T680/T660
RTOS SDK 使用说明

文档更新记录表

版本号	更新内容描述	更新人	更新日期
v0.9.0	初版	dortain	2019/6/27
v0.9.1	修改示例代码路径	dortain	2019/9/23
v0.9.2	添加 U 盘挂载说明及国际国密 SSL API 封装说明	dortain	2019/11/29

目录

1 概述

T680/T660 系列是由方寸微电子自主开发的新一代 Soc 网络终端存储安全芯片，具有功能丰富、性能强劲、功耗低、安全性高等特点，可广泛适用于加密移动硬盘、加密固态硬盘、视频链路加密机、VPN 终端网关、安全网关、网闸、单向导入导出设备、USB 安全网卡、密码卡、密码机、USB 接口芯片等众多安全领域产品。

芯片架构图如图 1.1 至图 1.2 所示。

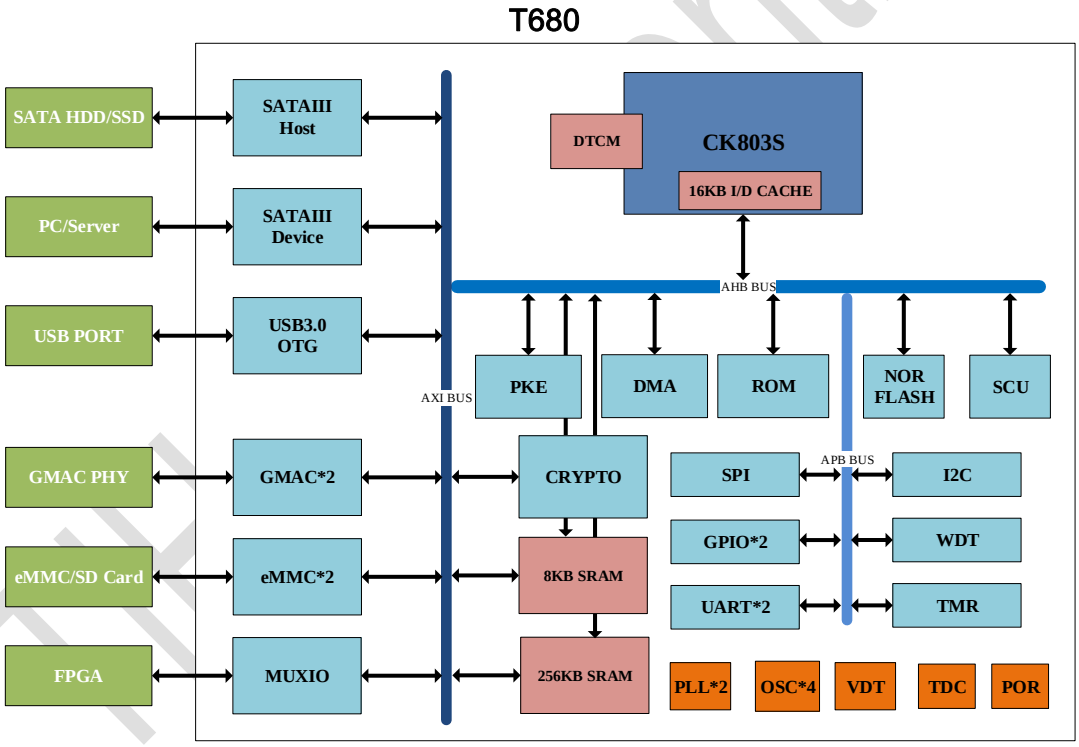


图 1.1 T680 芯片架构

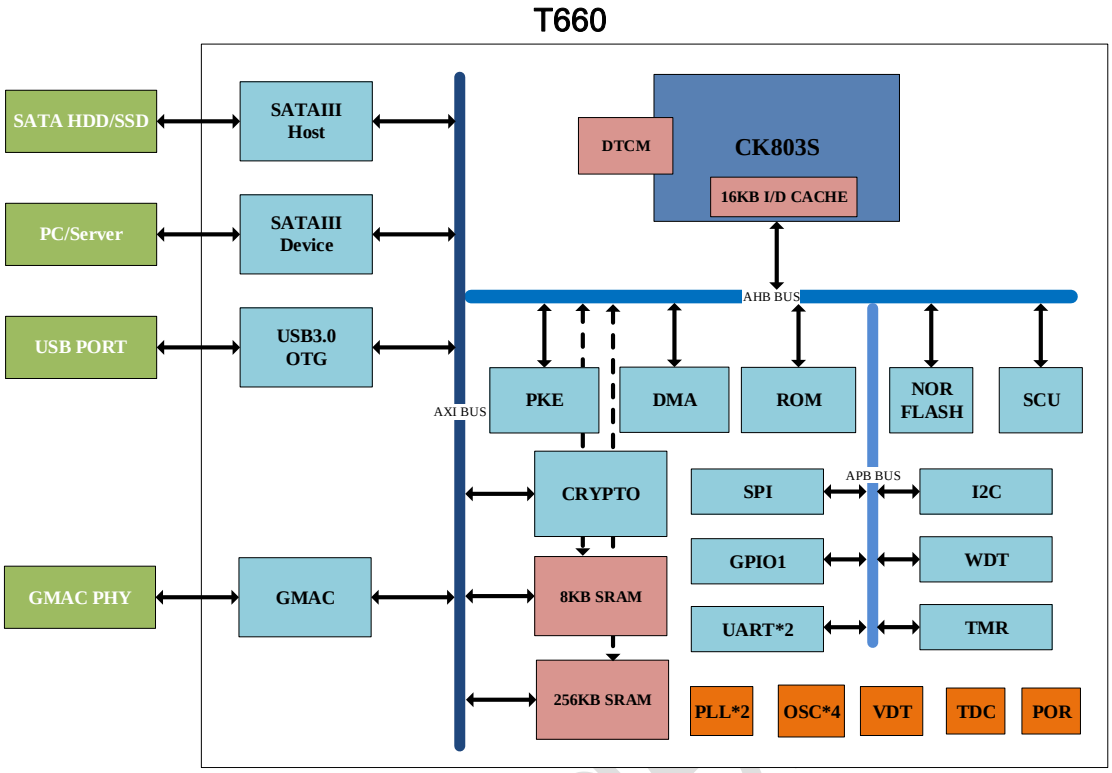


图 1.2 T660 芯片架构

T6x0 RTOS SDK 是基于 T680/T660（下文简称“T6x0”）芯片的 RTOS 开发平台，遵循了类 UNIX 接口规范和编码风格，采用了国产开源嵌入式操作系统 RT-Thread 作为 RTOS。基于 T6x0 RTOS SDK（下文简称“SDK”），开发者可方便快速地进行定制化应用方案开发，缩短产品开发周期、降低整体开发成本。本文档用于介绍 SDK 的相关开发说明。

SDK 由下列部分组成：

- T6x0 基础固件库
- T6x0 固件量产工具
- RTOS 源代码
- RTOS 扩展软件包
- RTOS 模块（API）例程
- SDK 相关文档

本文档第二章介绍 SDK 目录及 RTOS 简介，第三章介绍基于 RTOS 的 API 说明，第四章介绍开发指南。

2 SDK 说明

2.1 目录说明

2.1.1 SDK 顶层目录说明

SDK 顶层目录由 code、 document、 tool 三部分组成，如图 2.1 所示。

```
SDK
├─code        （SDK 代码文件）
├─document    （SDK 说明文档）
└─tools       （上位机工具及开发环境配置文件）
```

图 2.1 SDK 顶层目录结构

2.1.2 SDK 代码目录说明

```
code
├─header                          （上下位机公用头文件）
├─rt-thread                       （RT-Thread 目录）
│   ├─bsp
│   │   └─t6x0                    （T6x0 芯片 BSP 目录）
│   │       ├─app                 （应用类工程目录）
│   │       │   ├─entry
│   │       │   │   └─main.c      （main 线程用户 entry）
│   │       │   ├─xxx.cdkproj     （CDK 工程启动文件）
│   │       │   ├─rtconfig.h      （RT-Thread 组件配置文件）
│   │       │   ├─fw_jtag.ld      （Debug 链接脚本文件）
│   │       │   ├─fw.ld           （Release 链接脚本文件）
│   │       │   ├─clean.bat       （工程清理脚本）
│   │       │   └─……              （其他文件）
│   │       ├─drivers             （底层驱动移植代码）
│   │       └─libraries
│   │           ├─inc
│   │           │   └─tih         （T6x0 基础固件库头文件）
│   │           └─libt6x0_c.a     （T6x0 基础固件库）
│   ├─components                  （内核组件）
│   │   ├─dfs                     （虚拟文件系统组件）
│   │   ├─drivers                 （设备驱动框架组件）
│   │   ├─finsh                   （FinSH 命令行组件）
│   │   ├─libc                    （精简后的 libc 库组件）
```

```
    │   │   ├─net                    （网络协议栈 lwIP 组件）
    │   │   └─utilities              （工具类组件）
    │   ├─examples                   （各类 API 例程）
    │   │   ├─file                   （文件例程）
    │   │   ├─kernel                 （内核例程）
    │   │   ├─libc                   （libc 例程）
    │   │   ├─mbedtls                （mbed TLS 例程）
    │   │   ├─network                （网络例程）
    │   │   └─test                   （测试例程）
    │   ├─include                    （RT-Thread 头文件）
    │   ├─libcpu                     （CPU 硬件支持代码）
    │   ├─packages                   （扩展软件包）
    │   │   ├─lwext4-1.0.0           （EXT4 文件系统）
    │   │   ├─mbedtls-2.6.0          （mbed TLS）
    │   │   ├─netutils-1.0.0         （网络工具）
    │   │   ├─webclient-2.0.1        （http client）
    │   │   └─webnet-2.0.0           （http server）
    │   ├─src                        （RT-Thread 内核源代码）
    │   └─tools                      （RT-Thread 编译工具）
    └─software                       （上位机库及例程）
        ├─linux                      （Linux 平台）
        └─win                        （Window 平台）
```

图 2.2 SDK code 目录

2.2 RTOS 介绍

RT-Thread，全称是 Real Time-Thread，顾名思义，它是一个嵌入式实时多线程操作系统。它主要采用 C 语言编写，系统模块化设计及可用组件丰富并且可裁剪性非常好，浅显易懂，方便移植。它因实时性高、占用资源小等特点，非常适用于各种资源受限（如成本、功耗限制等）的场合。其中最小内核构成参见下图。

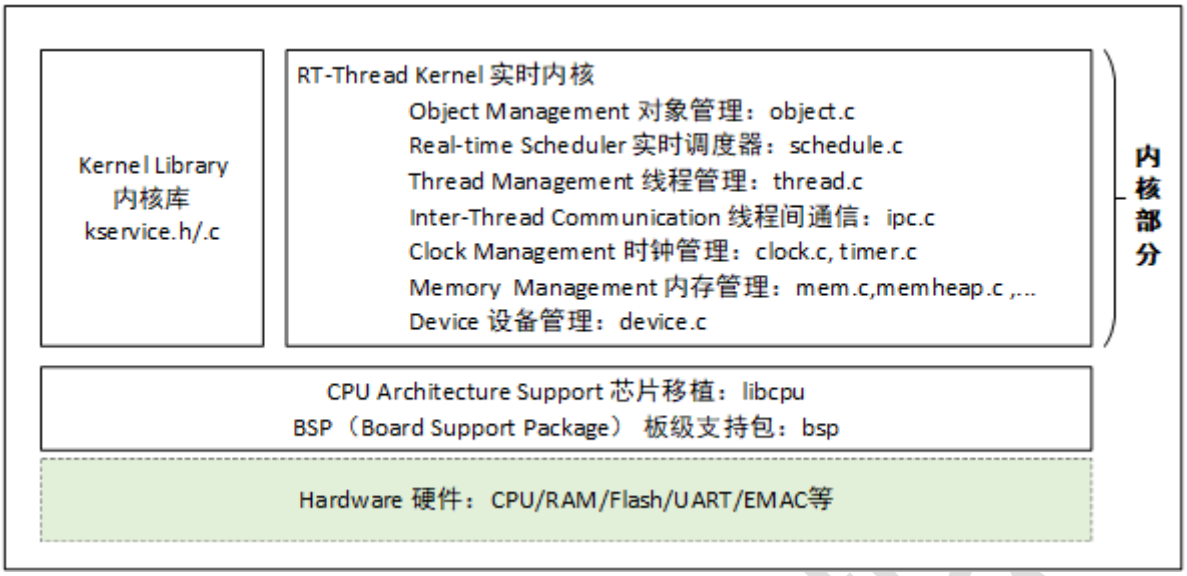


图 2.3 RT-Thread 最小内核构成

该 RTOS 与其他很多 RTOS 如 FreeRTOS、uC/OS 的主要区别还在于，它不仅仅是一个实时内核，还具备丰富的中间层组件和可选软件包，可扩展性功能比较强。为本 SDK 后续推出新的定制化组件及软件包提供了便利和参考。RT-Thread 整体框架图如图 2.4。

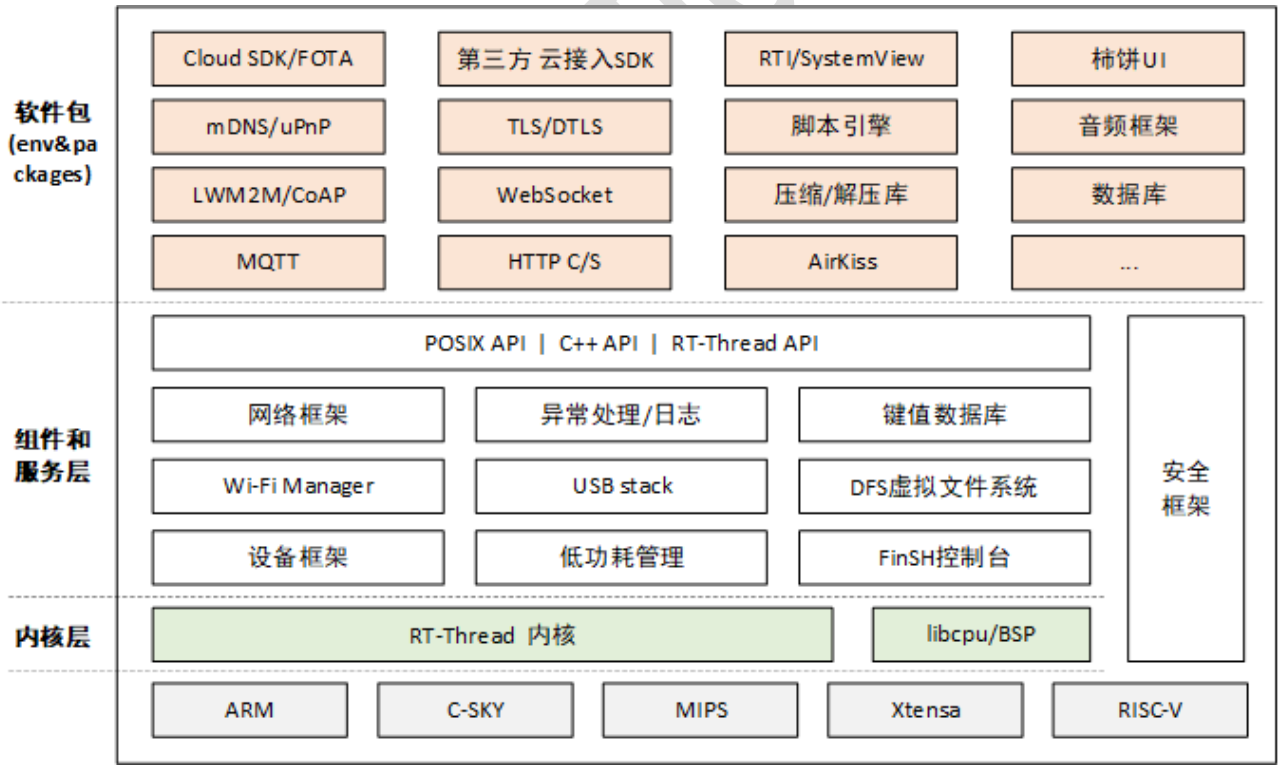


图 2.4 RT-Thread 架构图

2.2.1 系统介绍

本 SDK 采用了 RT-Thread 3.1.0 版本，遵循 GPL V2＋ 开源许可协议，可以免费在商业产品中使用，并且不需要公开私有代码。

RT-Thread 为了方便用户使用，遵循了类 UNIX 接口规范和编码风格，API 设计优雅。系统部分 API 主要由下面三块组成。

(1) RT-Thread 内核 API

RT-Thread API 设计精简，和系统本身结合最好，运行速度最高，但不方便其他 RTOS 的代码迁移。因此，该 API 比较适用于资源特别受限的场合。

(2) libc API

Libc 作为标准的 C 语言函数库，为 C 语言程序开发提供了便利，更逐渐成为操作系统的基础模块。一方面，RT-Thread 中提供了最基本的 libc 库，来方便进行用户程序的开发使用；另一方面，基于 GCC 编译工具链也为 RT-Thread 提供了精简后的必要的库函数支持（上面两种来源库函数，以下统称为 libc 库）。

