翼辉信息
ACOINFO

MS-RTOS 设备开发

API 手册

文档版本 V1.0 / 发布日期 2023-08-09

西安翼辉爱智物联技术有限公司

目 录

1 MS-RTOS 数据类型

> 本章将介绍 MS-RTOS 开发用到的基础数据类型和宏定义。

基础数据类型

类型	描述
ms_bool_t	布尔类型
ms_uint8_t	8 位无符号整型
ms_uint16_t	16 位无符号整型
ms_uint32_t	32 位无符号整型
ms_uint64_t	64 位无符号整型
ms_int8_t	8 位有符号整型
ms_int16_t	16 位有符号整型
ms_int32_t	32 位有符号整型
ms_int64_t	64 位有符号整型
ms_size_t	32 位无符号大小类型
ms_ssize_t	32 位有符号大小类型
ms_ptr_t	指针类型，MS_NULL 为空指针
ms_const_ptr_t	const 指针类型，MS_NULL 为空指针
ms_ulong_t	无符号 long 类型
ms_long_t	有符号 long 类型
ms_ullong_t	无符号 long long 类型
ms_llong_t	有符号 long long 类型

ms_bool_t

布尔类型，可取的值为以下的宏：

宏	含义
MS_TRUE	真
MS_FALSE	假

其它数据类型

类型	描述
ms_err_t	32 位错误码类型
ms_stack_t	堆栈类型
ms_addr_t	地址类型
ms_func_t	无参数无返回值的函数指针类型
ms_callback_t	有参数无返回值的回调函数指针类型
ms_printf_func_t	printf 格式打印函数指针类型
ms_handle_t	32 位句柄、ID 号类型，MS_HANDLE_INVALID 为无效的句柄、ID 号
ms_index_t	16 位资源索引类型
ms_prio_t	8 位的优先级类型，有效范围 0 - 254，数字越小，优先级越高，MS_PRIO_INVALID 为无效的优先级
ms_ipc_opt_t	IPC 对象选项
ms_pid_t	8 位的进程号类型，有效范围 0 - 254，MS_PID_KERN 为内核线程的进程号，MS_PID_INVALID 为无效的进程号
ms_arch_sr_t	CPU 状态寄存器类型
ms_mem_region_t	内存区域类型
ms_mem_layout_t	内存布局类型

ms_func_t

无参数无返回值的函数指针类型：

```
typedef void (*ms_func_t)(void);
```

ms_callback_t

有参数无返回值的回调函数指针类型：

```
typedef void (*ms_callback_t)(ms_ptr_t arg);
```

ms_printf_func_t

printf 格式打印函数指针类型：

```
typedef void (*ms_printf_func_t)(const char *fmt, ...) MS_PRINTF_ATTR;
```

ms_ipc_opt_t

IPC 对象选项类型，可取的值为以下的宏：

宏	含义
MS_WAIT_TYPE_PRIO	按优先级高低规则等待
MS_WAIT_TYPE_FIFO	按先来先服务规则等待
MS_IPC_OPT_PROCESS	进程私有的 IPC 对象
MS_IPC_OPT_SHARE	进程间共享的 IPC 对象

ms_arch_sr_t

CPU 状态寄存器类型用于关闭与恢复 CPU 中断，用于以下接口：

```
ms_arch_sr_t ms_arch_int_disable(void);
void ms_arch_int_resume(ms_arch_sr_t sr);
```

ms_mem_region_t

内存区域类型是一个结构体：

```
typedef struct {
    ms_addr_t   base;
    ms_size_t   size;
} ms_mem_region_t;
```

参数	说明
base	内存区域的基地址
size	内存区域的大小

ms_mem_layout_t

内存布局类型是一个内存区域类型的数组：

```
typedef ms_mem_region_t ms_mem_layout_t;
```

宏

MS-RTOS 定义了一些十分有用的宏：

宏	含义
MS_ARCH_CPU_BITS	CPU 位宽
MS_ARCH_STK_ALIGNMENT	堆栈对齐要求
MS_ARCH_STK_MIN_SIZE	堆栈空间的最小值
MS_ARCH_CACHE_LINE_SIZE	D-CACHE 行大小
MS_ARCH_DCACHE_LINE_SIZE	D-CACHE 行大小
MS_ARCH_ICACHE_LINE_SIZE	I-CACHE 行大小
MS_ARCH_IDLE()	CPU 进入空闲状态
MS_ARCH_ISB()	指令屏障
MS_ARCH_MB()	内存屏障
MS_ARCH_RMB()	读内存屏障
MS_ARCH_WMB()	写指令屏障
MS_ARCH_NOP()	NOP 指令
MS_WEAK	弱符号
MS_PRINTF_ATTR	printf 格式打印函数属性
MS_SNPRINTF_ATTR	snprintf 格式打印函数属性
MS_PRINTK_ATTR	ms_printk 格式打印函数属性
MS_LOG_ATTR	ms_log 格式打印函数属性
MS_HEAP_ALIGNMENT	内存堆分配内存的对齐要求
MS_STRUCT_PACK_BEGIN	pack 紧排结构体开始
MS_STRUCT_PACK_END	pack 紧排结构体结束
MS_STRUCT_PACK_FIELD(field)	pack 紧排结构体成员
MS_STRUCT_PACK_STRUCT	pack 紧排结构体
MS_ALIGN_ATTR(var, alignment)	定义一个地址对齐的变量
MS_FORCE_INLINE	强制内联函数
MS_SHELL_CMD(_name, _func, _help, var_name)	定义一条 shell 命令（仅内核空间可用）
MS_BIT(bit)	位
MS_ARRAY_SIZE(array)	数组的大小

宏	含义
MS_ROUND_UP(x, align)	向上圆整
MS_ROUND_DOWN(x, align)	向下圆整
MS_IS_ALIGNED(x, align)	判断是否对齐
MS_IS_POWER_OF_2(x)	判断是否为 2 的次方
MS_MIN(a, b)	最小值
MS_MAX(a, b)	最大值
MS_TICK_PER_SEC	每秒的 tick 数
MS_MS_PER_SEC	每秒的毫秒数
MS_US_PER_SEC	每秒的微秒数
MS_NS_PER_SEC	每秒的纳秒数
MS_US_PER_MS	每毫秒的微秒数
MS_NS_PER_MS	每毫秒的纳秒数
MS_NS_PER_US	每微秒的纳秒数
MS_NS_PER_TICK	每 tick 的纳秒数
MS_MS_TO_TICK(ms)	毫秒数转 tick 数
MS_TICK_TO_MS(tick)	tick 数转毫秒数
MS_CONTAINER_OF(entry, type, member)	通过成员地址获得容器（结构体）的地址

MS_STRUCT_PACK_???

