能描述：向定时器线程注册定时周期，注册成功后，定时器线程会按照total\_time的周期性给该服务线程发送TSN\_TIMER类型的msg。

由于会有多个应用服务需要注册定时器，为了对msg类型进行区分，因此msg\_type = TSN\_TIMER + service\_id。

（2）注销定时器API

定义：int hx\_unregister\_timer(int service\_id);

输入参数：服务ID

输出参数：成功返回0，失败返回-1

功能描述：向定时器线程注销定时周期。完成注销后，定时器线程不再给该服务线程周期性发送TSN\_TIMER类型的msg。

## 4.2 配置文本

配置文本包含参数配置文本和调度表配置文本。参数配置文本用于配置跟编译环境相关的内容，调度表配置文本用于配置各个线程间通信消息的编排规则。

### 4.2.1 参数配置

参数配置由np.cfg文本定义。具体参数描述如下：

1. net\_interface：网卡接口名称。用于衡芯应用软件框架与衡芯硬件平台建立通信。
2. controller\_ip：外部控制CPU的IP地址。用于衡芯应用软件框架与外部CPU建立socket通信。
3. controller\_port：外部控制CPU的端口号。用于衡芯应用软件框架与外部CPU建立socket通信。

np.cfg文本:

|  |
| --- |
| net\_interface=enp0s17  controller\_ip=192.168.1.10  controller\_port=5555 |

### 4.2.2 调度表配置

调度表配置由schedule\_table文本定义。具体参数描述如下：

1. type：用于标识表项的类型。目前只有调度表schedule。
2. src\_service\_id：源服务ID号。标识msg消息上一跳处理线程的服务ID编号，也是调度表的索引项，格式使用十六进制表示。
3. msg\_type：消息类型。用于标识线程间所传递的消息内容所属类别，也是调度表的索引项，使用十六进制表示。
4. dst\_service\_id：目的服务ID号。标识msg消息需要发往处理线程的服务ID编号，也是调度表查表的结果，使用十六进制表示。

np.cfg文本:

|  |
| --- |
| {  type:schedule  src\_service\_id:00,msg\_type:04,dst\_service\_id:01  src\_service\_id:00,msg\_type:30,dst\_service\_id:10  src\_service\_id:10,msg\_type:30,dst\_service\_id:01  } |

**5 编程示例**

为了方便用户编程，提供编程示例供参考。

编程示例分为功能开发、合入框架和编译调试三部分。编程示例以TEST服务线程为例，TEST线程主要功能：

（1）接收所有报文并解析获取MAC地址

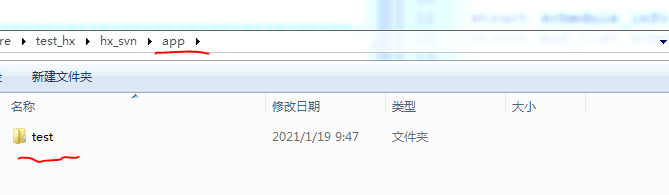
（2）根据MAC地址，指定该报文从对应的端口输出。

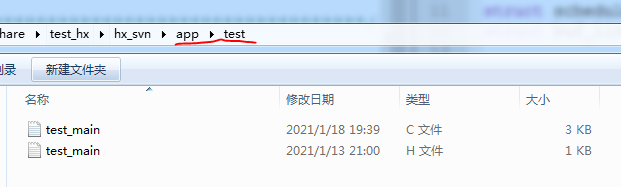
## 5.1 功能开发

### 5.1.1 创建目录和文件

在编程前，首先创建应用文件夹用于存储应用文件，然后创建应用文件用于编写代码。

1. 首先在app文件夹下创建应用文件夹，以test应用为例，创建test文件夹。
2. 在test文件夹下创建test应用的.c和.h文件，即test\_main.c和test\_main.h。





### 5.1.2 代码编写

在创建示例目录和文件后，开始进行代码编写。在编写代码时，首先需要定义消息类型和服务线程ID，然后创建线程，最后编写服务线程回调函数。

1. 定义消息类型和服务线程ID

在np.h中定义test消息类型和服务线程ID。

|  |
| --- |
| //msg类型  **typedef** enum  **{**  NP\_SR **=** 0x00**,**//sr类型    TEST **=** 0x30**,**//test  **}**msg\_type**;**  //服务ID  **typedef** enum  **{**  DATA\_REQUEST\_ID**=** 0x00**,** //数据请求线程  DATA\_RESPONSE\_ID **=** 0x01**,** //数据响应线程  CONTROL\_REQUEST\_ID **=** 0x04**,** //控制请求线程  CONTROL\_RESPONSE\_ID **=** 0x05**,** //控制响应线程    TEST\_SERVICE\_ID **=** 0x10**,** //test服务线程  **}**service\_id**;** |