(3) POSIX API

POSIX 是 “Portable Operating System Interface”（可移植操作系统接口）的缩写，POSIX 是 IEEE Computer Society 为了提高不同操作系统的兼容性和应用程序的可移植性而制定的一套标准。RT-Thread 在接口定义方面，除了自定义的 API 外，也兼顾了移植的便利性，实现了部分常用的 POSIX API 接口，如网络、文件系统、select、poll 及其他 API。

最后，具体详见第三章系统 API 和相应 RT-Thread 官网文档。

2.2.2 扩展软件包介绍

RT-Thread 提供了丰富的在线可选软件包。本 SDK 添加扩展了常用的软件包，方便了用户进行开发。

- mbed TLS 软件包

mbed TLS 是一个由 ARM 公司开源和维护的 SSL/TLS 算法库。其实现了 SSL/TLS 功能及各种加密算法，易于理解、使用、集成和扩展，方便开发人员轻松地在嵌入式产品中使用 SSL/TLS 功能。该软件包提供了 crypto 加解密库、SSL/TLS 库、X.509 证书库三种，方便了基于网络应用和本地线程应用的开发使用。本 SDK 针对 T6x0 支持的硬件加解密引擎对 mbed TLS 中的 crypto 库进行了硬件定制,详细请见 SDK 3.2.1 部分。

另外,基于 mbed TLS, 添加了部分 GMSSL 的功能，提供支持了国密套件 ECC-SM4-SM3。

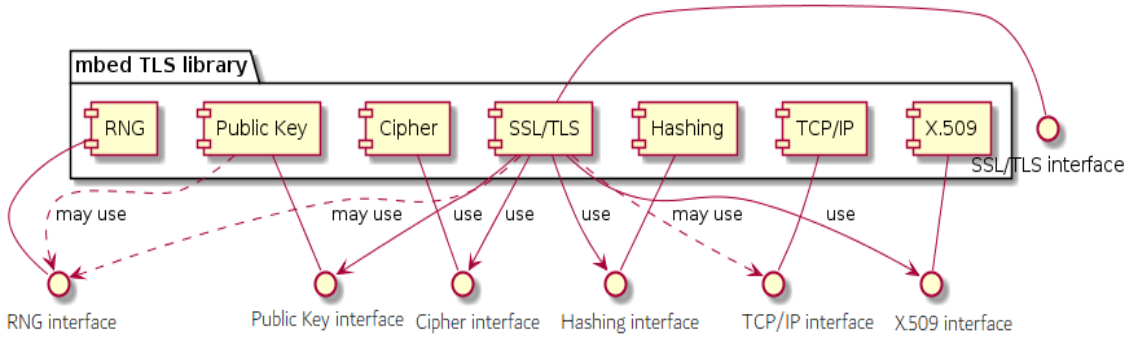


图 2.4 mbed TLS 架构图

- http client 软件包

WebClient 软件包是 RT-Thread 自主研发的，基于 HTTP 协议的客户端的实现，它提供设备与 HTTP Server 的通讯的基本功能外，还支持文件的上传和下载、HTTPS 加密传输等高级功能

- http server 软件包

WebNet 软件包是 RT-Thread 自主研发的，基于 HTTP 协议的服务器实现，而且支持多种模块功能扩展，如 CGI、文件上传、基本认证、HTTPS 加密传输等功能。

- utest 测试软件包

utest（unit test）是 RT-Thread 开发的单元测试框架，可以方便用户使用统一的框架接口编写测试程序，实现单元测试、覆盖测试以及集成测试的目的。应用框图如图 2.5：

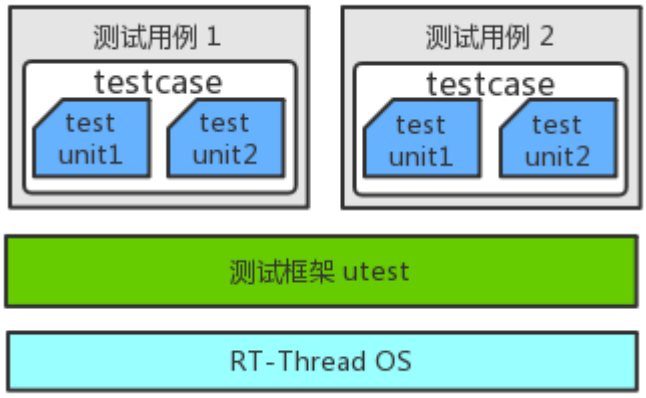


图 2.5 utest 应用框图

最后，以上软件包 API 请详见第三章软件包 API 章节和 RT-Thread 官网文档。

- net 工具软件包

NetUtils 作为 RT-Thread 自主整合的网络工具合集，具体使用可以参考官网 https://www.rt-thread.org/document/site/application-note/packages/netutils/an0018-system-netutils/，主要包括下面几类小工具：

名称	分类	描述
Ping	调试测试	利用 “ping” 命令可以检查网络是否连通，可以很好地帮助我们分

		析和判定网络故障
NTP	时间同步	网络时间协议
TFTP	文件传输	TFTP 是一个传输文件的简单协议，比 FTP 还要轻量级
Iperf	性能测试	测试最大 TCP 和 UDP 带宽性能，可以报告带宽、延迟抖动和数据包丢失

3 RTOS API

3.1 系统 API

RT-Thread 详细 API 介绍请参考官网文档 https://www.rt-thread.org/document/api/

3.1.1 RT-Thread API

3.1.1.1 最小内核 API

内核 API 的实现主要包括：对象管理、线程管理及调度器、线程间通信管理、时钟管理及内存管理等方面。常用 RT-Thread 内核 API 见下面函数列表：

函数名称
rt_thread_t rt_thread_create(const char* name, void (*entry)(void* parameter), void* parameter, rt_uint32_t stack_size, rt_uint8_t priority, rt_uint32_t tick);
rt_err_t rt_thread_delete(rt_thread_t thread);
rt_err_t rt_thread_init(struct rt_thread* thread, const char* name, void (*entry)(void* parameter), void* parameter, void* stack_start, rt_uint32_t stack_size, rt_uint8_t priority, rt_uint32_t tick);
rt_err_t rt_thread_detach (rt_thread_t thread);
rt_err_t rt_thread_startup(rt_thread_t thread);
rt_thread_t rt_thread_self(void);
rt_err_t rt_thread_yield(void);
rt_err_t rt_thread_sleep(rt_tick_t tick);
rt_err_t rt_thread_delay(rt_tick_t tick);
rt_err_t rt_thread_mdelay(rt_int32_t ms);
rt_err_t rt_thread_suspend (rt_thread_t thread);
rt_err_t rt_thread_resume (rt_thread_t thread);
rt_err_t rt_thread_control(rt_thread_t thread, rt_uint8_t cmd, void* arg);
rt_tick_t rt_tick_get(void);
rt_timer_t rt_timer_create(const char* name, void (*timeout)(void* parameter), void* parameter,

rt_tick_t time, rt_uint8_t flag);
rt_err_t rt_timer_delete(rt_timer_t timer);
void rt_timer_init(rt_timer_t timer, const char* name, void (*timeout)(void* parameter), void* parameter, rt_tick_t time, rt_uint8_t flag);
rt_err_t rt_timer_detach(rt_timer_t timer);
rt_err_t rt_timer_start(rt_timer_t timer);
rt_err_t rt_timer_stop(rt_timer_t timer);
rt_err_t rt_timer_control(rt_timer_t timer, rt_uint8_t cmd, void* arg);
rt_sem_t rt_sem_create(const char *name, rt_uint32_t value, rt_uint8_t flag);
rt_err_t rt_sem_delete(rt_sem_t sem);
rt_err_t rt_sem_init(rt_sem_t sem, const char *name, rt_uint32_t value, rt_uint8_t flag)
rt_err_t rt_sem_detach(rt_sem_t sem);
rt_err_t rt_sem_take (rt_sem_t sem, rt_int32_t time);
rt_err_t rt_sem_trytake(rt_sem_t sem);
rt_err_t rt_sem_release(rt_sem_t sem);
rt_mutex_t rt_mutex_create (const char* name, rt_uint8_t flag);
rt_err_t rt_mutex_delete (rt_mutex_t mutex);
rt_err_t rt_mutex_init (rt_mutex_t mutex, const char* name, rt_uint8_t flag);
rt_err_t rt_mutex_detach (rt_mutex_t mutex);
rt_err_t rt_mutex_take (rt_mutex_t mutex, rt_int32_t time);
rt_err_t rt_mutex_release(rt_mutex_t mutex);
rt_event_t rt_event_create(const char* name, rt_uint8_t flag);
rt_err_t rt_event_delete(rt_event_t event);
rt_err_t rt_event_init(rt_event_t event, const char* name, rt_uint8_t flag);
rt_err_t rt_event_detach(rt_event_t event);
rt_err_t rt_event_send(rt_event_t event, rt_uint32_t set);
rt_err_t rt_event_recv(rt_event_t event, rt_uint32_t set, rt_uint8_t option, rt_int32_t timeout, rt_uint32_t* recved);
rt_mailbox_t rt_mb_create (const char* name, rt_size_t size, rt_uint8_t flag);
rt_err_t rt_mb_delete (rt_mailbox_t mb);
rt_err_t rt_mb_init(rt_mailbox_t mb, const char* name, void* msgpool, rt_size_t size, rt_uint8_t flag)
rt_err_t rt_mb_detach(rt_mailbox_t mb);
rt_err_t rt_mb_send (rt_mailbox_t mb, rt_uint32_t value);
rt_err_t rt_mb_send_wait (rt_mailbox_t mb, rt_uint32_t value, rt_int32_t timeout);
rt_err_t rt_mb_recv (rt_mailbox_t mb, rt_uint32_t* value, rt_int32_t timeout);
rt_mq_t rt_mq_create(const char* name, rt_size_t msg_size, rt_size_t max_msgs, rt_uint8_t flag);
rt_err_t rt_mq_delete(rt_mq_t mq);
rt_err_t rt_mq_init(rt_mq_t mq, const char* name, void *msgpool, rt_size_t msg_size, rt_size_t pool_size, rt_uint8_t flag);
rt_err_t rt_mq_detach(rt_mq_t mq);
rt_err_t rt_mq_send (rt_mq_t mq, void* buffer, rt_size_t size);
rt_err_t rt_mq_urgent(rt_mq_t mq, void* buffer, rt_size_t size);

rt_err_t rt_mq_recv (rt_mq_t mq, void* buffer, rt_size_t size, rt_int32_t timeout);
rt_sighandler_t rt_signal_install(int signo, rt_sighandler_t[] handler);
void rt_signal_mask(int signo);
void rt_signal_unmask(int signo);
int rt_thread_kill(rt_thread_t tid, int sig);
int rt_signal_wait(const rt_sigset_t *set, rt_siginfo_t[] *si, rt_int32_t timeout);
rt_mp_t rt_mp_create(const char* name, rt_size_t block_count, rt_size_t block_size);
rt_err_t rt_mp_delete(rt_mp_t mp);
rt_err_t rt_mp_init(rt_mp_t mp, const char* name, void *start, rt_size_t size, rt_size_t block size);
rt_err_t rt_mp_detach(rt_mp_t mp);
void *rt_mp_alloc (rt_mp_t mp, rt_int32_t time);
void rt_mp_free (void *block);
void *rt_malloc(rt_size_t nbytes);
void rt_free (void *ptr);
void *rt_realloc(void *rmem, rt_size_t newsize);
void *rt_calloc(rt_size_t count, rt_size_t size);
void rt_interrupt_enter(void)
void rt_interrupt_leave(void)
rt_isr_handler_t rt_hw_interrupt_install(int vector, rt_isr_handler_t handler, void *param, char *name);
void rt_hw_interrupt_mask(int vector);
void rt_hw_interrupt_umask(int vector);
rt_base_t rt_hw_interrupt_disable(void);
void rt_hw_interrupt_enable(rt_base_t level);
rt_device_t rt_device_create(int type, int attach_size);
void rt_device_destroy(rt_device_t device);
rt_err_t rt_device_register(rt_device_t dev, const char* name, rt_uint8_t flags);
rt_err_t rt_device_unregister(rt_device_t dev);
rt_device_t rt_device_find(const char* name);
rt_err_t rt_device_init(rt_device_t dev);
rt_err_t rt_device_open(rt_device_t dev, rt_uint16_t oflags);
rt_err_t rt_device_close(rt_device_t dev);
rt_err_t rt_device_control(rt_device_t dev, rt_uint8_t cmd, void* arg);
rt_size_t rt_device_read(rt_device_t dev, rt_off_t pos,void* buffer, rt_size_t size);
rt_size_t rt_device_write(rt_device_t dev, rt_off_t pos,const void* buffer, rt_size_t size);
rt_device_t rt_device_find(const char* name);

3.1.1.2 内核网络组件 API

系统内核网络组件集成了 LwIP 协议栈来实现 TCP/IP 网络功能，大部分接口 API 兼容 POSIX 标准（详细参见 3.1.3.1 部分），为了开发移植方便，也有部分 API 继承了常见 Linux 版本形式，主要函数列表如下：

函数名称
int inet_pton(int family, const char *strptr, void *addrptr)
const char *inet_ntop(int family, const void *addrptr, char *strptr, size_t len)
struct hostent* gethostbyname(const char * hostname)
int gethostbyname_r(const char *name, struct hostent *ret, char *buf, size_t buflen, struct hostent **result, int *h_errnop)
int getaddrinfo(const char *nodename, const char *servname, const struct addrinfo *hints, struct addrinfo **res)
void freeaddrinfo(struct addrinfo *ai)
unsigned int if_nametoindex(const char *ifname)
char *if_indextoname(unsigned int if_index, char *ifname)

3.1.2 Libc API

该 libc 库包含了字符串数字处理、malloc 内存管理、时间处理、随机数等函数 API。主要函数 API 见下面函数列表:

（1） 字符串数字处理函数列表:

函数名称
char *strcpy(char *dest, const char *src)
char *strncpy(char *dest, const char *src, size_t siz)
size_t strlcpy(char *dst, const char *src, size_t siz)
int strcmp (const char *s1, const char *s2)
int strncmp(const char *cs,const char *ct, size_t count)
char *strcat(char * dest, const char * src)
char *strncat(char *dest, const char *src, size_t count)
char *strrchr(const char *t, int c)
int strncasecmp (const char* s1, const char* s2, size_t len)
int strcasecmp(const char *a, const char *b);
int tolower(int c)
int toupper(int c)
int vsscanf(const char * buf, const char * fmt, va_list args)
int sscanf(const char * buf, const char * fmt, ...)
size_t strspn(const char *s, const char *accept)
size_t strcspn(const char *s, const char *reject)
char *strtok(char *s, const char *delim)
char *strchr(const char *s1, int i)
char *strstr(const char * s1,const char * s2);
char *strdup(const char *s);
size_t strlen(const char *s);
long strtol(const char *str, char **endptr, int base)

long long strtoll(const char *str, char **endptr, int base)
double atof(_CONST char *s);
int atoi(_CONST char *s);
long atol(_CONST char *s);
long long atoll(_CONST char *str);
double strtod(const char *nptr, char **endptr);
float strtof(_CONST char *nptr, char **endptr);
long strtol(const char *nptr, char **endptr, int base);
long double strtold (const char *nptr, char **endptr);
long long strtoll(const char *nptr, char **endptr, int base);
unsigned long strtoul(const char *nptr, char **endptr, int base);
unsigned long long strtoull(const char *nptr, char **endptr, int base);
int abs(int i);
div_t div(int n, int d);
long labs(long i);
ldiv_t ldiv(long n, long d);
long long llabs(long long j);
lldiv_t lldiv(long long n, long long d);

（2） 内存管理函数列表

函数名称
void *malloc(size_t size)
void free(void *ptr)
void *realloc(void *ptr, size_t size)
void *calloc(size_t nelem, size_t elsize)

（3） 时间处理函数列表

函数名称
int clock_time_to_tick(const struct timespec *time)
int clock_getres(clockid_t clockid, struct timespec *res)
int clock_gettime(clockid_t clockid, struct timespec *tp)
int clock_settime(clockid_t clockid, const struct timespec *tp)
struct tm *gmtime_r(const time_t *timep, struct tm *r)
struct tm* localtime_r(const time_t* t, struct tm* r)
struct tm* localtime(const time_t* t)
time_t mktime(struct tm * const t)
char *asctime_r(const struct tm *t, char *buf)
char *asctime(const struct tm *timeptr)
char *ctime(const time_t *timep)
int gettimeofday(struct timeval *tp, void *ignore)
int _gettimeofday(struct timeval *tv, void *ignore)
time_t time(time_t *t)

RT_WEAK clock_t clock(void)

（4） 随机数函数列表

函数名称
int rand(void);
void srand (unsigned int seed);

3.1.3 POSIX API

3.1.3.1 网络 API

系统内核网络组件集成了 LwIP 协议栈并向用户应用程序提供标准了 POSIX Socket 接口，支持类文件操作 read/write 等，极大的提高了系统的兼容性。详请参见 RT-Thread 官网 https://www.rt-thread.org/document/api/group___n_e_t.html。