使用 `MS_STRUCT_PACK_???` 宏定义一个 pack 紧排结构体的方式如下：

```
MS_STRUCT_PACK_BEGIN
struct xxx {
    MS_STRUCT_PACK_FIELD(ms_uint32_t yyy);
} MS_STRUCT_PACK_STRUCT;
MS_STRUCT_PACK_END
```

MS_SHELL_CMD

使用 `MS_SHELL_CMD` 宏定义一条内核命令的方式如下：

```
#include "ms_shell.h"

/**
 * @brief xxx command.
```



```
 *
 * @param[in] argc    Arguments count
 * @param[in] argv    Arguments array
 * @param[in] io      Pointer to shell io driver
 *
 * @return N/A
 */
static void __ms_shell_xxx(int argc, char *argv[], const ms_shell_io_t *io)
{
    /* do some thing */
}
MS_SHELL_CMD(xxx, __ms_shell_xxx, "xxx", __ms_shell_cmd_xxx);
```

汇编文件使用的宏

针对汇编文件，MS-RTOS 还提供了以下的宏定义：

宏	含义
FILE_BEGIN	汇编文件的开始
FILE_END	汇编文件的结尾
ENTRY(name)	汇编函数入口
ENDPROC(name)	汇编函数结尾
LABEL(label)	定义行标签

2 MS-RTOS 错误码

> 本章将介绍 MS-RTOS 的错误码。
>
> 内核错误码的定义位于 libmsrtos/src/kern/ms_err.h 文件中。

MS-RTOS 内核错误码

宏	值	含义
MS_ERR_NONE	0	成功，没有错误
MS_ERR	-1	错误，errno 存放了 posix 错误码
MS_ERR_ARG_NULL_PTR	-10001	空指针参数
MS_ERR_ARG_INVALID	-10002	无效参数
MS_ERR_ARG_BAD_PTR	-10003	错误指针参数
MS_ERR_ARG_BAD_ADDR	-10004	错误地址参数
内核		
MS_ERR_KERN_RUNNING	-11001	内核已经运行
MS_ERR_KERN_NOT_RUNNING	-11002	内核未运行
MS_ERR_KERN_IN_ISR	-11003	在中断服务程序中
MS_ERR_KERN_NOT_IN_ISR	-11004	不在中断服务程序中
MS_ERR_KERN_INT_OVERFLOW	-11005	中断嵌套次数溢出
MS_ERR_KERN_SCHED_LOCK	-11006	内核调度器已被上锁
MS_ERR_KERN_SCHED_NOT_LOCK	-11007	内核调度器未被上锁
MS_ERR_KERN_SCHED_LOCK_OVERFLOW	-11008	内核调度器上锁次数溢出

宏	值	含义
MS_ERR_KERN_SLEEP_ABORT	-11009	线程休眠被终止
MS_ERR_KERN_NO_PERM	-11010	没有权限
线程		
MS_ERR_KERN_NO_THREAD	-12001	线程不足
MS_ERR_KERN_THREADID_INVALID	-12002	线程 ID 无效
MS_ERR_KERN_THREAD_INVALID	-12003	线程无效
计数信号量		
MS_ERR_KERN_NO_SEMC	-13001	计数信号量不足
MS_ERR_KERN_SEMCID_INVALID	-13002	计数信号量 ID 无效
MS_ERR_KERN_SEMC_INVALID	-13003	计数信号量无效
MS_ERR_KERN_SEMC_DESTROY	-13004	计数信号量被销毁
MS_ERR_KERN_SEMC_WAIT_TIMEOUT	-13005	计数信号量等待超时
MS_ERR_KERN_SEMC_WAIT_ABORT	-13006	计数信号量等待终止
MS_ERR_KERN_SEMC_EMPTY	-13007	计数信号量值为 0，尝试等待失败
MS_ERR_KERN_SEMC_OVERFLOW	-13008	计数信号量值溢出
MS_ERR_KERN_COND_EMPTY	-13007	条件变量值为 MS_FALSE，尝试等待失败
事件标志组		
MS_ERR_KERN_NO_EVENTSET	-14001	事件标志组不足
MS_ERR_KERN_EVENTSETID_INVALID	-14002	事件标志组 ID 无效

宏	值	含义
MS_ERR_KERN_EVENTSET_INVALID	-14003	事件标志组无效
MS_ERR_KERN_EVENTSET_DESTROY	-14004	事件标志组被销毁
MS_ERR_KERN_EVENTSET_WAIT_TIMEOUT	-14005	事件标志组等待超时
MS_ERR_KERN_EVENTSET_WAIT_ABORT	-14006	事件标志组等待终止
MS_ERR_KERN_EVENTSET_EMPTY	-14007	事件不满足，尝试等待失败
互斥量		
MS_ERR_KERN_NO_MUTEX	-15001	互斥量不足
MS_ERR_KERN_MUTEXID_INVALID	-15002	互斥量 ID 无效
MS_ERR_KERN_MUTEX_INVALID	-15003	互斥量无效
MS_ERR_KERN_MUTEX_BUSY	-15004	互斥量忙（使用中）
MS_ERR_KERN_MUTEX_WAIT_TIMEOUT	-15005	互斥量等待超时
MS_ERR_KERN_MUTEX_WAIT_ABORT	-15006	互斥量等待终止
MS_ERR_KERN_MUTEX_NOT_OWNER	-15007	不是互斥量的拥有者线程
MS_ERR_KERN_MUTEX_NOT_LOCK	-15008	互斥量没有上锁
MS_ERR_KERN_MUTEX_OVERFLOW	-15009	互斥量上锁次数溢出
MS_ERR_KERN_MUTEX_DESTROY	-15010	互斥量被销毁
MS_ERR_KERN_MUTEX_EMPTY	-15011	互斥量已被其它线程占有，尝试等待失败
消息队列		