1. 创建服务线程

在test\_main.c中编写创建服务线程的函数，并指定服务线程callback的名称。

创建服务线程伪代码示例：

|  |
| --- |
| int hx\_test\_handler**()**  **{**  pthread\_t test\_id**;**  //指定回调函数的名称hx\_test  ret**=**pthread\_create**(&**test\_id**,NULL,(**void **\*)**hx\_test**,NULL);**    **return** ret**;**  **}** |

1. 服务线程callback编写

Callback编写步骤如下：

（1）定义变量、设置参数

（2）资源初始化

（3）循环自己的rx\_q队列，取出msg

（4）对报文进行处理

（5）资源销毁

服务线程callback伪代码示例：

|  |
| --- |
| /\*测试线程\*/  //定义test\_context变量  struct hx\_context test\_context**;**  void hx\_test**(**void **\***argv**)**  **{**  //定义hx\_test\_arg变量  struct hx\_context\_arg hx\_test\_arg**;**  //设置参数：服务ID、发送队列和接收队列的长度  hx\_test\_arg**.**service\_id **=** TEST\_SERVICE\_ID**;**//  hx\_test\_arg**.**rxq\_size **=** 2048**;**  hx\_test\_arg**.**txq\_size **=** 2048**;**  //资源初始化  hx\_init**(&**test\_context**,&**hx\_test\_arg**);**    **while(**1**)**  **{**  //循环自己的rx\_q队列，取出msg  tmp\_msg\_node **=** hx\_read\_msg\_quene\_node\_blocking**(&(**test\_context**.**rxq**));**  /\*  对报文进行处理，获取报文的mac地址，然后指定输出端口号  \*/  **}**  //销毁资源  hx\_test\_contex\_destroy**();**  **}** |

## 5.2 合入框架

在对test功能开发完后，需要把test加入到hx的架构中，用于在hx架构在运行test。

1. 加入服务线程到主函数

在主函数中增加测试线程伪代码示例：

|  |
| --- |
| int main**(**int argc**,**char**\*** argv**[])**  **{**  hx\_txq\_rxq\_service\_list\_init**();**//平台服务队列初始化ALL  hx\_buf\_pool\_init**();**//平台缓存池初始化ALL  hx\_schedule\_init**();**//调度表部分初始化    /\*\*\*\*\*\*\*\*\*外部服务线程\*\*\*\*\*\*\*\*\*/  hx\_test\_handler**();**//在主函数中增加测试线程    hx\_data\_service\_thread\_init**();**//创建基础服务线程  **while(**1**)**//调度  **{**  hx\_msg\_schedule**();**  usleep**(**1**);**  **}**  hx\_schedule\_destroy**();**  hx\_buf\_pool\_destroy**();**//主线程结束，销毁缓存池（平台）  **return** SUCCESS**;**  **}** |

1. 填充get\_msg\_type\_from\_pkt函数以获取msg消息类型

伪代码示例：

|  |
| --- |
| u8 get\_msg\_type\_from\_pkt**(**u8**\*** pkt**,**u16 len**)**  **{**  /\*  解析接收到的报文，目前把所有的报文都送往TEST服务线程  \*/  msg\_type **=** 0x30**;**  **return** msg\_type**;** //返回消息类型为TEST    **}** |

## 5.3 编译调试

1. 修改makefile
2. 在makefile中添加对test服务线程的编译，生成test.o文件。
3. 在生成可执行文件hx时链接test.o文件。
4. 编写生成可执行文件
5. 执行make进行编译，生成可执行文件hx
6. 执行./hx，运行可执行文件。

makefile示例：

|  |
| --- |
| #!/bin/bash  CC**=**gcc  LDHDPATH **=** -I ./include  LDLIBS +**=**-lm -lpthread -lpcap -lnet  all**:** clean hx  hx**:**  $(CC) -o buf.o -c ./lib\_src/buf.c $(LDLIBS) $(LDHDPATH)  $(CC) -o quene.o -c ./lib\_src/quene.c $(LDLIBS) $(LDHDPATH)  $(CC) -o msg.o -c ./lib\_src/msg.c $(LDLIBS) $(LDHDPATH)    $(CC) -o data\_rec\_engine.o -c ./lib\_src/np\_data\_rec\_engine.c $(LDLIBS) $(LDHDPATH)  $(CC) -o data\_send\_engine.o -c ./lib\_src/np\_data\_send\_engine.c $(LDLIBS) $(LDHDPATH)  $(CC) -o basic.o -c ./lib\_src/basic.c $(LDLIBS) $(LDHDPATH)  $(CC) -o schedule.o -c ./lib\_src/schedule.c $(LDLIBS) $(LDHDPATH)  $(CC) -o tools.o -c ./lib\_src/tools.c $(LDLIBS) $(LDHDPATH)  $(CC) -o test.o -c ./app/test/test\_main.c $(LDLIBS) $(LDHDPATH) #test  $(CC) -o hx main.c buf.o quene.o msg.o data\_rec\_engine.o data\_send\_engine.o basic.o schedule.o tools.o test.o $(LDLIBS) $(LDHDPATH)    clean**:**  rm -rf hx \*.o |