Socket 主要函数列表：

函数名称
int socket (int domain, int type, int protocol);
int bind (int s, const struct sockaddr * name, socklen_t namelen);
int listen (int s, int backlog);
int accept(int s, struct sockaddr * addr, socklen_t * addrlen);
int connect(int s, const struct sockaddr * name, socklen_t namelen);
int send (int s, const void * dataptr, size_t size, int flag);
int recv (int s, void * mem, size_t len, int flags);
int sendto(int s, const void * dataptr, size_t size, int flags, const struct sockaddr * to, socklen_t tolen);
int recvfrom(int s, void * mem, size_t len,int flags, struct sockaddr * from, socklen_t * fromlen);
int closesocket (int s);
int shutdown(int s, int how);
int setsockopt(int s, int level, int optname, const void * optval, socklen_t optlen);
int getsockopt(int s, int level, int optname, void * optval, socklen_t * optlen);
int getpeername(int s, struct sockaddr * name, socklen_t * namelen);
int getsockname(int s, struct sockaddr * name, socklen_t * namelen);
int ioctlsocket(int s, long cmd, void * arg);

3.1.3.2 文件 API

RT-Thread 实现了部分文件操作的 POSIX 标准接口，可以很方便的将 Linux/Unix 的程序移植到 RT-Thread 操作系统上。主要函数 API 见下面函数列表：

函数名称
int open(const char *file, int flags, ...)
int close(int fd)
int read(int fd, void *buf,int len)
int write(int fd, const void *buf, int len)
off_t lseek(int fd, off_t offset, int whence)
int rename(const char *old, const char *new)
int unlink(const char *pathname)
int stat(const char *file, struct stat *buf)
int fstat(int fildes, struct stat *buf)
int fsync(int fildes)
int fcntl(int fildes, int cmd, ...)
int ioctl(int fildes, int cmd, ...)
int statfs(const char *path, struct statfs *buf)
int mkdir(const char *path, mode_t mode)
int rmdir(const char *pathname)
DIR *opendir(const char *name)
struct dirent *readdir(DIR *d)
long telldir(DIR *d)
void seekdir(DIR *d, off_t offset)
void rewinddir(DIR *d)
int closedir(DIR *d)
int chdir(const char *path)
int access(const char *path, int amode)
char *getcwd(char *buf, size_t size)

3.1.3.3 多路复用 API

IO 多路复用是指内核一旦发现进程指定的一个或者多个 IO 设备/文件准备就绪，它就通知该进程。本系统提供了两个兼容 POSIX 标准的 IO 多路复用函数: select 和 poll。两者功能及内部实现相似，仅接口形式不同。

（1） **select**

原型	int select(int nfds , fd_set *readset, fd_set *writeset, fd_set *exceptset,struct timeval *timeout)
说明	监视 I/O 设备状态

	该函数接口可以阻塞地同时探测一组支持非阻塞的 I/O 设备是否有事件发生（如可读，可写，有高优先级的错误输出，出现错误等等），直至某一个设备触发了事件或者超过了指定的等待时间。
输入参数	nfds：是一个整数值，是指集合中所有文件描述符的范围，即所有文件描述符的最　大值加 1。 readfds：（可选），指向一组等待可读性检查的文件。 writefds：（可选），指向一组等待可写性检查的文件。 exceptfds：（可选），指向一组等待错误检查的文件。 timeout：select()最多等待时间，对阻塞操作则为 NULL。
输出参数	readfds：（可选）指向一组等待可读性检查的文件。 writefds：（可选）指向一组等待可写性检查的文件。 exceptfds：（可选）指针，指向一组等待错误检查的文件。
返回值	<0：出现错误，此时所有描述符集会被清 0; =0：超时; >0：已经准备好的文件描述符数;

（2） **poll**

原型	int poll(struct pollfd *fds, nfds_t nfds, int timeout)
说明	监视 I/O 设备状态 该函数接口可以阻塞地同时探测一组支持非阻塞的 I/O 设备是否有事件发生（如可读，可写，有高优先级的错误输出，出现错误等等），直至某一个设备触发了事件或者超过了指定的等待时间。
输入参数	fds：是一个 struct pollfd 结构类型的数组，列出了我们需要 poll()检查的文件描述符; nfds：fds 中元素的个数; timeout：决定阻塞行为: -1：一直阻塞到 fds 数组中有一个达到就绪态或者捕获到一个信号 0：不会阻塞，立即返回 >0:阻塞时间
输出参数	fds：struct pollfd 结构类型的数组，当函数返回值>0 时，指示文件实际发生的事件（可读、可写）
返回值	<0：出现错误，函数调用失败; =0：poll 超时，数组 fds 中没有任何描述符指定的文件准备好读、写; >0：数组 fds 中准备好读、写文件描述符的总数量;

3.1.3.4 其他 API

除了上述 3 大类 POSIX API 接口外，还有一些常用的 POSIX 接口 API：

函数名称
unsigned int sleep(unsigned int seconds)

3.2 扩展软件包 API

3.2.1 Mbedtls API

mbed TLS 中提供了国际通用的 crypto 加解密库、SSL/TLS 库、X.509 证书库。而 T6x0 芯片内部集成了常用的硬件加解密引擎，因此本 SDK 对 mbed TLS 中 crypto 算法库进行了硬件加速定制，且保留了原有的 mbed TLS 算法库 API 接口，主要涉及了国际通用算法：AES、RSA、ECC、SHA1/224/256、RNG 随机数以及国密算法 SM2、SM3 和 SM4。借助于 SSL/TLS 库和 X.509 证书库,可以方便进行 SSL 应用类开发。具体请参考 mbed TLS 官网 API 介绍 https://tls.mbed.org/api/

从第一章 T6x0 芯片架构中了解到，CRYPTO 硬件模块挂载于 AXI 总线，可以访问 256KB SRAM 和 8KB SRAM 存储空间，但是不能访问 32KB 的 D-TCM 存储空间。因此，特别需要注意的是，在使用硬件模块进行 AES 和 SM4 对称加解密时，数据输入和输出地址不能来自于 D-TCM 存储空间。

Crypto 加解密库函数 API 主要列表见下面章节:

3.2.1.1 AES API

函数名称
void mbedtls_aes_init(mbedtls_aes_context *ctx);
void mbedtls_aes_free(mbedtls_aes_context *ctx);
int mbedtls_aes_setkey_enc(mbedtls_aes_context *ctx, const unsigned char *key, unsigned int keybits);
int mbedtls_aes_setkey_dec(mbedtls_aes_context *ctx, const unsigned char *key, unsigned int keybits);
int mbedtls_aes_crypt_ecb(mbedtls_aes_context *ctx, int mode, const unsigned char input[16], unsigned char output[16]);
int mbedtls_aes_crypt_cbc(mbedtls_aes_context *ctx, int mode, size_t length, unsigned char iv[16], const unsigned char *input, unsigned char *output);
int mbedtls_aes_crypt_cfb128(mbedtls_aes_context *ctx, int mode, size_t length, size_t *iv_off, unsigned char iv[16], const unsigned char *input, unsigned char *output);
int mbedtls_aes_crypt_cfb8(mbedtls_aes_context *ctx, int mode, size_t length, unsigned char iv[16],

const unsigned char *input, unsigned char *output);
int mbedtls_aes_crypt_ctr(mbedtls_aes_context *ctx, size_t length, size_t *nc_off, unsigned char nonce_counter[16], unsigned char stream_block[16], const unsigned char *input, unsigned char *output);

3.2.1.2 RSA API

函数名称
void mbedtls_rsa_init(mbedtls_rsa_context *ctx, int padding, int hash_id);
void mbedtls_rsa_set_padding(mbedtls_rsa_context *ctx, int padding, int hash_id);
int mbedtls_rsa_gen_key(mbedtls_rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng, unsigned int nbits, int exponent);
int mbedtls_rsa_check_pubkey(const mbedtls_rsa_context *ctx);
int mbedtls_rsa_check_privkey(const mbedtls_rsa_context *ctx);
int mbedtls_rsa_check_pub_priv(const mbedtls_rsa_context *pub, const mbedtls_rsa_context *prv);
int mbedtls_rsa_public(mbedtls_rsa_context *ctx, const unsigned char *input, unsigned char *output);
int mbedtls_rsa_private(mbedtls_rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng, const unsigned char *input, unsigned char *output);
int mbedtls_rsa_pkcs1_encrypt(mbedtls_rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng, int mode, size_t ilen, const unsigned char *input, unsigned char *output);
int mbedtls_rsa_rsaes_pkcs1_v15_encrypt(mbedtls_rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng, int mode, size_t ilen, const unsigned char *input, unsigned char *output);
int mbedtls_rsa_rsaes_oaep_encrypt(mbedtls_rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng, int mode, const unsigned char *label, size_t label_len, size_t ilen, const unsigned char *input, unsigned char *output);
int mbedtls_rsa_pkcs1_decrypt(mbedtls_rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng, int mode, size_t *olen, const unsigned char *input, unsigned char *output, size_t output_max_len);
int mbedtls_rsa_rsaes_pkcs1_v15_decrypt(mbedtls_rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng, int mode, size_t *olen, const unsigned char *input, unsigned char *output, size_t output_max_len);
int mbedtls_rsa_rsaes_oaep_decrypt(mbedtls_rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng, int mode, const unsigned char *label, size_t label_len, size_t *olen, const unsigned char *input, unsigned char *output, size_t output_max_len);
int mbedtls_rsa_pkcs1_sign(mbedtls_rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng, int mode, mbedtls_md_type_t md_alg, unsigned int hashlen, const unsigned char *hash, unsigned char *sig);

int mbedtls_rsa_rsassa_pkcs1_v15_sign(mbedtls_rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng, int mode, mbedtls_md_type_t md_alg, unsigned int hashlen, const unsigned char *hash, unsigned char *sig);
int mbedtls_rsa_rsassa_pss_sign(mbedtls_rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng, int mode, mbedtls_md_type_t md_alg, unsigned int hashlen, const unsigned char *hash, unsigned char *sig);
int mbedtls_rsa_pkcs1_verify(mbedtls_rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng, int mode, mbedtls_md_type_t md_alg, unsigned int hashlen, const unsigned char *hash, const unsigned char *sig);
int mbedtls_rsa_rsassa_pkcs1_v15_verify(mbedtls_rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng, int mode, mbedtls_md_type_t md_alg, unsigned int hashlen, const unsigned char *hash, const unsigned char *sig);
int mbedtls_rsa_rsassa_pss_verify(mbedtls_rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng, int mode, mbedtls_md_type_t md_alg, unsigned int hashlen, const unsigned char *hash, const unsigned char *sig);
int mbedtls_rsa_rsassa_pss_verify_ext(mbedtls_rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng, int mode, mbedtls_md_type_t md_alg, unsigned int hashlen, const unsigned char *hash, mbedtls_md_type_t mgf1_hash_id, int expected_salt_len, const unsigned char *sig);
int mbedtls_rsa_copy(mbedtls_rsa_context *dst, const mbedtls_rsa_context *src);
void mbedtls_rsa_free(mbedtls_rsa_context *ctx);

3.2.1.3 ECP API

函数名称
const mbedtls_ecp_curve_info *mbedtls_ecp_curve_list(void);
const mbedtls_ecp_group_id *mbedtls_ecp_grp_id_list(void);
const mbedtls_ecp_curve_info *mbedtls_ecp_curve_info_from_grp_id(mbedtls_ecp_group_id grp_id);
const mbedtls_ecp_curve_info *mbedtls_ecp_curve_info_from_tls_id(uint16_t tls_id);
const mbedtls_ecp_curve_info *mbedtls_ecp_curve_info_from_name(const char *name);
void mbedtls_ecp_point_init(mbedtls_ecp_point *pt);
void mbedtls_ecp_group_init(mbedtls_ecp_group *grp);
void mbedtls_ecp_keypair_init(mbedtls_ecp_keypair *key);
void mbedtls_ecp_point_free(mbedtls_ecp_point *pt);
void mbedtls_ecp_group_free(mbedtls_ecp_group *grp);
void mbedtls_ecp_keypair_free(mbedtls_ecp_keypair *key);

int mbedtls_ecp_copy(mbedtls_ecp_point *P, const mbedtls_ecp_point *Q);
int mbedtls_ecp_group_copy(mbedtls_ecp_group *dst, const mbedtls_ecp_group *src);
int mbedtls_ecp_set_zero(mbedtls_ecp_point *pt);
int mbedtls_ecp_is_zero(mbedtls_ecp_point *pt);
int mbedtls_ecp_point_cmp(const mbedtls_ecp_point *P, const mbedtls_ecp_point *Q);
int mbedtls_ecp_point_read_string(mbedtls_ecp_point *P, int radix, const char *x, const char *y);
int mbedtls_ecp_point_write_binary(const mbedtls_ecp_group *grp, const mbedtls_ecp_point *P, int format, size_t *olen, unsigned char *buf, size_t buflen);
int mbedtls_ecp_point_read_binary(const mbedtls_ecp_group *grp, mbedtls_ecp_point *P, const unsigned char *buf, size_t ilen);
int mbedtls_ecp_tls_read_point(const mbedtls_ecp_group *grp, mbedtls_ecp_point *pt, const unsigned char **buf, size_t len);
int mbedtls_ecp_tls_write_point(const mbedtls_ecp_group *grp, const mbedtls_ecp_point *pt, int format, size_t *olen, unsigned char *buf, size_t blen);
int mbedtls_ecp_group_load(mbedtls_ecp_group *grp, mbedtls_ecp_group_id id);
int mbedtls_ecp_tls_read_group(mbedtls_ecp_group *grp, const unsigned char **buf, size_t len);
int mbedtls_ecp_tls_write_group(const mbedtls_ecp_group *grp, size_t *olen, unsigned char *buf, size_t blen);
int mbedtls_ecp_mul(mbedtls_ecp_group *grp, mbedtls_ecp_point *R, const mbedtls_mpi *m, const mbedtls_ecp_point *P, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng);
int mbedtls_ecp_muladd(mbedtls_ecp_group *grp, mbedtls_ecp_point *R, const mbedtls_mpi *m, const mbedtls_ecp_point *P, const mbedtls_mpi *n, const mbedtls_ecp_point *Q);
int mbedtls_ecp_check_pubkey(const mbedtls_ecp_group *grp, const mbedtls_ecp_point *pt);
int mbedtls_ecp_check_privkey(const mbedtls_ecp_group *grp, const mbedtls_mpi *d);
int mbedtls_ecp_gen_keypair_base(mbedtls_ecp_group *grp, const mbedtls_ecp_point *G, mbedtls_mpi *d, mbedtls_ecp_point *Q, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng);
int mbedtls_ecp_gen_keypair(mbedtls_ecp_group *grp, mbedtls_mpi *d, mbedtls_ecp_point *Q, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng);
int mbedtls_ecp_gen_key(mbedtls_ecp_group_id grp_id, mbedtls_ecp_keypair *key, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng);
int mbedtls_ecp_check_pub_priv(const mbedtls_ecp_keypair *pub, const mbedtls_ecp_keypair *prv);

3.2.1.4 ECP_CURVE API

函数名称
const mbedtls_ecp_curve_info *mbedtls_ecp_curve_list(void);
const mbedtls_ecp_group_id *mbedtls_ecp_grp_id_list(void);

const mbedtls_ecp_curve_info *mbedtls_ecp_curve_info_from_grp_id(mbedtls_ecp_group_id grp_id);
const mbedtls_ecp_curve_info *mbedtls_ecp_curve_info_from_tls_id(uint16_t tls_id);
const mbedtls_ecp_curve_info *mbedtls_ecp_curve_info_from_name(const char *name);
void mbedtls_ecp_point_init(mbedtls_ecp_point *pt);
void mbedtls_ecp_group_init(mbedtls_ecp_group *grp);
void mbedtls_ecp_keypair_init(mbedtls_ecp_keypair *key);
void mbedtls_ecp_point_free(mbedtls_ecp_point *pt);
void mbedtls_ecp_group_free(mbedtls_ecp_group *grp);
void mbedtls_ecp_keypair_free(mbedtls_ecp_keypair *key);
int mbedtls_ecp_copy(mbedtls_ecp_point *P, const mbedtls_ecp_point *Q);
int mbedtls_ecp_group_copy(mbedtls_ecp_group *dst, const mbedtls_ecp_group *src);
int mbedtls_ecp_set_zero(mbedtls_ecp_point *pt);
int mbedtls_ecp_is_zero(mbedtls_ecp_point *pt);
int mbedtls_ecp_point_cmp(const mbedtls_ecp_point *P, const mbedtls_ecp_point *Q);
int mbedtls_ecp_point_read_string(mbedtls_ecp_point *P, int radix, const char *x, const char *y);
int mbedtls_ecp_point_write_binary(const mbedtls_ecp_group *grp, const mbedtls_ecp_point *P, int format, size_t *olen, unsigned char *buf, size_t buflen);
int mbedtls_ecp_point_read_binary(const mbedtls_ecp_group *grp, mbedtls_ecp_point *P, const unsigned char *buf, size_t ilen);
int mbedtls_ecp_tls_read_point(const mbedtls_ecp_group *grp, mbedtls_ecp_point *pt, const unsigned char **buf, size_t len);
int mbedtls_ecp_tls_write_point(const mbedtls_ecp_group *grp, const mbedtls_ecp_point *pt, int format, size_t *olen, unsigned char *buf, size_t blen);
int mbedtls_ecp_group_load(mbedtls_ecp_group *grp, mbedtls_ecp_group_id id);
int mbedtls_ecp_tls_read_group(mbedtls_ecp_group *grp, const unsigned char **buf, size_t len);
int mbedtls_ecp_tls_write_group(const mbedtls_ecp_group *grp, size_t *olen, unsigned char *buf, size_t blen);
int mbedtls_ecp_mul(mbedtls_ecp_group *grp, mbedtls_ecp_point *R, const mbedtls_mpi *m, const mbedtls_ecp_point *P, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng);
int mbedtls_ecp_muladd(mbedtls_ecp_group *grp, mbedtls_ecp_point *R, const mbedtls_mpi *m, const mbedtls_ecp_point *P, const mbedtls_mpi *n, const mbedtls_ecp_point *Q);
int mbedtls_ecp_check_pubkey(const mbedtls_ecp_group *grp, const mbedtls_ecp_point *pt);
int mbedtls_ecp_check_privkey(const mbedtls_ecp_group *grp, const mbedtls_mpi *d);
int mbedtls_ecp_gen_keypair_base(mbedtls_ecp_group *grp, const mbedtls_ecp_point *G, mbedtls_mpi *d, mbedtls_ecp_point *Q, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng);
int mbedtls_ecp_gen_keypair(mbedtls_ecp_group *grp, mbedtls_mpi *d, mbedtls_ecp_point *Q, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng);
int mbedtls_ecp_gen_key(mbedtls_ecp_group_id grp_id, mbedtls_ecp_keypair *key,

int (*f_rng)(void *, unsigned char *, size_t), void *p_rng);
int mbedtls_ecp_check_pub_priv(const mbedtls_ecp_keypair *pub, const mbedtls_ecp_keypair *prv);