宏	值	含义
MS_ERR_KERN_NO_MQUEUE	-16001	消息队列不足
MS_ERR_KERN_MQUEUEID_INVALID	-16002	消息队列 ID 无效
MS_ERR_KERN_MQUEUE_INVALID	-16003	消息队列无效
MS_ERR_KERN_MQUEUE_DESTROY	-16004	消息队列被销毁
MS_ERR_KERN_MQUEUE_WAIT_TIMEOUT	-16005	消息队列等待超时
MS_ERR_KERN_MQUEUE_WAIT_ABORT	-16006	消息队列等待终止
MS_ERR_KERN_MQUEUE_EMPTY	-16007	消息队列空，尝试等待失败
MS_ERR_KERN_MQUEUE_FULL	-16008	消息队列满，尝试发送失败
内存池		
MS_ERR_KERN_NO_MEMPOOL	-17001	内存池不足
MS_ERR_KERN_MEMPOOLID_INVALID	-17002	内存池 ID 无效
MS_ERR_KERN_MEMPOOL_INVALID	-17003	内存池无效
MS_ERR_KERN_MEMPOOL_DESTROY	-17004	内存池被销毁
MS_ERR_KERN_MEMPOOL_WAIT_TIMEOUT	-17005	内存池等待超时
MS_ERR_KERN_MEMPOOL_WAIT_ABORT	-17006	内存池等待终止
MS_ERR_KERN_MEMPOOL_EMPTY	-17007	内存池空，尝试分配失败
MS_ERR_KERN_MEMPOOL_BAD_PTR	-17008	释放的内存指针错误
内存堆		

宏	值	含义
MS_ERR_KERN_HEAP_NO_MEM	-18001	内存堆内存不足
MS_ERR_KERN_HEAP_NO_ZONE	-18002	内存堆内存区不足
MS_ERR_KERN_HEAP_LOCK_FAILED	-18003	内存堆上锁失败
MS_ERR_KERN_HEAP_BAD_PTR	-18004	释放的内存指针错误
MS_ERR_KERN_HEAP_DOUBLE_FREE	-18005	重复释放同一个内存指针
MS_ERR_KERN_HEAP_OVERFLOW	-18006	向后越界
MS_ERR_KERN_HEAP_UNDERFLOW	-18007	向前越界
中断		
MS_ERR_KERN_INT_NO_ISR	-19001	指定中断没有安装中断服务程序
MS_ERR_KERN_INT_IRQ_INVALID	-19002	中断号无效
进程		
MS_ERR_KERN_NO_PROCESS	-20001	进程不足
MS_ERR_KERN_PROCESSID_INVALID	-20002	进程 ID 无效
MS_ERR_KERN_PROCESS_INVALID	-20003	进程无效
MS_ERR_KERN_PROCESS_IMG_INVALID	-20004	进程镜像无效
MS_ERR_KERN_PROCESS_BAD_SYSCALL	-20005	错误的系统调用
MS_ERR_KERN_PROCESS_NO_MEM_REGION	-20006	进程内存区域不足
MS_ERR_KERN_PROCESS_NO_MEM	-20007	进程内存不足

宏	值	含义
MS_ERR_KERN_PROCESS_NO_EXIST	-20008	进程不存在
MS_ERR_KERN_APP_VERSION_NOT_COMP	-20009	APP 版本不兼容
MS_ERR_KERN_APP_HEAP_ALGO_NOT_COMP	-20010	APP 内存堆算法不兼容
MS_ERR_KERN_APP_TOOL_NOT_COMP	-20011	APP 生成工具不兼容
MS_ERR_KERN_APP_MACHINE_NOT_COMP	-20012	APP 机器不兼容
MS_ERR_KERN_APP_MAGIC_INVALID	-20013	APP 魔数无效
软件定时器		
MS_ERR_KERN_NO_TIMER	-21001	软件定时器不足
MS_ERR_KERN_TIMERID_INVALID	-21002	软件定时器 ID 无效
MS_ERR_KERN_TIMER_INVALID	-21003	软件定时器无效
MS_ERR_KERN_TIMER_OPT_INVALID	-21004	软件定时器选项无效
MS_ERR_KERN_TIMER_TIME_INVALID	-21005	软件定时器时间无效
读写锁		
MS_ERR_KERN_NO_RWLOCK	-22001	读写锁不足
MS_ERR_KERN_RWLOCKID_INVALID	-22002	读写锁 ID 无效
MS_ERR_KERN_RWLOCK_INVALID	-22003	读写锁无效
MS_ERR_KERN_RWLOCK_BUSY	-22004	读写锁忙（使用中）
MS_ERR_KERN_RWLOCK_WAIT_TIMEOUT	-22005	读写锁等待超时

宏	值	含义
MS_ERR_KERN_RWLOCK_WAIT_ABORT	-22006	读写锁等待终止
MS_ERR_KERN_RWLOCK_NOT_OWNER	-22007	不是读写锁的拥有者线程
MS_ERR_KERN_RWLOCK_NOT_LOCK	-22008	读写锁没有上锁
MS_ERR_KERN_RWLOCK_OVERFLOW	-22009	读写锁上锁次数溢出
MS_ERR_KERN_RWLOCK_DESTROY	-22010	读写锁被销毁
MS_ERR_KERN_RWLOCK_EMPTY	-22011	读写锁已被其它线程占有，尝试等待失败
IO 子系统		
MS_ERR_IO	-25001	IO 错误
MS_ERR_IO_EXISTED	-25002	IO 命名节点已存在
MS_ERR_IO_NO_EXISTED	-25003	IO 命名节点不存在
MS_ERR_IO_BUSY	-25004	IO 忙错误
MS-FLASHFS 文件系统		
MS_ERR_FLASHFS_FILE_NO_EXIST	-26001	文件不存在
MS_ERR_FLASHFS_FILE_CRC_FAIL	-26002	文件数据 CRC 错误
MS_ERR_FLASHFS_FILE_EXIST	-26003	文件已存在
MS_ERR_FLASHFS_NO_SPACE	-26004	没有足够磁盘空间
MS_ERR_FLASHFS_SOURCE_NO_EXIST	-26005	外部文件系统的源文件不存在
MS_ERR_FLASHFS_NO_MOUNT	-26006	没有挂载

宏	值	含义
MS_ERR_FLASHFS_NO_FORMAT	-26007	没有格式化
MS_ERR_FLASHFS_NO_INIT	-26008	没有初始化
MS_ERR_FLASHFS_FILE_BIG	-26009	文件太大
MS_ERR_FLASHFS_MOUNTED	-26010	已经挂载
MS_ERR_FLASHFS_DIR_END	-26011	目录流结束
MS_ERR_FLASHFS_ACTION_INVALID	-26012	动作无效
MS_ERR_FLASHFS_READONLY	-26013	文件系统只读
MS_ERR_FLASHFS_EXT_FS	-26014	外部文件系统错误
MS_ERR_FLASHFS_MKFS_PARAM	-26015	格式化参数错误
MS-LAUNCHER 启动器		
MS_ERR_LAUNCHER_NO_IMPLEMENT	-27000	MS-LAUNCHER 没有实现
MS_ERR_LAUNCHER_PATH_BAD	-27001	启动参数文件 path 错误
MS_ERR_LAUNCHER_BASE_BAD	-27002	启动参数文件 base 错误
MS_ERR_LAUNCHER_SIZE_BAD	-27003	启动参数文件 size 错误
MS_ERR_LAUNCHER_MPU_PROTECT_BAD	-27004	启动参数文件 mpu_protect 错误
MS_ERR_LAUNCHER_APP_NAME_BAD	-27005	启动参数文件 APP 名字错误
MS_ERR_LAUNCHER_IMAGE_FILE_BAD	-27006	启动参数文件 img_file 错误
MS_ERR_LAUNCHER_AUTO_START_BAD	-27007	启动参数文件 auto_start 错误

宏	值	含义
MS_ERR_LAUNCHER_MEM_SIZE_BAD	-27008	启动参数文件 mem_size 错误
MS_ERR_LAUNCHER_MAIN_STK_SIZE_BAD	-27009	启动参数文件 main_stk_size 错误
MS_ERR_LAUNCHER_MAIN_PRIO_BAD	-27010	启动参数文件 main_prio 错误
MS_ERR_LAUNCHER_MAIN_TIME_SLICE_BAD	-27011	启动参数文件 main_time_slice 错误
MS_ERR_LAUNCHER_CRASH_REBOOT_BAD	-27012	启动参数文件 crash_reboot 错误
MS_ERR_LAUNCHER_HIGHEST_PRIO_BAD	-27013	启动参数文件 highest_prio 错误
MS_ERR_LAUNCHER_LOG_LEVEL_BAD	-27014	启动参数文件 log_level 错误
MS_ERR_LAUNCHER_THREAD_MAX_BAD	-27015	启动参数文件 thread_max 错误
MS_ERR_LAUNCHER_EVENTSET_MAX_BAD	-27016	启动参数文件 eventset_max 错误
MS_ERR_LAUNCHER_MEMPOOL_MAX_BAD	-27017	启动参数文件 mempool_max 错误
MS_ERR_LAUNCHER_MQUEUE_MAX_BAD	-27018	启动参数文件 mqueue_max 错误
MS_ERR_LAUNCHER_MUTEX_MAX_BAD	-27019	启动参数文件 mutex_max 错误
MS_ERR_LAUNCHER_SEMC_MAX_BAD	-27020	启动参数文件 semc_max 错误
MS_ERR_LAUNCHER_RWLOCK_MAX_BAD	-27021	启动参数文件 rwlock_max 错误
MS_ERR_LAUNCHER_FILE_MAX_BAD	-27022	启动参数文件 file_max 错误
MS_ERR_LAUNCHER_FILE_PERM_BAD	-27023	启动参数文件 file_perm 错误
MS_ERR_LAUNCHER_EEPROM_W_PERM_BAD	-27024	启动参数文件 eeprom_w_perm 错误
MS_ERR_LAUNCHER_OP_PERM_BAD	-2702	启动参数文件 op_perm 错误

宏	值	含义
	5	
MS_ERR_LAUNCHER_APP_NO_EXIST	-27026	APP 不存在
MS_ERR_LAUNCHER_IPC_WAIT_TYPE_BAD	-27027	启动参数文件 wait_type 错误
MS_ERR_LAUNCHER_IPC_INIT_VALUE_BAD	-27028	启动参数文件 init_value 错误
MS_ERR_LAUNCHER_IPC_MAX_VALUE_BAD	-27029	启动参数文件 max_value 错误
MS_ERR_LAUNCHER_IPC_MEM_BASE_BAD	-27030	启动参数文件 mem_base 错误
MS_ERR_LAUNCHER_IPC_N_BLK_BAD	-27031	启动参数文件 n_blk 错误
MS_ERR_LAUNCHER_IPC_BLK_SIZE_BAD	-27032	启动参数文件 blk_size 错误
MS_ERR_LAUNCHER_IPC_MSG_BUF_BAD	-27033	启动参数文件 msg_buf 错误
MS_ERR_LAUNCHER_IPC_N_MSG_BAD	-27034	启动参数文件 n_msg 错误
MS_ERR_LAUNCHER_IPC_MSG_SIZE_BAD	-27035	启动参数文件 msg_size 错误
MS_ERR_LAUNCHER_MODULE_NAME_BAD	-27036	启动参数文件 MODULE 名字错误
MS_ERR_LAUNCHER_DIR_PERM_BAD	-27037	启动参数文件 dir_perm 错误
MS_ERR_LAUNCHER_APP_SIGNATURE_FAIL	-27038	APP 数字签名检查失败
UPDATE 更新		
MS_ERR_UPDATE_REQ_OPEN	-27041	更新请求文件打开失败
MS_ERR_UPDATE_REQ_NO_EXIST	-27042	更新请求文件不存在
MS_ERR_UPDATE_REQ_LEN	-27043	更新请求文件长度错误

宏	值	含义
MS_ERR_UPDATE_REQ_READ	-27044	读更新请求文件失败
MS_ERR_UPDATE_ACTION_OPEN	-27045	更新动作文件打开失败
MS_ERR_UPDATE_ACTION_NO_EXIST	-27046	更新动作文件不存在
MS_ERR_UPDATE_ACTION_LEN	-27047	更新动作文件长度错误
MS_ERR_UPDATE_ACTION_READ	-27048	读更新动作文件失败
MS-MODULE 内核模块		
MS_ERR_MODULE_FD_READ_FAILED	-28001	读文件失败
MS_ERR_MODULE_HEADER_INVALID	-28002	模块头无效
MS_ERR_MODULE_NO_DYN	-28003	没有 .dynamic section
MS_ERR_MODULE_NO_SYMTAB	-28004	没有 .symtab section
MS_ERR_MODULE_NO_STRTAB	-28005	没有 .strtab section
MS_ERR_MODULE_NO_TEXT	-28006	没有 .text section
MS_ERR_MODULE_NO_LOAD_SEGMENTS	-28007	没有需要加载的 segments
MS_ERR_MODULE_RELOCATE_FAILED	-28008	重定位失败
MS_ERR_MODULE_SECTION_NOT_FOUND	-28009	找不到 section
MS_ERR_MODULE_SYM_NOT_FOUND	-28010	找不到符号
MS_ERR_MODULE_TYPE_INVALID	-28011	重定位类型无效
VMM 虚拟内存管理		