3.2.1.5 SHA1 API

函数名称
void mbedtls_sha1_init(mbedtls_sha1_context *ctx);
void mbedtls_sha1_set(mbedtls_sha1_context *ctx, bool is_offload)
void mbedtls_sha1_free(mbedtls_sha1_context *ctx);
void mbedtls_sha1_clone(mbedtls_sha1_context *dst, const mbedtls_sha1_context *src); 不支持 offload 功能
void mbedtls_sha1_starts(mbedtls_sha1_context *ctx);
void mbedtls_sha1_update(mbedtls_sha1_context *ctx, const unsigned char *input, size_t ilen);
void mbedtls_sha1_finish(mbedtls_sha1_context *ctx, unsigned char output[20]);
void mbedtls_sha1_process(mbedtls_sha1_context *ctx, const unsigned char data[64]);
void mbedtls_sha1(const unsigned char *input, size_t ilen, unsigned char output[20]);

3.2.1.6 SHA256 API

函数名称
void mbedtls_sha256_init(mbedtls_sha256_context *ctx);
void mbedtls_sha256_set(mbedtls_sha256_context *ctx, bool is_offload)
void mbedtls_sha256_free(mbedtls_sha256_context *ctx);
void mbedtls_sha256_clone(mbedtls_sha256_context *dst, const mbedtls_sha256_context *src); 不支持 offload 功能
void mbedtls_sha256_starts(mbedtls_sha256_context *ctx);
void mbedtls_sha256_update(mbedtls_sha256_context *ctx, const unsigned char *input, size_t ilen);
void mbedtls_sha256_finish(mbedtls_sha256_context *ctx, unsigned char output[20]);
void mbedtls_sha256_process(mbedtls_sha256_context *ctx, const unsigned char data[64]);
void mbedtls_sha256(const unsigned char *input, size_t ilen, unsigned char output[20]);

3.2.1.7 SHA512 API

函数名称
void mbedtls_sha512_init(mbedtls_sha512_context *ctx);
void mbedtls_sha512_free(mbedtls_sha512_context *ctx);
void mbedtls_sha512_clone(mbedtls_sha512_context *dst, const mbedtls_sha512_context *src);

void mbedtls_sha512_starts(mbedtls_sha512_context *ctx);
void mbedtls_sha512_update(mbedtls_sha512_context *ctx, const unsigned char *input, size_t ilen);
void mbedtls_sha512_finish(mbedtls_sha512_context *ctx, unsigned char output[20]);
void mbedtls_sha512_process(mbedtls_sha512_context *ctx, const unsigned char data[64]);
void mbedtls_sha512(const unsigned char *input, size_t ilen, unsigned char output[20]);

3.2.1.8 ENTROPY API

函数名称
void mbedtls_entropy_init(mbedtls_entropy_context *ctx);
void mbedtls_entropy_free(mbedtls_entropy_context *ctx);
int mbedtls_entropy_add_source(mbedtls_entropy_context *ctx, mbedtls_entropy_f_source_ptr f_source, void *p_source, size_t threshold, int strong);
int mbedtls_entropy_gather(mbedtls_entropy_context *ctx);
int mbedtls_entropy_func(void *data, unsigned char *output, size_t len);
int mbedtls_entropy_update_manual(mbedtls_entropy_context *ctx, const unsigned char *data, size_t len);

3.2.1.9 ENTROPY_POLL API

函数名称
int mbedtls_null_entropy_poll(void *data, unsigned char *output, size_t len, size_t *olen);
int mbedtls_hardclock_poll(void *data, unsigned char *output, size_t len, size_t *olen);
int mbedtls_hardware_poll(void *data, unsigned char *output, size_t len, size_t *olen);

3.2.1.10 BIGNUM API

函数名称
void mbedtls_mpi_init(mbedtls_mpi *X);
void mbedtls_mpi_free(mbedtls_mpi *X);
int mbedtls_mpi_grow(mbedtls_mpi *X, size_t nblimbs);
int mbedtls_mpi_shrink(mbedtls_mpi *X, size_t nblimbs);
int mbedtls_mpi_copy(mbedtls_mpi *X, const mbedtls_mpi *Y);
void mbedtls_mpi_swap(mbedtls_mpi *X, mbedtls_mpi *Y);
int mbedtls_mpi_safe_cond_assign(mbedtls_mpi *X, const mbedtls_mpi *Y, unsigned char assign);
int mbedtls_mpi_safe_cond_swap(mbedtls_mpi *X, mbedtls_mpi *Y, unsigned char assign);
int mbedtls_mpi_lset(mbedtls_mpi *X, mbedtls_mpi_sint z);
int mbedtls_mpi_get_bit(const mbedtls_mpi *X, size_t pos);

int mbedtls_mpi_set_bit(mbedtls_mpi *X, size_t pos, unsigned char val);
size_t mbedtls_mpi_lsb(const mbedtls_mpi *X);
size_t mbedtls_mpi_bitlen(const mbedtls_mpi *X);
size_t mbedtls_mpi_size(const mbedtls_mpi *X);
int mbedtls_mpi_read_string(mbedtls_mpi *X, int radix, const char *s);
int mbedtls_mpi_write_string(const mbedtls_mpi *X, int radix, char *buf, size_t buflen, size_t *olen);
int mbedtls_mpi_read_file(mbedtls_mpi *X, int radix, FILE *fin);
int mbedtls_mpi_write_file(const char *p, const mbedtls_mpi *X, int radix, FILE *fout);
int mbedtls_mpi_read_binary(mbedtls_mpi *X, const unsigned char *buf, size_t buflen);
int mbedtls_mpi_write_binary(const mbedtls_mpi *X, unsigned char *buf, size_t buflen);
int mbedtls_mpi_shift_l(mbedtls_mpi *X, size_t count);
int mbedtls_mpi_shift_r(mbedtls_mpi *X, size_t count);
int mbedtls_mpi_cmp_abs(const mbedtls_mpi *X, const mbedtls_mpi *Y);
int mbedtls_mpi_cmp_mpi(const mbedtls_mpi *X, const mbedtls_mpi *Y);
int mbedtls_mpi_cmp_int(const mbedtls_mpi *X, mbedtls_mpi_sint z);
int mbedtls_mpi_add_abs(mbedtls_mpi *X, const mbedtls_mpi *A, const mbedtls_mpi *B);
int mbedtls_mpi_sub_abs(mbedtls_mpi *X, const mbedtls_mpi *A, const mbedtls_mpi *B);
int mbedtls_mpi_add_mpi(mbedtls_mpi *X, const mbedtls_mpi *A, const mbedtls_mpi *B);
int mbedtls_mpi_sub_mpi(mbedtls_mpi *X, const mbedtls_mpi *A, const mbedtls_mpi *B);
int mbedtls_mpi_add_int(mbedtls_mpi *X, const mbedtls_mpi *A, mbedtls_mpi_sint b);
int mbedtls_mpi_sub_int(mbedtls_mpi *X, const mbedtls_mpi *A, mbedtls_mpi_sint b);
int mbedtls_mpi_mul_mpi(mbedtls_mpi *X, const mbedtls_mpi *A, const mbedtls_mpi *B);
int mbedtls_mpi_mul_int(mbedtls_mpi *X, const mbedtls_mpi *A, mbedtls_mpi_uint b);
int mbedtls_mpi_div_mpi(mbedtls_mpi *Q, mbedtls_mpi *R, const mbedtls_mpi *A, const mbedtls_mpi *B);
int mbedtls_mpi_div_int(mbedtls_mpi *Q, mbedtls_mpi *R, const mbedtls_mpi *A, mbedtls_mpi_sint b);
int mbedtls_mpi_mod_mpi(mbedtls_mpi *R, const mbedtls_mpi *A, const mbedtls_mpi *B);
int mbedtls_mpi_mod_int(mbedtls_mpi_uint *r, const mbedtls_mpi *A, mbedtls_mpi_sint b);
int mbedtls_mpi_exp_mod(mbedtls_mpi *X, const mbedtls_mpi *A, const mbedtls_mpi *E, const mbedtls_mpi *N, mbedtls_mpi * _RR);
int mbedtls_mpi_fill_random(mbedtls_mpi *X, size_t size, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng);
int mbedtls_mpi_gcd(mbedtls_mpi *G, const mbedtls_mpi *A, const mbedtls_mpi *B);
int mbedtls_mpi_inv_mod(mbedtls_mpi *X, const mbedtls_mpi *A, const mbedtls_mpi *N);
int mbedtls_mpi_is_prime(const mbedtls_mpi *X, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng);
int mbedtls_mpi_gen_prime(mbedtls_mpi *X, size_t nbits, int dh_flag, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng);

3.2.1.11 SM2 API

对于 SM2 国密算法，mbed TLS 中并没有支持。为了开发者方便统一使用，借鉴了 mbed TLS 中 RSA 模块 API 样式扩展了 SM2 算法 API，详细介绍如下。

原型	int mbedtls_sm2_genkey(mbedtls_ecp_group *grp, mbedtls_mpi *d, mbedtls_ecp_point *Q);
说明	执行 SM2 密钥生成操作
输入参数	grp: 指向 ECP 椭圆曲线的指针
输出参数 1	d: 指向秘钥的指针
输出参数 2	Q: 指向公钥的指针
返回值	0 : 成功 其他 ： 失败

原型	int mbedtls_sm2_e_get(mbedtls_ecp_group *grp, const unsigned char *m, size_t mlen, const unsigned char *z, size_t zlen, unsigned char *e, size_t elen);
说明	获取 SM2 算法 e 值
输入参数 1	grp: 指向 ECP 椭圆曲线的指针
输入参数 2	m: 指向消息 m 的指针
输入参数 3	mlen: m 的长度，单位：byte
输入参数 4	z: 指向 z 值的指针，z 值长度为 32bytes
输入参数 5	elen: e 的长度，单位：byte
输出参数	e: 指向运算结果 e 值的指针，e 值长度 32bytes
返回值	0 : 成功 其他 ： 失败

原型	int mbedtls_sm2_z_get(mbedtls_ecp_group *grp, mbedtls_ecp_point *Q, const unsigned char *id, size_t ilen, unsigned char *z, size_t zlen);
说明	获取 SM2 算法 z 值
输入参数 1	grp: 指向 ECP 椭圆曲线的指针
输入参数 2	Q: 指向公钥的指针
输入参数 3	id: 指向用户 ID 的指针
输入参数 4	ilen: 用户 ID 长度，单位：byte，参数取值： 0 至 8192
输入参数 5	zlen: z 的长度，单位：byte
输出参数	z: 指向运算结果 z 值的指针，z 值长度 32bytes
返回值	0 : 成功 其他 ： 失败

原型	int mbedtls_sm2_encrypt(mbedtls_ecp_group *grp, mbedtls_ecp_point *Q,

	const unsigned char *m, size_t mlen, unsigned char *buf, size_t *blen);
说明	执行 SM2 加密操作
输入参数 1	grp: 指向 ECP 椭圆曲线的指针
输入参数 2	Q: 指向公钥的指针
输入参数 3	m: 指向明文数据的指针
输入参数 4	mlen : 明文数据长度，单位：byte
输出参数 1	buf: 指向密文数据的指针
输出参数 2	blen: 密文数据长度，单位：byte，通常为 mlen + 96bytes
返回值	0 : 成功 其他 ： 失败

原型	int mbedtls_sm2_decrypt(mbedtls_ecp_group *grp, mbedtls_mpi *d, const unsigned char *buf, size_t blen, unsigned char *m, size_t *mlen);
说明	执行 SM2 解密操作
输入参数 1	grp： 指向 ECP 椭圆曲线的指针
输入参数 2	d： 指向秘钥的指针
输入参数 3	buf： 指向密文数据的指针
输入参数 4	blen： 密文数据长度，单位：byte 注： 密文数据需大于 96byte
输出参数 1	m： 指向明文数据的指针 注： 密文数据缓冲区需大于 m_nbytes + 96byte
输出参数 2	mlen： 明文数据长度，单位：byte，通常为 m_nbytes + 96bytes
返回值	0 : 成功 其他 ： 失败

原型	int mbedtls_ecdh_sm2_gen_public(mbedtls_ecp_group *grp, mbedtls_mpi *d, mbedtls_ecp_point *Q, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng);
说明	执行 SM2 生成公钥操作
输入参数 1	grp： 指向 ECP 椭圆曲线的指针
输入参数 2	f_rng： 指向随机数函数的指针
输入参数 3	p_rng： 指向随机数函数参数的指针
输出参数 1	d： 指向秘钥的指针
输出参数 2	Q: 指向公钥的指针
返回值	0 : 成功 其他 ： 失败

原型	void mbedtls_ecdh_sm2_init(mbedtls_ecdh_sm2_context *ctx);
说明	初始化 SM2 的秘钥交换上下文

输入参数	ctx: 指向 SM2 秘钥交换上下文的指针
输出参数	无
返回值	无

原型	void mbedtls_ecdh_sm2_free(mbedtls_ecdh_sm2_context *ctx);
说明	释放 SM2 的秘钥交换上下文
输入参数	ctx: 指向 SM2 秘钥交换上下文的指针
输出参数	无
返回值	无

原型	int mbedtls_ecdh_sm2_make_params(mbedtls_ecdh_sm2_context *ctx, size_t *olen, unsigned char *buf, size_t blen, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng);
说明	为 SM2 进行秘钥交换准备参数:选定并输出椭圆曲线参数和公钥
输入参数 1	ctx: 指向 SM2 秘钥交换上下文的指针
输入参数 2	blen: 椭圆曲线参数 buf 长度，单位 byte
输入参数 3	f_rng: 指向随机数函数的指针
输入参数 4	p_rng: 指向随机数函数参数的指针
输出参数 1	olen: 指向输出公钥数据的指针
输出参数 2	buf: 指向输出椭圆曲线参数的指针
返回值	0 : 成功 其他 ： 失败

原型	int mbedtls_ecdh_sm2_make_params2(mbedtls_ecdh_sm2_context *ctx, unsigned char *buf, size_t blen);
说明	为 SM2 进行秘钥交换准备输出 S1、SA 的值
输入参数 1	ctx: 指向 SM2 秘钥交换上下文的指针
输入参数 2	blen: 输出缓存 buf 的长度,单位 byte
输出参数	buf: 指向输出缓存 buf=S1+SA 的指针
返回值	0 : 成功 其他 ： 失败

原型	int mbedtls_ecdh_sm2_read_params(mbedtls_ecdh_sm2_context *ctx, const unsigned char **buf, const unsigned char *end);
说明	为 SM2 进行秘钥交换读入密钥交换参数:椭圆曲线参数和公钥
输入参数 1	ctx: 指向 SM2 秘钥交换上下文的指针
输入参数 2	buf: 指向输入椭圆曲线及公钥数据指针的指针
输入参数 3	end: 指向输出缓存末尾的指针
输出参数	无
返回值	0 : 成功 其他 ： 失败

原型	int mbedtls_ecdh_sm2_get_params(mbedtls_ecdh_sm2_context *ctx, const mbedtls_ecp_keypair *key, mbedtls_ecdh_sm2_side side);
说明	为 SM2 进行秘钥交换获取临时秘钥信息 注: 当为己方时,获取临时秘钥和公钥信息; 当为对方时,获取临时公钥信息,
输入参数 1	ctx: 指向 SM2 秘钥交换上下文的指针
输入参数 2	key: 指向椭圆曲线密钥对结构体的指针
输入参数 3	side: SM2 密钥交换中己方或者对方,参数选择: MBEDTLS_ECDH_SM2_OURS MBEDTLS_ECDH_SM2_THEIRS
输出参数	无
返回值	0 : 成功 其他 : 失败