宏	值	含义
MS_ERR_VMM_NO_PROCESS_SPACE	-29001	进程虚拟地址空间不足
MS_ERR_VMM_NO_USER_SPACE	-29002	用户态虚拟地址空间不足
MS_ERR_VMM_NO_DEV_SPACE	-29003	设备虚拟地址空间不足
MS_ERR_VMM_NO_MAP	-29004	虚拟地址没有映射
MPU 内存保护单元		
MS_ERR_MPU_NO_REGION	-30001	MPU 区域不足
MS_ERR_MPU_REGION_ID_INVALID	-30002	MPU 区域 ID 无效

posix 错误码

宏	值	含义
EPERM	1	操作不允许
ENOENT	2	没有这样的文件或目录
ESRCH	3	没有这样的过程
EINTR	4	系统调用被中断
EIO	5	I/O 错误
ENXIO	6	没有这样的设备或地址
E2BIG	7	参数列表太长
ENOEXEC	8	执行格式错误
EBADF	9	坏的文件描述符
ECHILD	10	没有子进程
EAGAIN	11	资源暂时不可用
ENOMEM	12	内存溢出
EACCES	13	拒绝许可
EFAULT	14	错误的地址

宏	值	含义
ENOTBLK	15	块设备请求
EBUSY	16	设备或资源忙
EEXIST	17	文件已存在
EXDEV	18	无效的交叉链接
ENODEV	19	设备不存在
ENOTDIR	20	不是一个目录
EISDIR	21	是一个目录
EINVAL	22	无效的参数
ENFILE	23	打开太多的文件系统
EMFILE	24	打开的文件过多
ENOTTY	25	不是 tty 设备
ETXTBSY	26	文本文件忙
EFBIG	27	文件太大
ENOSPC	28	设备上没有空间
ESPIPE	29	非法移位
EROFS	30	只读文件系统
EMLINK	31	太多的链接
EPIPE	32	管道破裂
EDOM	33	数值结果超出范围
ERANGE	34	数值结果不具代表性
ENOMSG	35	没有期望类型的消息
EIDRM	36	标识符删除
ECHRNG	37	频道数目超出范围
EL2NSYNC	38	2 级没有同步
EL3HLT	39	3 级中断
EL3RST	40	3 级复位
ELNRNG	41	链接编号超出范围
EUNATCH	42	协议驱动程序没有连接

宏	值	含义
ENOCSI	43	没有可用的 CSI 架构
EL2HLT	44	2 级中断
EDEADLK	45	资源死锁错误
ENOLCK	46	没有可用的锁
EBADE	50	无效的交换
EBADR	51	请求描述符无效
EXFULL	52	交换全
ENOANO	53	没有阳极
EBADRQC	54	无效的请求代码
EBADSLT	55	无效的槽
EDEADLOCK	56	和 EDEADLK 一样
EBFONT	57	错误的字体文件格式
ENOSTR	60	设备不是字符流
ENODATA	61	无可用数据
ETIME	62	计时器到期
ENOSR	63	流资源溢出
ENONET	64	机器不在网络上
ENOPKG	65	没有安装软件包
EREMOTE	66	对象是远程的
ENOLINK	67	联系被切断
EADV	68	广告错误
ESRMNT	69	srmount 错误
ECOMM	70	发送时的通讯错误
EPROTO	71	协议错误
EMULTIHOP	74	多跳尝试
ELBIN	75	Inode 是远程的 (不是实际的错误)
EDOTDOT	76	挂载点出现交叉 (不是实际的错误)
EBADMSG	77	坏消息

宏	值	含义
EFTYPE	79	不适当的文件类型或格式
ENOTUNIQ	80	不是唯一的 log 名称
EBADFD	81	非法的操作
EREMCHG	82	远程的地址改变了
ELIBACC	83	不能访问需要的共享库
ELIBBAD	84	访问一个被占用的共享库
ELIBSCN	85	a.out 中的 .lib 段被占用
ELIBMAX	86	尝试链接太多的库
ELIBEXEC	87	尝试执行一个共享库
ENOSYS	88	函数未实现
ENMFILE	89	没有更多的文件
ENOTEMPTY	90	目录不为空
ENAMETOOLONG	91	文件或路径名称过长
ELOOP	92	太多的符号链接
EOPNOTSUPP	95	socket 不支持的操作
EPFNOSUPPORT	96	协议簇不支持
ECONNRESET	104	链接由对方复位
ENOBUFS	105	没有足够的缓存空间
EAFNOSUPPORT	106	协议不支持的地址
EPROTOTYPE	107	socket 协议类型错误
ENOTSOCK	108	在非套接字上的套接字操作
ENOPROTOOPT	109	协议不可用
ESHUTDOWN	110	传输后无法发送
ECONNREFUSED	111	拒绝连接
EADDRINUSE	112	地址已经使用
ECONNABORTED	113	软件引起的连接中断
ENETUNREACH	114	网络不可达
ENETDOWN	115	网络瘫痪

宏	值	含义
ETIMEDOUT	116	连接超时
EHOSTDOWN	117	主机已关闭
EHOSTUNREACH	118	没有主机的路由
EINPROGRESS	119	正在运行
EALREADY	120	已运行
EDESTADDRREQ	121	需要目标地址
EMSGSIZE	122	消息太长
EPROTONOSUPPORT	123	不支持的协议
ESOCKTNOSUPPORT	124	套接字类型不支持
EADDRNOTAVAIL	125	地址无效
ENETRESET	126	网络连接中断
EISCONN	127	传输端点已经连接
ENOTCONN	128	传输端点没有连接
ETOOMANYREFS	129	太多的参考
EPROCLIM	130	太多的进程
EUSERS	131	太多的用户
EDQUOT	132	超出磁盘配额
ESTALE	133	陈旧的文件句柄
ENOTSUP	134	不支持
ENOMEDIUM	135	没有磁盘被发现
ENOSHARE	136	没有这个主机或网络通路
ECASECLASH	137	文件名已经存在
EILSEQ	138	非法的字节序
EOVERFLOW	139	文件类型的数值溢出
ECANCELED	140	取消操作
ENOTRECOVERABLE	141	状态不可恢复
EOWNERDEAD	142	之前的拥有者消亡
ESTRPIPE	143	流管道错误