原型	int mbedtls_ecdh_sm2_get_params2(mbedtls_ecdh_sm2_context *ctx, const mbedtls_ecp_keypair *key, mbedtls_ecdh_sm2_side side);
说明	为 SM2 进行秘钥交换获取固定秘钥信息 注: 当为己方时,获取固定秘钥信息; 当为对方时,获取固定公钥信息,
输入参数 1	ctx: 指向 SM2 秘钥交换上下文的指针
输入参数 2	key: 指向圆曲线密钥对结构体的指针
输入参数 3	side: SM2 密钥交换中己方或者对方,参数选择: MBEDTLS_ECDH_SM2_OURS MBEDTLS_ECDH_SM2_THEIRS
输出参数	无
返回值	0 : 成功 其他 : 失败

原型	int mbedtls_ecdh_sm2_get_params3(mbedtls_ecdh_sm2_context *ctx, const mbedtls_ecp_keypair *key, const unsigned char *id, size_t ilen, mbedtls_ecdh_sm2_side side);
说明	为 SM2 进行秘钥交换获取己方或者对方 Z 值信息
输入参数 1	ctx: 指向 SM2 秘钥交换上下文的指针
输入参数 2	key: 指向 SM2 密钥对结构体的指针
输入参数 3	id: 指向 SM2 用户 ID 的指针
输入参数 4	ilen: 用户 ID 的长度,单位 byte
输入参数 5	side: SM2 密钥交换中己方或者对方,参数选择: MBEDTLS_ECDH_SM2_OURS

	MBEDTLS_ECDH_SM2_THEIRS
输出参数	无
返回值	0： 成功 其他 ： 失败

原型	int mbedtls_ecdh_sm2_get_params4(mbedtls_ecdh_sm2_context *ctx, size_t zlen, mbedtls_ecdh_sm2_side side);
说明	为 SM2 进行秘钥交换设置己方或对方信息,和使用的共享秘钥 z 长度
输入参数 1	ctx: 指向 SM2 秘钥交换上下文的指针
输入参数 2	zlen: 使用共享秘钥 z 长度,单位:byte
输入参数 3	side: SM2 密钥交换中己方或者对方,参数选择: MBEDTLS_ECDH_SM2_OURS MBEDTLS_ECDH_SM2_THEIRS
输出参数	无
返回值	0： 成功 其他 ： 失败

原型	int mbedtls_ecdh_sm2_make_public(mbedtls_ecdh_sm2_context *ctx, size_t *olen,unsigned char *buf, size_t blen, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng);
说明	为 SM2 进行秘钥交换导出己方公钥值
输入参数 1	ctx: 指向 SM2 秘钥交换上下文的指针
输入参数 2	blen: 输出缓存的长度,单位:byte
输入参数 3	f_rng: 指向随机数函数的指针
输入参数 4	p_rng: 指向随机数函数参数的指针
输出参数 1	olen: 指向实际输出字节数量的指针
输出参数 2	buf: 指向输出缓存己方公钥数据的指针
返回值	0： 成功 其他 ： 失败

原型	int mbedtls_ecdh_sm2_read_public(mbedtls_ecdh_sm2_context *ctx, const unsigned char *buf, size_t blen);
说明	为 SM2 进行秘钥交换导入对方公钥值
输入参数 1	ctx: 指向 SM2 秘钥交换上下文的指针
输入参数 1	buf: 指向输入缓存公钥数据的指针
输入参数 2	blen: 输入缓存的长度,单位 byte
输出参数	无
返回值	0： 成功 其他 ： 失败

原型	int mbedtls_ecdh_sm2_calc_secret(mbedtls_ecdh_sm2_context *ctx, size_t *olen, unsigned char *buf, size_t blen, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng);
说明	执行 SM2 密钥交换计算,输出计算后的共享秘钥
输入参数 1	ctx: 指向 SM2 秘钥交换上下文的指针
输入参数 2	blen: 输出缓存的长度,单位:byte
输入参数 3	f_rng: 指向随机数函数的指针
输入参数 4	p_rng: 指向随机数函数参数的指针
输出参数 1	olen: 指向输出共享秘钥数据实际长度的指针
输出参数 2	buf: 指向输出共享秘钥数据的指针
返回值	0 : 成功 其他 : 失败

原型	int mbedtls_ecdsa_sm2_sign(mbedtls_ecp_group *grp, mbedtls_mpi *r, mbedtls_mpi *s, const mbedtls_mpi *d, const unsigned char *buf, size_t blen, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng);
说明	SM2 签名计算
输入参数 1	grp: 指向 ECP 椭圆曲线的指针
输入参数 2	d: 指向私钥的指针
输入参数 3	buf: 指向计算 hash 值后消息数据的指针
输入参数 4	blen: buf 的长度,单位:byte
输入参数 5	f_rng: 指向随机数函数的指针
输入参数 6	p_rng: 指向随机数函数参数的指针
输出参数 1	r: 指向第一个输出值 r 的指针
输出参数 2	s: 指向第二个输出值 d 的指针
返回值	0 : 成功 其他 : 失败

原型	int mbedtls_ecdsa_sm2_sign_det(mbedtls_ecp_group *grp, mbedtls_mpi *r, mbedtls_mpi *s, const mbedtls_mpi *d, const unsigned char *buf, size_t blen, mbedtls_md_type_t md_alg);
说明	不使用随机数来进行 SM2 签名计算
输入参数 1	grp: 指向 ECP 曲线参数的指针
输入参数 2	d: 指向私钥的指针
输入参数 3	buf: 指向计算 hash 值后消息数据的指针
输入参数 4	blen: buf 的长度,单位:byte
输入参数 5	md_alg: 使用的 hash 算法类型
输出参数 1	r: 指向第一个输出值 r 的指针

输出参数 2	s: 指向第二个输出值 d 的指针
返回值	0 : 成功 其他 : 失败

原型	int mbedtls_ecdsa_sm2_verify(mbedtls_ecp_group *grp, const unsigned char *buf, size_t blen, const mbedtls_ecp_point *Q, const mbedtls_mpi *r, const mbedtls_mpi *s);
说明	SM2 验签
输入参数 1	grp: 指向 ECP 曲线参数的指针
输入参数 2	buf: 指向计算 hash 值后消息数据的指针
输入参数 3	blen: buf 的长度,单位:byte
输入参数 4	Q: 指向公钥的指针
输入参数 5	r: 指向第一个输入值 r 的指针
输入参数 6	s: 指向第二个输入值 d 的指针
返回值	0 : 成功 其他 : 失败

原型	int mbedtls_ecdsa_sm2_write_signature(mbedtls_ecdsa_sm2_context *ctx, mbedtls_md_type_t md_alg, const unsigned char *hash, size_t hlen, unsigned char *sig, size_t *slen, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng);
说明	计算 SM2 签名并输出
输入参数 1	ctx: 指向 SM2 DSA 上下文的指针
输入参数 2	md_alg: hash 消息算法类型
输入参数 3	hash: 指向计算 hash 值后消息数据的指针
输入参数 4	hlen: hash 后消息值长度,单位:byte
输入参数 5	f_rng: 指向随机数函数的指针
输入参数 6	p_rng: 指向随机数函数参数的指针
输出参数 1	sig: 指向输出签名数据的指针
输出参数 2	slen: 指向输出签名数据长度的指针
返回值	0 : 成功 其他 : 失败

原型	int mbedtls_ecdsa_sm2_write_signature_det(mbedtls_ecdsa_sm2_context *ctx, const unsigned char *hash, size_t hlen, unsigned char *sig, size_t *slen, mbedtls_md_type_t md_alg)
说明	不使用随机数来计算 SM2 签名并输出
输入参数 1	ctx: 指向 SM2 DSA 上下文的指针
输入参数 2	hash: 指向计算 hash 值后消息数据的指针

输入参数 3	hlen: hash 后消息值长度,单位:byte
输入参数 4	md_alg: hash 消息算法类型
输出参数 1	sig: 指向输出签名数据的指针
输出参数 2	slen: 指向输出签名数据长度的指针
返回值	0 : 成功 其他 : 失败

原型	int mbedtls_ecdsa_sm2_read_signature(mbedtls_ecdsa_sm2_context *ctx, const unsigned char *hash, size_t hlen, const unsigned char *sig, size_t slen);
说明	读取并验证 SM2 签名正确性
输入参数 1	ctx: 指向 SM2 DSA 上下文的指针
输入参数 2	hash: 指向计算 hash 值后消息数据的指针
输入参数 3	hlen: hash 后消息值长度,单位:byte
输入参数 4	sig: 指向输出签名数据的指针
输入参数 5	slen: 指向输出签名数据长度的指针
输出参数	无
返回值	0 : 成功 其他 : 失败

原型	int mbedtls_ecdsa_sm2_genkey(mbedtls_ecdsa_sm2_context *ctx, mbedtls_ecp_group_id gid, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng);
说明	执行 SM2 密钥生成操作
输入参数 1	ctx: 指向 SM2 DSA 上下文的指针
输入参数 2	gid: 椭圆曲线 ID 编号
输入参数 3	f_rng: 指向随机数函数的指针
输入参数 4	p_rng: 指向随机数函数参数的指针
输出参数	无
返回值	0 : 成功 其他 : 失败

原型	int mbedtls_ecdsa_sm2_from_keypair(mbedtls_ecdsa_sm2_context *ctx, const mbedtls_ecp_keypair *key);
说明	通过密钥对生成 SM2 DSA 上下文
输入参数 1	ctx: 指向 SM2 DSA 上下文的指针
输入参数 2	key: 指向密钥对的指针
输出参数	无
返回值	0 : 成功 其他 : 失败

原型	void mbedtls_ecdsa_sm2_init(mbedtls_ecdsa_sm2_context *ctx);
说明	初始化 SM2 DSA 使用的上下文

输入参数	ctx: 指向 SM2 DSA 上下文的指针
输出参数	无
返回值	无

原型	void mbedtls_ecdsa_sm2_free(mbedtls_ecdsa_sm2_context *ctx)
说明	释放 SM2 DSA 使用的上下文
输入参数	ctx: 指向 SM2 DSA 上下文的指针
输出参数	无
返回值	无

3.2.1.12 SM3 API

对于 SM3 国密消息摘要算法，mbed TLS 中也没有支持。为了开发者方便统一使用，借鉴了 mbed TLS 中 SHA256 模块 API 样式扩展了 SM3 算法 API，详细介绍如下。

原型	void mbedtls_sm3_init(mbedtls_sm3_context *ctx);
说明	初始化 SM3 上下文
输入参数	ctx： 指向 SM3 上下文的指针
输出参数	无
返回值	无

原型	void mbedtls_sm3_free(mbedtls_sm3_context *ctx);
说明	释放 SM3 上下文
输入参数	ctx： 指向 SM3 上下文的指针
输出参数	无
返回值	无

原型	void mbedtls_sm3_clone(mbedtls_sm3_context *dst, const mbedtls_sm3_context *src);
说明	复制 SM3 上下文 注：不支持硬件加速 offload 功能
输出参数	dst：指向目标 SM3 上下文的指针
输入参数	src：指向源 SM3 上下文的指针
返回值	无

原型	void mbedtls_sm3_starts(mbedtls_sm3_context *ctx);
说明	开始初始化 SM3 算法
输入参数	ctx： 指向 SM3 上下文的指针
输出参数	无

返回值	无

原型	void mbedtls_sm3_update(mbedtls_sm3_context *ctx, const unsigned char *input, size_t ilen);
说明	执行 SM3 算法数据输入操作
输入参数 1	ctx： 指向 SM3 上下文的指针
输入参数 2	input： 数据输入缓冲区位置指针，指向缓冲区起始位置
输入参数 3	ilen： 输入数据长度，单位：byte
输出参数	无
返回值	无

原型	void mbedtls_sm3_finish(mbedtls_sm3_context *ctx, unsigned char output[32]);
说明	执行 SM3 算法数据输出操作
输入参数	ctx： 指向 SM3 上下文的指针
输出参数	output： 数据输出缓冲区位置指针，指向缓冲区起始位置
返回值	无

原型	void mbedtls_sm3(const unsigned char *input, size_t ilen, unsigned char output[32]);
说明	执行一次完整的 SM3 算法
输入参数 1	input： 数据输入缓冲区位置指针，指向缓冲区起始位置
输入参数 2	ilen： 输入数据长度，单位：byte
输出参数	output： 数据输出缓冲区位置指针，指向缓冲区起始位置
返回值	无

3.2.1.13 SM4 API

对于 SM4 国密对称加密算法，mbed TLS 中也没有支持。为了开发者方便统一使用，借鉴了 mbed TLS 中 AES 模块 API 样式扩展了 SM4 算法 API，详细介绍如下。

原型	void mbedtls_sm4_init(mbedtls_sm4_context *ctx);
说明	初始化 SM4 上下文
输入参数	ctx： 指向 SM4 上下文的指针
输出参数	无
返回值	无

原型	void mbedtls_sm4_free(mbedtls_sm4_context *ctx);
说明	释放 SM4 上下文
输入参数	ctx： 指向 SM4 上下文的指针

输出参数	无
返回值	无

原型	int mbedtls_sm4_setkey_enc(mbedtls_sm4_context *ctx, const unsigned char *key, unsigned int keybits);
说明	设置 SM4 加密使用的密钥
输入参数 1	ctx: 指向 SM4 上下文的指针
输入参数 2	key: 指向 SM4 秘钥的指针
输入参数 3	keybits: 密钥长度,单位 bit
输出参数	无
返回值	0： 成功 其他： 失败

原型	int mbedtls_sm4_setkey_dec(mbedtls_sm4_context *ctx, const unsigned char *key, unsigned int keybits);
说明	设置 SM4 解密使用的密钥
输入参数 1	ctx: 指向 SM4 上下文的指针
输入参数 2	key: 指向 SM4 秘钥的指针
输入参数 3	keybits: 密钥长度,单位 bit
输出参数	无
返回值	0： 成功 其他： 失败

原型	int mbedtls_sm4_crypt_ecb(mbedtls_sm4_context *ctx, int mode, const unsigned char input[16], unsigned char output[16]);
说明	设置使用 SM4 ECB 模式进行加解密,每次为 16byte
输入参数 1	ctx: 指向 SM4 上下文的指针
输入参数 2	mode: 算法模式选择,参数选择: MBEDTLS_SM4_ENCRYPT MBEDTLS_SM4_DECRYPT
输入参数 3	input: 数据输入缓冲区位置指针，指向缓冲区起始位置,长度为 16byte
输出参数	output: 数据输出缓冲区位置指针，指向缓冲区起始位置，长度为 16byte
返回值	0： 成功 其他： 失败

原型	int mbedtls_sm4_crypt_cbc(mbedtls_sm4_context *ctx, int mode, size_t length, unsigned char iv[16],

	const unsigned char *input, unsigned char *output);
说明	设置使用 SM4 CBC 模式进行加解密，长度必须为 16byte 的倍数
输入参数 1	ctx: 指向 SM4 上下文的指针
输入参数 2	mode: 算法模式选择,参数选择: MBEDTLS_SM4_ENCRYPT MBEDTLS_SM4_DECRYPT
输入参数 3	length: 输入数据长度,单位:byte
输入参数 4	input: 数据输入缓冲区位置指针，指向缓冲区起始位置
输入/输出参数	iv: 指向初始化向量的指针,长度为 16byte 注:使用后会被更新
输出参数	output: 数据输出缓冲区位置指针，指向缓冲区起始位置
返回值	0： 成功 其他： 失败

原型	int mbedtls_sm4_crypt_cfb128(mbedtls_sm4_context *ctx, int mode, size_t length, size_t *iv_off, unsigned char iv[16], const unsigned char *input, unsigned char *output);
说明	设置使用 SM4 CFB 128 模式进行加解密，长度必须为 16 的倍数
输入参数 1	ctx: 指向 SM4 上下文的指针
输入参数 2	mode: 算法模式选择,参数选择: MBEDTLS_SM4_ENCRYPT MBEDTLS_SM4_DECRYPT
输入参数 3	length: 输入数据长度,单位:byte
输入参数 4	input: 数据输入缓冲区位置指针，指向缓冲区起始位置
输入/输出参数 1	iv_off: 指向 iv 内部的偏移位置 注:使用后会被更新
输入/输出参数 2	iv: 指向初始化向量的指针,长度为 16byte 注:使用后会被更新
输出参数	output: 数据输出缓冲区位置指针，指向缓冲区起始位置
返回值	0： 成功 其他： 失败

原型	int mbedtls_sm4_crypt_cfb8(mbedtls_sm4_context *ctx, int mode, size_t length, unsigned char iv[16], const unsigned char *input, unsigned char *output);
说明	设置使用 SM4 CFB 8 模式进行加解密，长度必须为 16 的倍数
输入参数 1	ctx: 指向 SM4 上下文的指针
输入参数 2	mode: 算法模式选择,参数选择: MBEDTLS_SM4_ENCRYPT

	MBEDTLS_SM4_DECRYPT
输入参数 3	length: 输入数据长度,单位:byte
输入参数 4	input: 数据输入缓冲区位置指针，指向缓冲区起始位置
输入/输出参数 1	iv: 指向初始化向量的指针,长度为 16byte 注:使用后会被更新
输出参数	output: 数据输出缓冲区位置指针，指向缓冲区起始位置
返回值	0： 成功 其他： 失败

原型	int mbedtls_sm4_crypt_ofb(mbedtls_sm4_context *ctx, int mode, size_t length, unsigned char iv[16], const unsigned char *input, unsigned char *output);
说明	设置使用 SM4 OFB 模式进行加解密，长度必须为 16 的倍数
输入参数 1	ctx: 指向 SM4 上下文的指针
输入参数 2	mode: 算法模式选择,参数选择: MBEDTLS_SM4_ENCRYPT MBEDTLS_SM4_DECRYPT
输入参数 3	length: 输入数据长度,单位:byte
输入参数 4	input: 数据输入缓冲区位置指针，指向缓冲区起始位置
输入参数 5	iv: 指向初始化向量的指针,长度为 16byte
输出参数	output: 数据输出缓冲区位置指针，指向缓冲区起始位置
返回值	0： 成功 其他： 失败

原型	int mbedtls_sm4_crypt_ctr(mbedtls_sm4_context *ctx, size_t length, size_t *nc_off, unsigned char nonce_counter[16], unsigned char stream_block[16], const unsigned char *input, unsigned char *output);
说明	设置使用 SM4 CTR 模式进行加解密，长度必须为 16 的倍数
输入参数 1	ctx: 指向 SM4 上下文的指针
输入参数 2	length: 输入数据长度,单位:byte
输入参数 3	nc_of: 指向当前数据流块的偏移地址
输入参数 4	nonce_counter : 128 比特的 nonce 计数值,长度为 16byte
输入参数 5	input: 数据输入缓冲区位置指针，指向缓冲区起始位置
输入/输出参数	stream_block: 指向当前操作的数据流块,长度为 16byte 注:会被更新
输出参数	output: 数据输出缓冲区位置指针，指向缓冲区起始位置
返回值	0： 成功 其他： 失败

3.2.2 Http Client API

本软件包提供了基于 http 协议的 web 客户端功能，它能够为设备提供与 HTTP Server 的通讯的基本功能。详请参见 RT-Thread 官网文档 https://www.rt-thread.org/document/site/submodules/webclient/docs/api/

主要函数 API 列表:

函数名称
struct webclient_session *webclient_session_create(size_t header_sz)
int webclient_close(struct webclient_session *session)
int webclient_get(struct webclient_session *session, const char *URI)
int webclient_get_position(struct webclient_session *session, const char *URI, int position);
int webclient_post(struct webclient_session *session, const char *URI, const char *post_data)
int webclient_write(struct webclient_session *session, const unsigned char *buffer, size_t size)
int webclient_read(struct webclient_session *session, unsigned char *buffer, size_t size)
int webclient_set_timeout(struct webclient_session *session, int millisecond)
int webclient_header_fields_add(struct webclient_session *session, const char *fmt, ...)
const char *webclient_header_fields_get(struct webclient_session *session, const char *fields)
int webclient_response(struct webclient_session *session, unsigned char **response)
int webclient_request(const char *URI, const char *header, const char *post_data, unsigned char **response)
int webclient_resp_status_get(struct webclient_session *session)
int webclient_content_length_get(struct webclient_session *session)
int webclient_get_file(const char *URI, const char *filename)
int webclient_post_file(const char *URI, const char *filename, const char *form_data);

3.2.3 Http Server API

本软件包提供了基于 http 协议的 web 服务器功能，可以向用户提供 web 页面访问。详请参见 RT-Thread 官网文档 https://www.rt-thread.org/document/site/submodules/webnet/docs/api/

主要函数 API 列表:

函数名称
int webnet_init(void)
void webnet_set_port(int port)
int webnet_get_port(void)
void webnet_set_root(const char* webroot_path)

const char* webnet_get_root(void)
const char* mime_get_type(const char* url)
void webnet_asp_add_var(const char* name, void (*handler)(struct webnet_session* session))
void webnet_cgi_register(const char* name, void (*handler)(struct webnet_session* session));
void webnet_cgi_set_root(const char* root)
void webnet_auth_set(const char* path, const char* username_password)
void webnet_alias_set(char* old_path, char* new_path)
void webnet_session_set_header(struct webnet_session* session,const char* mimetype, int code, const char* title, int length)
int webnet_session_write(struct webnet_session* session, const rt_uint8_t* data, rt_size_t size)
void webnet_session_printf(struct webnet_session* session, const char* fmt, ...)
const char* webnet_upload_get_filename(struct webnet_session* session)
const char* webnet_upload_get_content_type(struct webnet_session* session)
const char* webnet_upload_get_nameentry(struct webnet_session* session, const char* name)
const void* webnet_upload_get_userdata(struct webnet_session* session)

3.2.4 Utest 测试框架 API

为了实现测试代码格式的统一，utest 框架提供了一套通用的测试程序编程接口。具体使用详见官网 https://www.rt-thread.org/document/site/programming-manual/utest/utest/。

函数名称
uassert_true(value)
uassert_not_null(value)
uassert_int_equal(a, b)
uassert_in_range(value, min, max)
UTEST_UNIT_RUN(test_unit_func)
UTEST_TC_EXPORT(testcase, name, init, cleanup, timeout)
void utest_log_lv_set(rt_uint8_t lv);

4 开发指南

4.1 一般注意事项

4.1.1 工程配置

4.1.1.1 修改 rtconfig.h 文件

本 SDK 选用的 RT-Thread 是一个高度可配置的嵌入式实时操作系统。该 RTOS 可以通过修改 app 目录下面的 rtconfig.h 中的对应宏定义进行组件配置。配置 rtconfig.h 文件后，需要对 CDK 工程引用的源代码进行相应修改。具体请参考《CDK IDE 用户手册》 。

例如，若要通过动态方式创建对象则需要在 rtconfig.h 文件里开启 RT_USING_HEAP 宏定义。

```
#ifndef RT_CONFIG_H__
#define RT_CONFIG_H__
......
#define RT_USING_MEMHEAP
......
#endif
```

这种进行工程配置的方式要求用户对工程依赖关系十分了解。于是，为了方便用户配置工程，RT-Thread基于scons构建脚本开发了env工具进行工程管理配置。具体使用请参考章节4.1.1.2。

4.1.1.2 使用 env 配置工具

Env 是 RT-Thread 推出的开发辅助工具，针对基于 RT-Thread 操作系统的项目工程，提供编译构建环境、图形化系统配置及软件包管理功能。其内置的 menuconfig 提供了简单易用的配置剪裁工具，可对内核、组件和扩展软件包进行自由裁剪，使系统以搭积木的方式进行构建。详细了解请参考官网文档 https://www.rt-thread.org/document/site/programming-manual/env/env/。

通过 env 工具配置生成 CDK 项目工程文件，可以通过下面命令行进行：

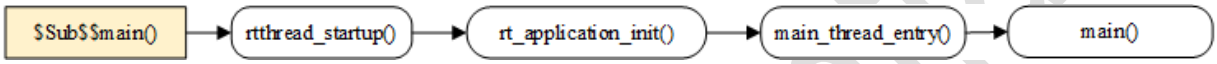
命令	说明
scons --genconfig	根据现有 rtconfig.h 文件生成 menuconfig 使用的.config 配置信息
menuconfig	图形化配置方式,按需选择配置选项
scons --target=cdk -s	根据生成的配置信息,生成 CDK 工程文件

当 CDK 工程文件生成后，编译有如下两种方式:
（1） Env 工具命令行方式使用 scons 命令编译;
（2） 打开 CDK IDE，进行可视化编译;

4.1.2 用户代码入口

4.1.2.1 main 函数入口

在系统启动时，系统会创建 main 线程，它的入口函数为 main_thread_entry()，用户的应用入口函数 main() 就是从这里真正开始的，系统调度器启动后，main 线程就开始运行，过程如下图，用户可以在 main() 函数里添加自己的应用程序初始化代码。



4.1.2.2 命令行入口

RT-Thread 提供了一套供用户在命令行调用的操作接口，主要用于调试或查看系统信息。用户可以使用自定义宏 MSH_CMD_EXPORT(name, desc)导出命令到 FinSH 终端中。这种加载运行方式使用 FinSH 来承载了函数入口的功能。

但是需要注意的是，这时运行上下文为 FinSH 线程本身，需要特别关注 FinSH 线程的栈空间。请参考 4.3 章节中关于 FinSH 控制台的使用。

4.1.3 工程示例使用

本 SDK 开发包中默认配置了扩展软件包，CDK 工程中包含了部分示例程序，并使用 MSH_CMD_EXPORT 宏导出到命令行。用户可以在命令行控制台使用 help 命令查看当前导出的示例程序列表。

另外，还有一些简单的示例程序代码（example 目录）供参考。当运行时，添加到 CDK 工程构建目录中编译运行即可。 请参考 4.1.2 用户代码入口章节。

扩展软件包示例	路径名	说明
mbed TLS 示例程序	rt-thread/packages/mbedtls-2.6.0/samples/	示例了 crypto 和 tls client 使用，其中 tls_client 支持了***国密***和***国际*** SSL 的选择使用
http client 示例程序	rt-thread/packages/webclient-2.0.1/samples/	示例了 http client get/post 基本功能
http server 示例程序	rt-thread/packages/ webnet-2.0.0/samples/	示例了 http server 基本功能
example 示例程序	rt-thread/examples	暂无

扩展软件包具体配置使用流程，请参考 RT-Thread 官网文档 https://www.rt-thread.org/document/site/#中“软件包”章节。

4.1.4 Backtrace 使用

当想查看线程调用栈信息时，就可以使用 backtrace 来实现。Backtrace 会跟踪打印线程当前实时调用栈信息，方便用户进行调试问题。

RT-Thread 内核配置中使用宏 RT_DEBUG_BACKTRACE 来开启 backtrace 功能。但是 backtrace 的使用需要编译选项中加入-mbacktrace 来支持。有如下两种方式来使用 backtrace 功能。

（1） 当前线程调用 rt_backtrace_current 函数

当程序遇到断言错误或者地址未对齐访问等异常时，会自动调用该函数来打印当前调用栈信息。

（2） 命令行调用 rt_backtrace 函数

在 FinSH 控制台中使用 rt_backtrace 命令来打印指定线程的调用栈信息。

原型	void rt_backtrace(int argc, char **argv)
说明	在编译选项中加入-mbacktrace 后，可以使用 rt_backtrace 命令（MSH）获得某一线程的函数调用信息，
输入参数 1	argv 对应的命令字符串中的参数个数 + 1
输入参数 2	命令行参数字符串指针数组指针
输出参数	无
返回值	无

原型	void rt_backtrace_current(void)
说明	在编译选项中加入-mbacktrace 后，可以使用 rt_backtrace_current 函数获得当前线程的函数调用信息。
输入参数	无
输出参数	无
返回值	无

命令行使用示例：

```
- RT -     Thread Operating System
 / | \     3.1.0 build Jun 26 2019
 2006 - 2018 Copyright by rt-thread team
RT-Thread test main entry
msh />
msh />
msh />
msh />
msh />
msh />spi_sample
msh />
msh />list_thread
thread pri  status      sp     stack size max used left tick  er
ror
------ ---  ------- ---------- ----------  ------  ---------- --
-
spi2     2  suspend 0x000000b4 0x00000400    17%   0x0000000a 00
0
spi1     2  suspend 0x000000b4 0x00000400    17%   0x0000000a 00
0
tshell  20  ready   0x00000070 0x00001000    12%   0x00000001 00
0
tidle   31  ready   0x00000044 0x00000100    26%   0x00000020 00
0
main    10  close   0x0000006c 0x00000400    25%   0x00000013 00
0
msh />
msh />
msh />rt_backtrace spi1
stack info is as follows:
[0x0000f218]
[0x0000f846]
[0x0000f8d4]
[0x01063730]
msh />
```

图 4.1 backtrace 命令行示例

4.1.5 完善中……

4.2 FinSH 控制台

FinSH 是 RT-Thread 的命令行组件，提供一套供用户在命令行调用的操作接口，主要用于调试或查看系统信息，按照配置不同，可以分别支持传统命令行和 C 语言解释器模式。本身内置了一些常用的命令外，还可以用户自定义自己的命令。具体使用可以参考 RT-Thread 官网 https://www.rt-thread.org/document/site/programming-manual/finsh/finsh/。

可以把自定义的命令函数使用 MSH_CMD_EXPORT(name, desc)宏进行导出，添加到 CDK IDE 的构建目录中编译，在运行时通过命令行窗口就可以查看和使用添加的自定义命令了。

例子：

```
#include <rtthread.h>

void hello(void)
{
    rt_kprintf("hello RT-Thread!\n");
```

```
}

MSH_CMD_EXPORT(hello , say hello to RT-Thread);
```

编译，下载至目标板，系统运行起来后，在 FinSH 控制台按 tab 键可以看到导出的命令：

```
msh />
RT-Thread shell commands:
hello                - say hello to RT-Thread
version              - show RT-Thread version information
list_thread          - list thread
……
```

需要注意的是，导出的自定义命令运行上下文为 FinSH 线程本身，没有独立的堆栈空间，不是独立的线程。如果有独立于 FinSH 运行的特殊需求，需要手动开启新的线程来执行。

4.3 文件系统挂载

4.3.1 exFAT 支持

本 SDK 使用的 RT-Thread 提供了 exFAT 文件系统的支持。通过使用 exFAT 文件系统，可以方便对 SATA 盘和 U 盘进行挂载后进行文件操作及管理。

默认情况下，exFAT 的文件命名有如下缺点：

- 文件名（不含后缀）最长不超过 8 个字符，后缀最长不超过 3 个字符，文件名和后缀超过限制后将会被截断；
- 文件名不支持大小写（显示为大写）。

如果需要支持长文件名，则需要打开支持长文件名选项。选项和配置对应关系：

宏定义	描述
RT_DFS_ELM_USE_LFN_0	关闭长文件名
RT_DFS_ELM_USE_LFN_1	采用静态缓冲区支持长文件名，多线程操作文件名时将会带来重入问题
RT_DFS_ELM_USE_LFN_2	采用栈内临时缓冲区支持长文件名。对栈空间需求较大
RT_DFS_ELM_USE_LFN_3	使用 heap（malloc 申请）缓冲区存放长文件名，最安全（默认方式）

对文件系统进行初始化需要遵循以下流程：

(1) 确保 DFS 组件初始化成功；

(2) 注册文件系统 exFAT（rtconfig 中的文件系统和打开文件数量宏定义是否满足需要）；

(3) 驱动将设备注册成块设备；
(4) 将设备格式化（mkfs 命令）成相应的文件系统
(5) 创建（mkdir 命令）挂载点
(6) 将设备挂载（dfs_mount 函数）到挂载点目录

4.3.2 SATA 盘挂载

T6x0 芯片集成了 AHCI 控制器，可以连接 SATA 接口的硬盘来实现扩容存储功能。本示例展示了使用内核组件提供的 exFAT 文件系统来挂载使用硬盘的功能。

使用命令行挂载 exFAT 文件系统示例图：

图 4.2 文件系统挂载示例

4.3.3 U 盘挂载

T6x0 芯片集成了 XHCI 控制器，可以连接 USB 接口的存储设备。本示例展现了使用内核组件提供的 exFAT 文件系统来使用 USB 存储设备的功能。

```
msh />
msh />
A USB3 Device connected to port (1)...
found part[0], begin: 8257536, size: 28.647GB

msh />ls
Directory /:
exfat               <DIR>
ext4                <DIR>
ram                 <DIR>
rom                 <DIR>
usb                 <DIR>
msh />cd usb
msh /usb>mkdir test
msh /usb>ls
Directory /usb:
System Volume Information<DIR>
test                <DIR>
msh /usb>cd t
msh /usb>cd test
msh /usb/test>ls
Directory /usb/test:
msh /usb/test>echo ABCDEFabcdef123456 1
msh /usb/test>ls
Directory /usb/test:
1                   18
msh /usb/test>echo !@#$%%^&*() 2
msh /usb/test>ls
Directory /usb/test:
1                   18
2                   11
msh /usb/test>cat 1
ABCDEFabcdef123456msh /usb/test>
msh /usb/test>cat 2
!@#$%%^&*()msh /usb/test>
msh /usb/test>rm 2
msh /usb/test>ls
Directory /usb/test:
1                   18
msh /usb/test>cd /
msh />ls
Directory /:
exfat               <DIR>
ext4                <DIR>
ram                 <DIR>
rom                 <DIR>
usb                 <DIR>
msh />
```

4.4 网络配置

4.4.1 TCP/IP 协议栈

RT-Thread 内核组件集成的 LwIP 协议栈，是一个小型开源的 TCP/IP 协议栈，占用资源少，适合在嵌入式系统中使用。 关于 LwIP 的详细信息可以参见官网文档 http://savannah.nongnu.org/projects/lwip/。

本组件当前配置使用的主要协议：

- ARP 协议
- IP 协议
- ICMP 协议
- IGMPv2 协议
- UDP 协议
- TCP 协议
- DNS 协议
- DHCP 协议

参数配置选项：

参数项	默认值	说明
RT_LWIP_IGMP	定义	配置 IGMP 协议
RT_LWIP_ICMP	定义	配置 ICMP 协议
RT_LWIP_DNS	定义	配置 DNS 协议
RT_LWIP_DHCP	定义	配置 DHCP 协议
IP_SOF_BROADCAST	1	UDP/网络 RAW socket 是否可以发送广播包
IP_SOF_BROADCAST_RECV	1	UDP/网络 RAW socket 是否可以接收广播包
RT_LWIP_ETH_MTU	1500	MTU,最大 1500byte
RT_USING_LWIP_IPV6	1	配置 IPv6 协议
RT_LWIP_RAW	1	配置网络层 raw socket
RT_LWIP_RAWLL	1	配置链路层 raw socket
RT_USING_ETH0	定义	配置 GMAC0
RT_USING_ETH1	定义	配置 GMAC1
RT_LWIP_HW_OFFLOAD	1	配置网卡 checksum 计算硬件加速功能
RT_LWIP_SUPPORT_VLAN	未定义	配置 VLAN 功能；（注：网卡不支持对 VLAN 进行 checksum 硬件加速计算功能，所以配置 VLAN 时必须关闭