宏	值	含义
EWOULDBLOCK	EAG	操作可能阻塞

3 MS-RTOS 内核接口

> 本章将介绍 MS-RTOS 内核接口的使用。

内核相关 API

下表展示了内核相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_rtos_version	●	●
ms_rtos_name	●	●
ms_rtos_license	●	●
ms_rtos_logo	●	●
ms_rtos_reboot	●	●
ms_rtos_shutdown	●	●
ms_rtos_update	●	●
ms_rtos_init		●
ms_rtos_start		●
ms_printk		●
ms_printk_set_level		●
ms_sched_lock		●
ms_sched_unlock		●
ms_schedule		●
ms_kern_tick		●
ms_access_ok		●
ms_op_perm_ok		●
ms_dev_perm_ok		●
ms_dir_perm_ok		●
ms_eeprom_write_ok		●
ms_arch_int_disable		●
ms_arch_int_resume		●

API	用户空间	内核空间
ms_arch_ffs		●
ms_arch_memcpy		●
ms_write32		●
ms_read32		●
ms_write16		●
ms_read16		●
ms_write8		●
ms_read8		●
ms_idle_hook_add		●
ms_idle_hook_remove		●

ms_rtos_version()

- **描述** 获得 MS-RTOS 版本号
- **函数原型**

```
ms_uint32_t ms_rtos_version(void);
```

- **参数** 无
- **返回值** MS-RTOS 版本号
- **注意事项** 无
- **示例**

```
int main(int argc, char *argv[])
{
    ms_uint32_t version = ms_rtos_version();

    // do some thing

    return 0;
}
```

ms_rtos_name()

- **描述** 获得 MS-RTOS 名字字符串
- **函数原型**

```
const char *ms_rtos_name(void);
```

- **参数** 无
- **返回值** MS-RTOS 名字字符串
- **注意事项** 无
- **示例**

```
int main(int argc, char *argv[])
{
    const char *name = ms_rtos_name();

    // do some thing

    return 0;
}
```

ms_rtos_license()

- **描述** 获得 MS-RTOS license 字符串
- **函数原型**

```
const char *ms_rtos_license(void);
```

- **参数** 无
- **返回值** MS-RTOS license 字符串
- **注意事项** 无
- **示例**

```
int main(int argc, char *argv[])
{
    const char *license = ms_rtos_license();

    // do some thing

    return 0;
}
```

ms_rtos_logo()

- **描述** 获得 MS-RTOS logo 字符串
- **函数原型**


```
const char *ms_rtos_logo(void);
```

- **参数** 无
- **返回值** MS-RTOS logo 字符串
- **注意事项** 无
- **示例**

```
int main(int argc, char *argv[])
{
    const char *logo = ms_rtos_logo();

    // do some thing

    return 0;
}
```

ms_rtos_reboot()

- **描述** 重启机器
- **函数原型**

```
ms_err_t ms_rtos_reboot(void);
```

- **参数** 无
- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
int main(int argc, char *argv[])
{
    ms_rtos_reboot();

    return 0;
}
```

ms_rtos_shutdown()

- **描述** 关机
- **函数原型**

```
ms_err_t ms_rtos_shutdown(void);
```

- **参数** 无
- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
int main(int argc, char *argv[])
{
    ms_rtos_shutdown();

    return 0;
}
```

ms_rtos_update()

- **描述** 更新 MS-RTOS OS 和 APP 镜像
- **函数原型**

```
ms_err_t ms_rtos_update(void);
```

- **参数** 无
- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
int main(int argc, char *argv[])
{
    ms_rtos_update();

    return 0;
}
```

ms_rtos_init()

- **描述** 初始化 MS-RTOS 内核和各种子系统
- **函数原型**

```
ms_err_t ms_rtos_init(ms_mem_layout_t *mem_layout);
```

- **参数**

输入/输出	参数	描述
[in]	mem_layout	内存布局类型指针

内存布局类型每一个成员代表哪一个内存区域已经被固定好：

宏	值	描述
MS_FLASH_REGION	0 U	FLASH 区域，进程可读、可执行，必须有效
MS_EXT_FLASH_REGION	1 U	扩展的 FLASH 区域，进程可读、可执行，可以没有
MS_KERN_TEXT_REGION	2 U	内核代码区域，进程不能访问，必须有效
MS_KERN_DATA_REGION	3 U	内核数据区域，进程不能访问，必须有效
MS_KERN_HEAP_REGION	4 U	内核内存堆区域，在内核数据区域里面，必须有效
MS_SHARED_RAM_REGION	5 U	共享内存区域，进程可读、可写、可执行，地址和大小可以为 0（即无效）
MS_PROCESS_MEM_REGION	6 U	进程内存区域，进程可读、可写，支持进程特性的 MS-RTOS 必须有效
MS_DEV_SPACE_REGION	7 U	设备虚拟地址空间区域（带有 MMU 的系统才需要）
MS_USER_SPACE_REGION	8 U	用户虚拟地址空间区域（带有 MMU 的系统才需要）

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在 MS-RTOS 启动后调用，不能重复调用，一般在 CPU 复位后初始化基本硬件设施后马上调用
- **示例** 请参考系统移植章节

ms_rtos_start()

- **描述** 启动 MS-RTOS
- **函数原型**

```
ms_err_t ms_rtos_start(void);
```

- **参数** 无
- **返回值** MS-RTOS 内核错误码

- **注意事项** 不能在中断中调用，在 MS-RTOS 初始化后调用，不能重复调用
- **示例** 请参考系统移植章节

ms_printk()

- **描述** 内核打印信息，信息通过 `ms_bsp_printk` 接口进行输出
- **函数原型**

```
void ms_printk(ms_pk_level_t level, const char *fmt, ...) MS_PRINTK_ATTR;
```

- **参数**

输入/输出	参数	描述
[in]	level	日志级别
[in]	fmt	格式化字符串

日志级别为以下的宏：

宏	值	描述
MS_PK_EMERG	0U	紧急事件消息，系统崩溃之前提示，表示系统不可用
MS_PK_ALERT	1U	报告消息，表示必须立即采取措施
MS_PK_CRIT	2U	临界条件，通常涉及严重的硬件或软件操作失败
MS_PK_ERR	3U	错误条件，驱动程序常用 MS_PK_ERR 来报告硬件的错误
MS_PK_WARNING	4U	警告条件，对可能出现问题的情况进行警告
MS_PK_NOTICE	5U	正常但又重要的条件，用于提醒
MS_PK_INFO	6U	提示信息，如驱动程序启动时，打印硬件信息
MS_PK_DEBUG	7U	调试级别的消息

- **返回值** 无
- **注意事项** 无
- **示例**