		RT_LWIP_HW_OFFLOAD）
RT_LWIP_VLAN_ID	99	VLAN ID
RT_LWIP_IPADDR	"192.168.1.30"	网卡默认 IP 地址
RT_LWIP_GWADDR	"192.168.1.1"	网卡默认网关地址
RT_LWIP_MSKADDR	"255.255.255.0"	网卡默认子网掩码
RT_LWIP_UDP	定义	配置 UDP 协议
RT_LWIP_TCP	定义	配置 TCP 协议
RT_MEMP_NUM_NETCONN	28	socket 管理结构个数，表示可同时建立的 socket 总个数;
RT_LWIP_PBUF_NUM	40	接收缓冲区个数
RT_LWIP_RAW_PCB_NUM	4	可建立的 RAW socket 个数
RT_LWIP_UDP_PCB_NUM	8	可建立的 UDP socket 个数
RT_LWIP_TCP_PCB_NUM	8	可建立的 TCP socket 个数
RT_LWIP_TCP_SEG_NUM	20	TCP 发送队列中最大分段个数
RT_LWIP_TCP_SND_BUF	1460*5	TCP 发送缓冲区大小
RT_LWIP_TCP_WND 1460*5	1460*5	TCP 窗口大小
RT_LWIP_TCPTHREAD_PRIORITY	10	LWIP 核心线程优先级
RT_LWIP_TCPTHREAD_MBOX_SIZE	10	LWIP 核心线程邮箱空间
RT_LWIP_TCPTHREAD_STACKSIZE	1024	LWIP 核心线程堆栈大小
LWIP_NO_RX_THREAD	未定义	网络数据包接收是否使用单独的线程，默认使用单独的线程
LWIP_NO_TX_THREAD	1	网络数据包发送是否使用单独的线程，默认不使用单独的线程，数据包直接在用户线程中发送。
RT_LWIP_ETHTHREAD_PRIORITY	10	线程优先级
RT_LWIP_ETHTHREAD_STACKSIZE	1024	数据包发送/接收线程堆栈大小
RT_LWIP_ETHTHREAD_MBOX_SIZE	20	数据包发送/接收线程邮箱大小
RT_LWIP_REASSEMBLY_FRAG	1	配置 IP 分片与重组功能
LWIP_SO_RCVTIMEO	1	配置 socket 接收超时选项
LWIP_SO_SNDTIMEO	1	配置 socket 发送超时选项
LWIP_SO_RCVBUF	1	配置 socket 接收缓冲区选项

4.4.2 网卡配置

本 SDK 网卡驱动提供了丰富的控制接口来配置网卡使用的硬件功能，通过类似于 ioctrl 方法供应用层调用，方便用户管理网络设备。

当前可用的网卡配置选项：

控制选项	选项 ID	说明
NIOCTL_GADDR	0x01	获取 MAC 地址
NIOCTL_SADDR	0x02	设置 MAC 地址
NIOCTL_SWKUP	0x03	设置样式唤醒方式的样式帧
NIOCTL_SLEEP	0x04	设置唤醒事件并进入休眠模式
NIOCTL_WAKEUP	0x05	退出休眠模式，并重新初始化网卡
NIOCTL_SSPED	0x06	设置网卡速率（10、100、1000Mbits/s）及双工模式(全双工、半双工)
NIOCTL_SFLOW	0x07	打开/关闭流控
NIOCTL_GCUNT	0x08	获取网卡收发统计信息
NIOCTL_SHASHTB	0x09	设置多播 hash 过滤表
NIOCTL_GHASHTB	0x0A	获取多播 hash 过滤表值
NIOCTL_STX	0x0B	打开/关闭 TX
NIOCTL_SRX	0x0C	打开/关闭 RX
NIOCTL_SJUMBO	0x0D	打开/关闭巨型帧
NIOCTL_SCHKSUM	0x0E	打开/关闭硬件 checksum offload
NIOCTL_SFILTER	0x0F	设置过滤器
NIOCTL_GFILTER	0x10	获取过滤器配置

示例源代码请参见 RTT_ROOT/examples/network/netcontrol.c

需要注意的是，对于巨型帧，若用户配置了 NIOCTL_SJUMBO，仅表示使能了硬件支持，此时并不能收/发巨型帧，用户还需要设置 MTU 来告知协议栈开启了巨型帧支持。

4.4.3 网络基础命令示例

4.4.3.1 ifconfig

命令 ifconfig 可以显示当前的网络连接状态、IP 地址、网关地址、dns 等信息，也可以用来手动设置网卡 IP 地址、网关地址以及子网掩码。

```
msh />ifconfig
network interface: e0 (Default)
MTU: 1500
MAC: 00 04 9f 05 44 e5
FLAGS: UP LINK_UP ETHARP
ip address: 192.168.12.127
gw address: 192.168.10.1
net mask  : 255.255.0.0
dns server #0: 192.168.10.1
dns server #1: 223.5.5.5
msh />
```

图 4.3 ifconfig 命令示例

4.4.3.2 netstat

命令 netstat 可以显示出当前所有的 TCP / IP 连接信息。

```
Active PCB states:
#0 192.168.12.186:49153 <==> 139.196.135.135:1883 snd_nxt 0x00001
AEA rcv_nxt 0x9D4D8F35 state: ESTABLISHED
Listen PCB states:
TIME-WAIT PCB states:
Active UDP PCB states:
#0 4 0.0.0.0:68 <==> 0.0.0.0:67
```

图 4.4 netstat 命令示例

4.4.3.3 dns

命令 dns 可以显示出现在使用的 dns 服务器地址，也可以通过输入 dns 服务器 IP 地址来手动设置 dns 服务器地址

```
msh />dns                                查看dns
dns server #0: 192.168.10.1
dns server #1: 223.5.5.5
msh />dns 114.114.114.114                设置dns
dns : 114.114.114.114
msh />dns                                查看dns
dns server #0: 114.114.114.114
dns server #1: 223.5.5.5
msh />
```

图 4.5 dns 命令示例

4.4.4 网络扩展命令示例

本 SDK 除了提供三个基础的网络命令（ifconfig、ping 、netstat）外，同时又集成 NetUtils 扩展软件包来丰富了网络常用命令，提供了 ping、tftp、ntp 等，极大提高了用户开发效率。

4.4.4.1 ping

命令 ping 是一种网络诊断工具，用来测试数据包能否到达特定主机。估算与主机间的丢失数据包率（丢包率）和数据包往返时间 RTT（网络时延，Round-trip delay time)。

- ping 域名

```
msh />ping rt-thread.org
60 bytes from 116.62.244.242 icmp_seq=0 ttl=49 time=11 ticks
60 bytes from 116.62.244.242 icmp_seq=1 ttl=49 time=10 ticks
60 bytes from 116.62.244.242 icmp_seq=2 ttl=49 time=12 ticks
60 bytes from 116.62.244.242 icmp_seq=3 ttl=49 time=10 ticks
msh />
```

图 4.6 ping 域名命令示例

- ping IP 地址

```
msh />ping 192.168.10.12
60 bytes from 192.168.10.12 icmp_seq=0 ttl=64 time=5 ticks
60 bytes from 192.168.10.12 icmp_seq=1 ttl=64 time=1 ticks
60 bytes from 192.168.10.12 icmp_seq=2 ttl=64 time=2 ticks
60 bytes from 192.168.10.12 icmp_seq=3 ttl=64 time=3 ticks
msh />
```

图 4.7 ping IP 地址命令示例

4.4.4.2 ntp

命令 ntp 用来同步网络中各个计算机时间，通过自动或者手动方式连接指定的时间服务器来获取当前 UTC 时间，并更新到本地。

使用该命令需要配置 RTC 设备，使用命令行调用方式同步时间示例：

```
msh />ntp_sync
Get local time from NTP server: Sat Feb 10 15:22:33 2018
The system time is updated. Timezone is 8.
msh />
```

图 4.8 ntp 命令示例

4.4.4.3 tftp

TFTP（Trivial File Transfer Protocol，简单文件传输协议）是 TCP/IP 协议族中的一个用来在客户机与服务器之间进行简单文件传输的协议，端口号为**69**，适用于小型的嵌入式产品上。

具体使用流程介绍如下：

（1） 安装 tftp 客户端

在 PC 机上面，安装 tftp 客户端软件，如 Tftpd64-4.60-setup.exe 。

（2） 启动 tftp 服务器

在 RT-Thread 上使用 FinSH 控制台启动 tftp 服务器：

```
msh />tftp_server
TFTP server start successfully.
msh />
```

图 4.9 tftp 命令示例

（3） tftp 客户端配置

打开刚安装的 Tftpd64 软件，按下图操作进行配置：

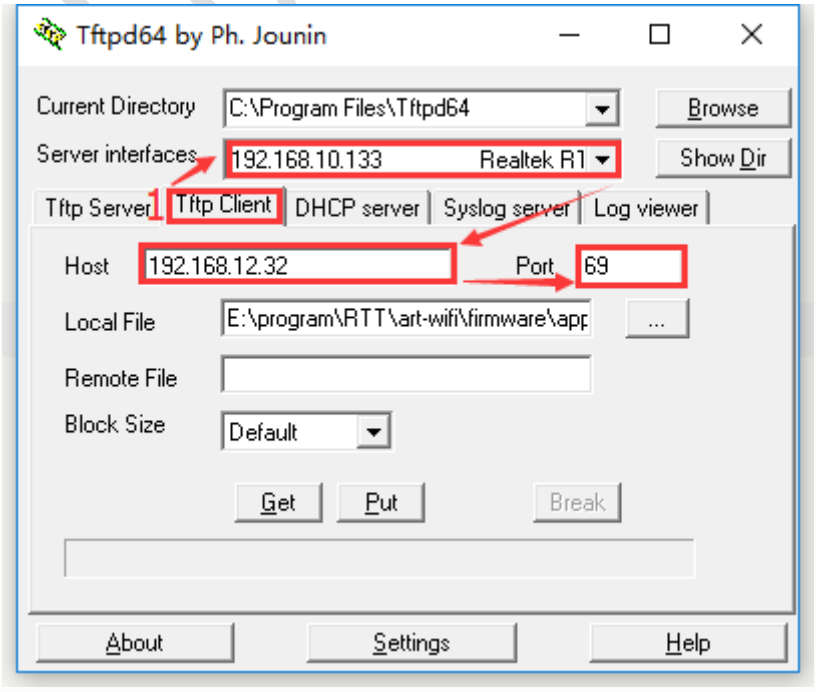


图 4.9 tftp 客户端配置 1

（4） 发送文件到 RT-Thread

设置服务器端保存文件的路径（包括文件名）时，配置相对路径或者绝对路径后，点击 put 即可完成发送。

需要注意的是，如果 RT-Thread DFS_USING_WORKDIR 未开启，同时 Remote File 为空，文件会将保存至根路径下。

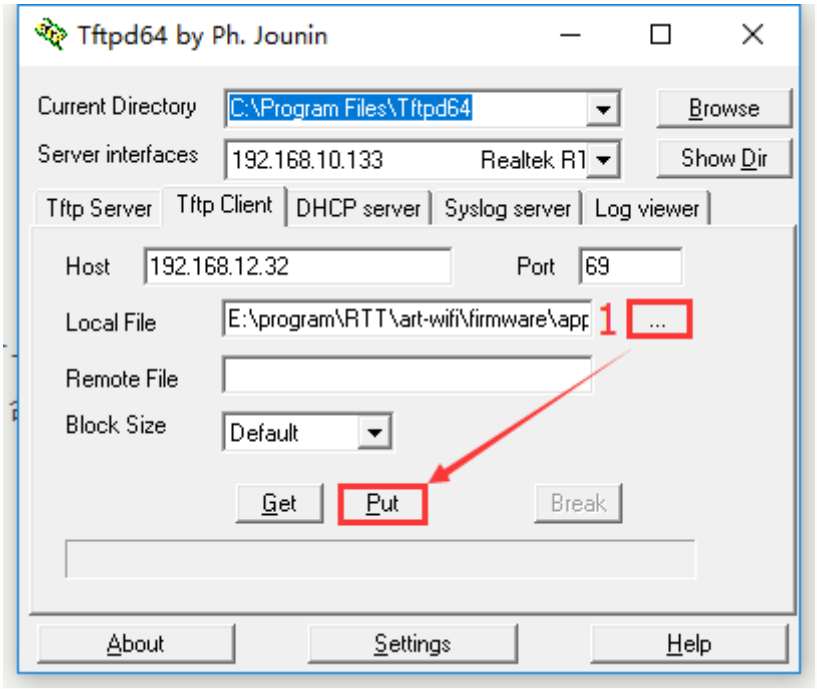


图 4.10 tftp 客户端配置 2

（5） 从 RT-Thread 接收文件

设置接收服务器端文件存放的路径（包括文件名）时，配置相对路径或者绝对路径后点击 get 即可完成接收。

如下图所示，将 /web_root/image.jpg 保存到本地，这里使用的是绝对路径：

```
msh /web_root>ls                ## 查看文件是否存在
Directory /web_root:
image.jpg                10559
msh /web_root>
```

图 4.11 tftp 接收文件

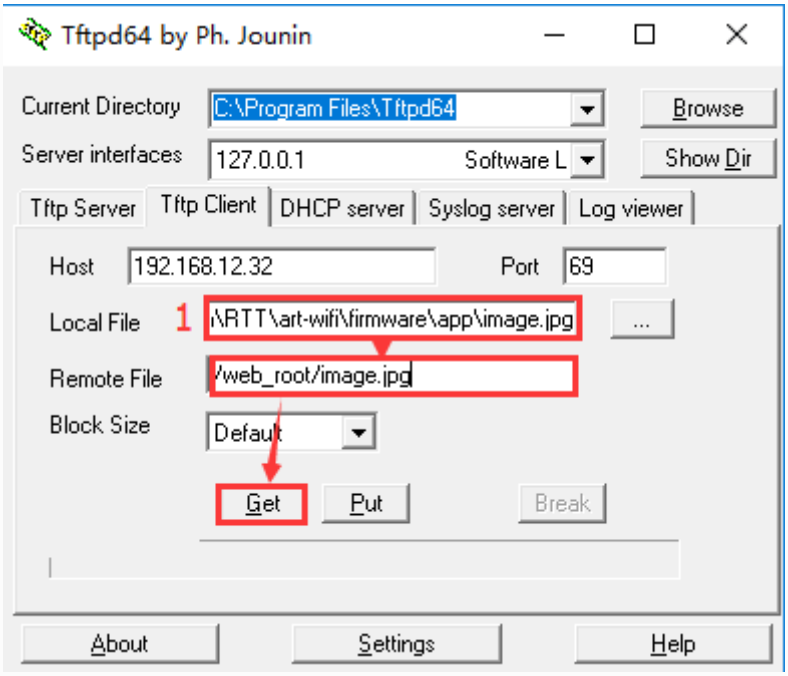


图 4.12 tftp 客户端配置 3

4.5 网络开发

4.5.1 socket 编程

一般 socket 编程模型如下图所示：

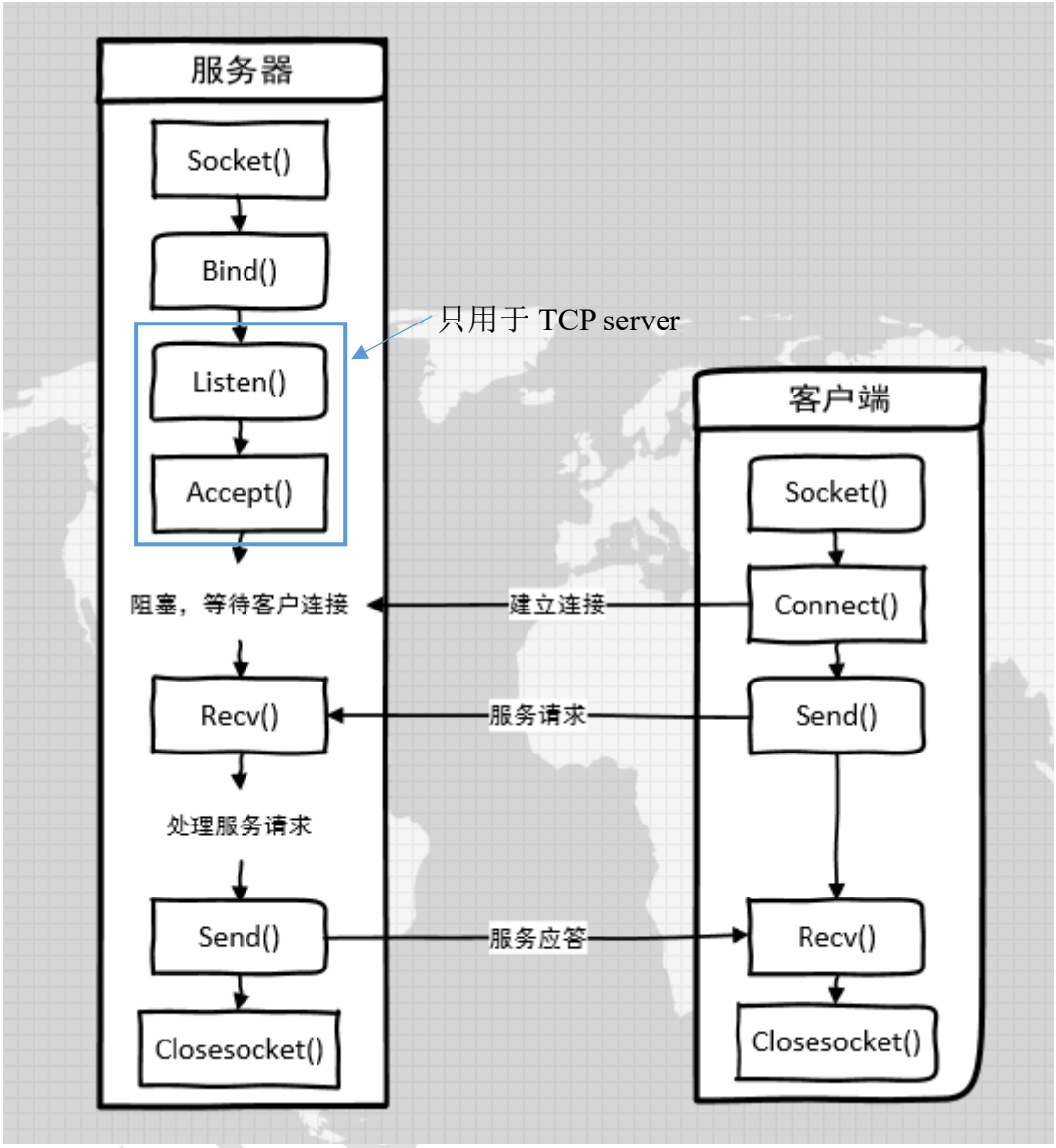


图 4.13 socket 编程模型

(1) 服务器使用流程：

socket() 创建一个 socket，返回套接字的描述符，并为其分配系统资源。

bind() 将套接字绑定到一个本地地址和端口上。

listen() 将套接字设为监听模式并设置监听数量，准备接收客户端请求(TCP server)。

accept() 接受客户端发起连接，并返回已接受连接的新套接字描述符(TCP server)。

recv()/send() 用新套接字与客户端进行通信。

closesocket() 关闭 socket，回收资源。

(2) 客户端使用流程：

socket() 创建一个 socket，返回套接字的描述符，并为其分配系统资源。

connect() 向服务器发出连接请求。

send()/recv() 与服务器进行通信。

closesocket() 关闭 socket，回收资源。

(3) 示例代码

示例程序路径	说明
examples/network/tcpclient.c	IPv4 TCP 客户端示例
examples/network/tcpserver.c	IPv4 TCP 服务端示例
examples/network/udpclient.c	IPv4 UDP 客户端示例
examples/network/udpserver.c	IPv4 UDP 服务端示例

上述示例源代码对应的上位机(Linux)程序:

示例程序路径	说明
software/linux/tester/tcp/server.c	IPv4 TCP 服务端示例
software/linux/tester/tcp/client.c	IPv4 TCP 客户端示例
software/linux/tester/udp/server.c	IPv4 UDP 服务端示例
software/linux/tester/udp/client.c	IPv4 UDP 客户端示例

4.5.2 非阻塞 socket 编程

在上一节 socket 编程示例中，程序是阻塞的，socket 函数都要等返回后才能进行下一步，如果 recv 一直没有数据，那么就一直不会返回，整个线程就阻塞在那。相对于阻塞模式，还有非阻塞模式，介绍如下:

(1) 网络中阻塞和非阻塞模型的区别

套接字阻塞与非阻塞模式的区别可分为以下四类：

- 输入操作函数 read、recv 和 recvfrom

  对于 TCP 的阻塞模式来说，如果某个线程调用这些输入函数，而该 socket 的接收缓冲区中没有数据可读，那么该线程将被投入睡眠，直到有数据到达。由于 TCP 是字节流协议，该线程唤醒可能是单个字节，也可能是一个完整的 TCP 分段数据。

  对于 UDP 的阻塞模式来说，如果某个线程对调用这些输入函数，而该 socket 的接收缓冲区中没有数据可读，那么该线程将被投入睡眠，直到有数据到达。由于 UDP 是数据报协议，该线程的唤醒是要有一个完整的数据报可读。

  对于非阻塞模式来说，如果 TCP/UDP 输入操作不能被满足（对于 TCP 即至少有一个字节的数据可读，对于 UDP 有一个完整的数据报可读），相应调用将立即返回一个 EWOULDBLOCK 错误。通过这种错误码，提示用户可以重新再调用一次。

- 输出操作函数 write、send 和 sendto

  对于 TCP/UDP 输出，内核将从应用线程的缓冲区复制数据到 socket 发送缓冲区即可。

  对于阻塞模式，如果 socket 发送缓冲区中没有空间，线程将被投入睡眠，直到有空间为止。

对于非阻塞模式，如果 socket 发送缓冲区中没有空间，输出函数将立即返回 EWOULDBLOCK 错误。如果 socket 发送缓冲区中仅有部分空间，将返回实际复制的字节数(也叫做不足计数)。

- 接受连接函数 accept

对于 TCP 的阻塞模式，线程调用 accept 函数后，并且尚无新的连接到达时，将被投入睡眠。

对于 TCP 的非阻塞模式，线程调用 accept 函数后，并且尚无新的连接到达，将立即返回一个 EWOULDBLOCK 错误。

- 发起连接函数 connect

对于 TCP 的阻塞模式，由于连接的建立涉及一个三次握手过程，线程调用 connect 函数要等收到本次连接 SYN 的 ACK 才返回。这意味者 TCP 的每一次 connect 总会阻塞调用线程至少一个到服务器的 RTT 时间。

对于 TCP 的非阻塞模式，一般连接并不能立即建立，线程调用 connect 函数一般会立刻返回一个 EINPROGRESS 错误。但是，有些连接的服务器和客户机处于同一个主机的情况下，也可以立即建立。因此，即便是非阻塞的 connect，也要处理 connect 成功返回的情况。

(2) 非阻塞 socket 模型

- 方式一：设置阻塞 socket 超时参数

setsockopt(sock, SOL_SOCKET, SO_SNDTIMEO, (void *)&timeout, sizeof(timeout))
setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, (void *)&timeout, sizeof(timeout))
分别为发送或接收设定等待超时时间；若在设定的时间仍未发送成功或接收到数据，则返回。

- 方式二：设置 socket 为非阻塞模式

对于 read/write()操作，可以使用下面 3 这种方式设置：

编号	函数设置
1	ioctl（sock, FIONBIO, true）
2	ioctlsocket（sock, FIONBIO, true）IPv4
3	fcntl（sock ，F_SETFL，O_NONBLOCK）

对于 send/sendto()/recv()/recvfrom()操作，可以设置

编号	参数设置
1	flag |=MSG_DONTWAIT;

- 方式三：使用 select 或 poll 轮询

select()/poll() 可以阻塞地探测一组支持非阻塞的 I / O 设备是否有事件发生（如可读，可写，出现异常等等）。当触发了事件或者超时后，就会函数返回来实现无阻塞地读取到数据。它主要用来处理 I / O 多路复用的情况。

(3) 示例代码

示例程序路径	说明
examples/network/ipraw.c	网络层 RAW socket
examples/network/packetraw.c	链路层 RAW socket

4.5.3 raw socket 编程

常用的网络 socket 编程都是在应用层进行的数据收发操作，也就是使用流式套接字(SOCK_STREAM)和数据报式套接字(SOCK_DGRAM)。这些应用数据的打包/拆包都是由系统协议栈实现，然而在某些情况下需要执行更底层的操作，比如修改报文头、避开系统协议栈等。这个时候，就需要使用其他类型的套接字。

4.5.3.1 IP 层 raw socket

SOCK_RAW socket 是一种不同于 SOCK_STREAM、SOCK_DGRAM 的 socket，它实现于协议栈核心，可以实现构造 IP 报文。SOCK_RAW socket 不仅可以处理 ICMP、IGMP 等网络报文，而且通过 IP_HDRINCL 选项可以处理特殊的 IPv4 报文，甚至由用户构造 IP 头。

使用上，和普通 socket 的区别有一些区别：

如果设置 IP_HDRINCL 选项，SOCK_RAW 可以操作 IP 头+数据（也就是用户需填充 IP 头及 payload）；

如果使用 bind 函数绑定本地 IP，且 IP_HDRINCL 选项不设置，则用此 IP 填充源 IP 地址；若不调用 bind 函数则将外出接口的主 IP 地址填充源 IP 地址。

如果使用 connect 函数设置目标 IP，则可以使用 send 或者 write 函数发送报文，而不需要使用 sendto 函数。

另外，端口对于 SOCK_RAW 而言没有任何意义。

4.5.3.2 链路层 raw socket

使用链路层 raw socket，可以对数据链路帧进行自定义，然后协议栈直接在设备驱动层接收或发送原始链路数据帧, 方便在物理层以上构建通信协议。

使用链路层 raw socket 发送时，用户必须手动构建包括物理头部在内的完整的链路数据帧后，将此帧加入网卡发送队列。

使用链路层 raw socket 接收时，默认会接收所有接口的链路数据帧. 当使用 bind 来绑定 sockaddr_ll 地址结构后，仅接收本接口的包。

另外，构建链路层 raw socket 时，需要指定 protocol family 为 PF_PACKET, socket_type 为 SOCK_RAW。示例代码请参考：

示例程序路径	说明
examples/network/ipraw.c	综合示例代码

examples/network/packetraw.c	上传文件示例代码

4.5.4 国密/国际 SSL 编程

本 SDK 在 mbed TLS 基础上对 SSL API 接口进行了封装，使其使用起来更方便。其中支持了国际 TLS 标准和新增加的国密标准，但是国密 SSL 密码套件仅支持 ECC-SM4-SM3。

其中，使用国密 SSL 标准时，需要在 rtconfig.h 文件中打开宏 MBEDTLS_GMSSL。

该 API 分为客户端和服务器端，客户端或者服务器端可以选择自己使用的 SSL 标准进行连接。如下进行宏配置：

名称	SSL 配置	宏配置	备注
客户端	国际 SSL	SSL_PRESET_SUITE_GE:全部套件 SSL_PRESET_SUITE_GE_V12:仅使用 TLS1.2 套件	默认支持
	国密 SSL	SSL_PRESET_SUITE_GM	宏 MBEDTLS_GMSSL 开启
服务器端	国际 SSL	SSL_PRESET_SUITE_GE:全部套件 SSL_PRESET_SUITE_GE_V12:仅使用 TLS1.2 套件	默认支持
	国密 SSL	SSL_PRESET_SUITE_GM	宏 MBEDTLS_GMSSL 开启
	国际 SSL+国密 SSL	SSL_PRESET_SUITE_GE_AND_GM	

其中示例参考代码简介如下:

示例程序路径	说明
examples/mbedtls/ssl_rw_client.c	client 使用示例代码
examples/mbedtls/ssl_rw_server.c	server 使用示例代码

4.5.4.1 客户端 client

客户端 SSL API 如下:

函数名称
tls_cli_session *tls_cli_session_create(int8 *host, int8 *port, int32 buf_size)
int tls_cli_session_destroy(tls_cli_session *session)
int tls_cli_session_init(tls_cli_session *session, int suite_preset)
int tls_cli_session_close(tls_cli_session *session)
int tls_cli_session_config(tls_cli_session *session)
int tls_cli_session_connect(tls_cli_session *session)
int tls_cli_session_read(tls_cli_session *session, unsigned char *buf , size_t len)
int tls_cli_session_write(tls_cli_session *session, const unsigned char *buf , size_t len)

4.5.4.2 服务器端 server

服务器端 SSL API 如下:

函数名称
tls_srv_context* tls_srv_create(int8 *bind_host, int8 *bind_port, int suite_preset)
int tls_srv_destroy(tls_srv_context *ctx)
int tls_srv_listen(tls_srv_context *ctx)
tls_srv_session* tls_srv_session_create(tls_srv_context *ctx)
int tls_srv_session_destroy(tls_srv_session *session)
int tls_srv_session_init(tls_srv_context *ctx, tls_srv_session *session)
int tls_srv_session_close(tls_srv_session *session)
int tls_srv_session_config(tls_srv_session *session)
int tls_srv_session_read(tls_srv_session *session, unsigned char *buf, size_t len)
int tls_srv_session_write(tls_srv_session *session, const unsigned char *buf, size_t len)

4.6 存储空间分配

T6x0 片内存储空间主要包含 ROM，Norflash，I/D-TCM，SRAM，等。其运行速度为 I/D-TCM> SRAM>Norflash，各空间用途说明如下表所示。

T6x0 芯片包含国产 CK803S CPU 核，用来运行 RT-Thread RTOS(简称固件)。使用量产工具可以把固件下载存储在 NOR FLASH 中，当运行时由 BOOTROM 负责搬移至具体的存储空间中。其中，固件代码分为快速代码（加载到 SRAM 运行）和慢速代码（在 NOR FLASH 运行）两部分；为了加速代码执行，可以配置 16KB I/D cache 来加速。

存储空间说明

存储空间	容量	CPU 访问速度(A>B>C)	用途说明	开发是否可用
ROM	32KB	B	芯片启动引导程序 BOOTROM	否
NOR FLASH	512KB	C	固件、应用私有数据存储区; 固件慢速代码运行区	是
D-TCM	32KB	A	固件数据段、BSS 段和中断栈空间运行区；	是
SRAM	8KB	B	硬件模块配置信息数据区	否
SRAM	256KB	B	固件快速代码段、.EXDATA 数据和堆运行区； 数据缓冲 BUF 区	是

下面对开发中使用的存储空间进行具体介绍：

4.6.1 D-TCM

D-TCM，为 32KB，靠近 CPU，运行数据速度最快，但掉电数据丢失。
D-TCM 大部分用于固件的数据、BSS 段和中断栈空间。应用例程当前默认使用情况：

表 D-TCM 空间划分信息

区域名称	起始位置	区域大小	说明
变量区	0	26KB	固件全局数据、BSS 段
中断栈区	26KB	4KB	固件运行的中断栈空间

但是需要特别注意的是，由于 D-TCM 靠近 CPU，有些硬件模块访问不到此处存储空间，在使用时需要特别注意，如 GMAC、SATA HOST/DEVICE、CRYPTO 和 USB。

4.6.2 SRAM

T6x0 内部包含两块 SRAM，大小分别为 256KB 和 8KB，较靠近 CPU，运行代码数据速度较快，数据掉电丢失。

（1） 256KB 的 SRAM 划分用于固件快速代码、.EXDATA 数据和堆运行区。此外还用于数据 BUF 缓冲区。256KB SRAM 使用分配如下表所示（请参考.ld 链接文件中定义），特说明如下：

区域名称	起始位置	区域大小	说明
.快速代码区	0	SIZEOF(.text_1)	固件中添加__fast 属性代码段
EXDATA 数据区	128KB	SIZEOF(.exdata)	固件中添加__exdata 属性的数据区
堆区	紧邻.exdata 数据区	用户自定义	固件堆管理区
缓冲 BUF 区	紧邻堆区	上述剩余空间	数据缓冲 BUF 区

1) 为了方便固件快速代码使用该 SRAM 运行， SDK 定义了__fast 属性。凡是具有__fast 属性的代码会被加载至此 SRAM 中，否则属于慢速代码运行在 NOR FLASH 中。__fast 属性使用方式如下面代码清单所示。

```
1. void __fast function(void)
2. {
3.     ...
4. }
```

2) 为了方便固件 EXDATA 数据使用该 SRAM，SDK 还定义了__exdata 属性，凡是具有__exdata 属性的全局变量会被编译器分配在 256KB SRAM 上。紧随添加__fast 属性的代码段之后。具体使用方式可参考下面代码片段。

```
#define IO_BUF_SIZE     0x8000
uint8 __exdata io_buf[IO_BUF_SIZE];
```

3) 为了方便固件堆区使用该 SRAM，跟随__exdata 属性定义的全局缓冲区之后，被划分为 malloc 堆管理区。大小可以自己分配。用户可以使用 malloc/free 来合理使用该区域。

4) 用户可以在堆区后，定义自己专用的 BUF 数据缓冲区，不能和堆区重叠，可选配。

（2） 8KB 的 SRAM 专用于硬件模块配置信息数据区。开发中不能再次使用该 SRAM，避免硬件模块使用的配置信息被覆盖。

4.6.3 NOR FLASH

T6x0 芯片内部包含一块 512KB NOR FLASH，运行代码数据速度最慢，但具有掉电数据不易失特性，常用于存储固件代码和应用数据。有时候，作为存储空间的扩展之用，对性能无关的代码只读数据可以运行在 NOR FLASH。其空间使用划分如下图所示。

图 4.14 NOR FLASH 存储空间划分

如上图示，NOR FLASH 起始位置为固件存储区，随后为用户应用数据区，尾部为私有数据区，使用时请确保各区域空间划分不存在重叠。