```
void xxx(void)
{
    ms_printk(MS_PK_DEBUG, "xxx %d\n", 1);

    // do some thing
}
```


ms_printk_set_level()

- **描述** 设置 ms_printk 函数可打印的日志级别（默认为 MS_PK_DEBUG）
- **函数原型**

```
ms_err_t ms_printk_set_level(ms_pk_level_t level);
```

- **参数**

输入/输出	参数	描述
[in]	level	可打印的日志级别

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
void xxx(void)
{
    ms_printk_set_level(MS_PK_NOTICE);

    // do some thing
}
```

ms_sched_lock()

- **描述** 锁住内核调度器
- **函数原型**

```
ms_err_t ms_sched_lock(void);
```

- **参数** 无
- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核启动前调用，与 ms_sched_unlock 配对使用
- **示例** 见 ms_sched_unlock()

ms_sched_unlock()

- **描述** 解锁内核调度器
- **函数原型**

```
ms_err_t ms_sched_unlock(void);
```

- **参数** 无
- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核启动前调用，与 `ms_sched_lock` 配对使用
- **示例**

```
void xxx(void)
{
    ms_sched_lock();

    // do some thing

    ms_sched_unlock();
}
```

ms_schedule()

- **描述** 进行一次线程调度
- **函数原型**

```
void ms_schedule(void);
```

- **参数** 无
- **返回值** 无
- **注意事项** 中断中、内核启动前、内核锁定期间调用并不会进行线程调度
- **示例**

```
void xxx(void)
{
    ms_schedule();
}
```

ms_kern_tick()

- **描述** 维护内核心跳时间
- **函数原型**

```
ms_err_t ms_kern_tick(ms_tick_t n_tick);
```

- **参数**

输入/输出	参数	描述
[in]	n_tick	经过了多少个嘀嗒

- **返回值** MS-RTOS 内核错误码
- **注意事项** 只能由硬件定时器中断服务函数或中断入口函数调用
- **示例**

如果使能了 ISR 管理 `MS_CFG_KERN_ISR_MANAGE_EN`，在硬件定时器中断服务函数中调用：

```
static ms_irq_ret_t timer_isr(ms_ptr arg)
{
    // do some thing

    ms_kern_tick(1U);

    return MS_IRQ_HANDLED;
}
```

如果没有使能 ISR 管理 `MS_CFG_KERN_ISR_MANAGE_EN`，在硬件定时器中断入口函数中调用：

```
void timer_int_handle(void)
{
    // enter interrupt
    ms_int_enter();

    // do some thing

    ms_kern_tick(1U);

    // exit interrupt
    ms_int_exit();
}
```

ms_access_ok()

- **描述** 判断当前线程指定的内存范围是否能够以指定的访问模式来访问
- **函数原型**

```
ms_bool_t ms_access_ok(ms_const_ptr_t addr, ms_size_t size, ms_access_mode_t mode);
```

- **参数**

输入/输出	参数	描述
[in]	addr	内存基地址

输入/输出	参数	描述
[in]	size	内存大小
[in]	mode	访问模式

访问模式为以下的宏：

宏	含义
MS_ACCESS_R	只读
MS_ACCESS_W	只写
MS_ACCESS_RW	读写

- **返回值** MS_TRUE：可以访问，MS_FALSE：不能访问
- **注意事项** 无
- **示例**

```
void xxx(void)
{
    ms_bool_t ret = ms_access_ok(0xc0000000, 4, MS_ACCESS_R);

    // do some thing
}
```

ms_op_perm_ok()

- **描述** 判断当前线程是否有指定的操作权限
- **函数原型**

```
ms_bool_t ms_op_perm_ok(ms_op_perm_t op_perm);
```

- **参数**

输入/输出	参数	描述
[in]	op_perm	操作权限

操作权限为以下的宏：

宏	含义
MS_OP_PERM_REBOOT	重启、关机
MS_OP_PERM_UPDATE	升级

宏	含义
MS_OP_PERM_TIME_SET	设置时间
MS_OP_PERM_APP_START	启动 APP
MS_OP_PERM_APP_KILL	杀死 APP
MS_OP_PERM_APP_SIGNAL	给 APP 发信号

- **返回值** MS_TRUE：有权限，MS_FALSE：没权限
- **注意事项** 无
- **示例**

```
void xxx(void)
{
    ms_bool_t ret = ms_op_perm_ok(MS_OP_PERM_TIME_SET);

    // do some thing
}
```

ms_dev_perm_ok()

- **描述** 判断当前线程是否有指定设备的指定读写权限
- **函数原型**

```
ms_bool_t ms_dev_perm_ok(const char *path, ms_dev_perm_t perm);
```

- **参数**

输入/输出	参数	描述
[in]	path	设备路径
[in]	perm	设备读写权限

设备读写权限为以下的宏：

宏	含义
MS_DEV_PERM_R	读设备
MS_DEV_PERM_W	写设备
MS_DEV_PERM_RW	读写设备

- **返回值** MS_TRUE：有权限，MS_FALSE：没权限

- **注意事项** 无
- **示例**

```
void xxx(void)
{
    ms_bool_t ret = ms_dev_perm_ok("/dev/xxx", MS_DEV_PERM_W);

    // do some thing
}
```

ms_dir_perm_ok()

- **描述** 判断当前线程是否有指定目录的访问权限
- **函数原型**

```
ms_bool_t ms_dir_perm_ok(const char *path);
```

- **参数**

输入/输出	参数	描述
[in]	path	目录路径

- **返回值** MS_TRUE：有权限，MS_FALSE：没权限
- **注意事项** 无
- **示例**

```
void xxx(void)
{
    ms_bool_t ret = ms_dir_perm_ok("/nor/update");

    // do some thing
}
```

ms_eeprom_write_ok()

- **描述** 判断当前线程是否有指定的 EEPROM 空间的写权限
- **函数原型**

```
ms_bool_t ms_eeprom_write_ok(ms_uint32_t addr, ms_size_t len);
```

- **参数**

输入/输出	参数	描述
[in]	addr	EEPROM 空间的起始地址
[in]	len	EEPROM 空间的长度

- **返回值** MS_TRUE：有权限，MS_FALSE：没权限
- **注意事项** 无
- **示例**

```
void xxx(void)
{
    ms_bool_t ret = ms_eeprom_write_ok(0,  10);

    // do some thing
}
```

ms_arch_int_disable()

- **描述** 屏蔽 CPU 中断
- **函数原型**

```
ms_arch_sr_t ms_arch_int_disable(void);
```

- **参数** 无
- **返回值** 屏蔽前的 CPU 中断使能状态
- **注意事项** 无
- **示例** 见 `ms_arch_int_resume`

ms_arch_int_resume()

- **描述** 恢复 CPU 中断使能状态
- **函数原型**

```
void ms_arch_int_resume(ms_arch_sr_t sr);
```

- **参数**

输入/输出	参数	描述
[in]	sr	CPU 中断使能状态（由 ms_arch_int_disable 返回）

- **返回值** 无
- **注意事项** 无

- **示例**

```
void xxx(void)
{
    ms_arch_sr_t sr = ms_arch_int_disable();

    // do some thing

    ms_arch_int_resume(sr);
}
```

ms_arch_ffs()

- **描述** 查找无符号 32 位数中第一个置 1 的位
- **函数原型**

```
ms_uint32_t ms_arch_ffs(ms_uint32_t value);
```

- **参数**

输入/输出	参数	描述
[in]	value	需要查找的无符号 32 位数

- **返回值** 第一个置 1 的位的编号，正确返回 [1 ~ 32]，如果没有位被置 1，返回 0
- **注意事项** 无
- **示例**

```
void xxx(void)
{
    ms_uint32_t bit_no = ms_arch_ffs(0x00010000);

    // do some thing
}
```

ms_arch_memcpy()

- **描述** 内存拷贝
- **函数原型**

```
void ms_arch_memcpy(ms_ptr_t dest, ms_const_ptr_t src, ms_size_t size);
```

- **参数**

输入/输出	参数	描述
[in]	dest	目的内存地址
[in]	src	源内存地址
[in]	size	需要拷贝的长度

- **返回值** 无
- **注意事项** 无
- **示例**

```
void xxx(void)
{
    ms_uint8_t  buf_src[100];
    ms_uint8_t  buf_dst[100];

    ms_arch_memcpy(buf_dst, buf_src, sizeof(buf_src));

    // do some thing
}
```

ms_write32()

- **描述** 写 32 位数据到指定地址
- **函数原型**

```
void ms_write32(ms_uint32_t data, ms_addr_t addr)
```

- **参数**

输入/输出	参数	描述
[in]	data	数据
[in]	addr	地址

- **返回值** 无
- **注意事项** 无
- **示例**

```
void xxx(void)
{
    ms_write32(0x00010000, 0xc0000000);
```

```
    // do some thing
}
```

ms_read32()

- **描述** 从指定地址读 32 位数据
- **函数原型**

```
ms_uint32_t ms_read32(ms_addr_t addr)
```

- **参数**

输入/输出	参数	描述
[in]	addr	地址

- **返回值** 32 位数据
- **注意事项** 无
- **示例**

```
void xxx(void)
{
    ms_uint32_t val = ms_read32(0xc0000000);

    // do some thing
}
```

ms_write16()

- **描述** 写 16 位数据到指定地址
- **函数原型**

```
void ms_write16(ms_uint16_t data, ms_addr_t addr)
```

- **参数**

输入/输出	参数	描述
[in]	data	数据
[in]	addr	地址

- **返回值** 无
- **注意事项** 无

- **示例**

```
void xxx(void)
{
    ms_write16(0x0001, 0xc0000000);

    // do some thing
}
```

ms_read16()

- **描述** 从指定地址读 16 位数据
- **函数原型**

```
ms_uint16_t ms_read16(ms_addr_t addr)
```

- **参数**

输入/输出	参数	描述
[in]	addr	地址

- **返回值** 16 位数据
- **注意事项** 无
- **示例**

```
void xxx(void)
{
    ms_uint16_t val = ms_read16(0xc0000000);

    // do some thing
}
```

ms_write8()

- **描述** 写 8 位数据到指定地址
- **函数原型**

```
void ms_write8(ms_uint8_t data, ms_addr_t addr)
```

- **参数**

输入/输出	参数	描述
[in]	data	数据
[in]	addr	地址

- **返回值** 无
- **注意事项** 无
- **示例**

```
void xxx(void)
{
    ms_write8(0x01, 0xc0000000);

    // do some thing
}
```

ms_read8()

- **描述** 从指定地址读 8 位数据
- **函数原型**

```
ms_uint8_t ms_read8(ms_addr_t addr)
```

- **参数**

输入/输出	参数	描述
[in]	addr	地址

- **返回值** 8 位数据
- **注意事项** 无
- **示例**

```
void xxx(void)
{
    ms_uint8_t val = ms_read8(0xc0000000);

    // do some thing
}
```

ms_idle_hook_add()

- **描述** 添加一个 idle 线程勾子函数
- **函数原型**

```
ms_err_t ms_idle_hook_add(ms_idle_hook_t *hook);
```

- **参数**

输入/输出	参数	描述
[in]	hook	idle 线程勾子指针

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
static ms_idle_hook_t myhook;

static void my_idle_hook(ms_ptr_t arg)
{
    // do something
}

void xxx(void)
{
    myhook.callback = my_idle_hook;
    myhook.arg      = MS_NULL;

    ms_idle_hook_add(&myhook);
}
```

ms_idle_hook_remove()

- **描述** 移除一个 idle 线程勾子函数
- **函数原型**

```
ms_err_t ms_idle_hook_remove(ms_idle_hook_t *hook);
```

- **参数**

输入/输出	参数	描述
[in]	hook	idle 线程勾子指针

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**


```
void xxx(void)
{
    ms_idle_hook_remove(&myhook);
}
```


4 MS-RTOS 中断管理

> 本章将介绍 MS-RTOS 中断管理接口的使用。

中断相关数据类型

类型	描述
ms_irq_t	16 位中断号类型
ms_isr_ret_t	中断服务函数、中断底半部工作函数返回值类型
ms_isr_t	中断服务函数、中断底半部工作函数类型

ms_isr_ret_t

中断服务函数、中断底半部工作函数返回值类型为以下的宏：

宏	值	描述
MS_IRQ_FAILED_DIS	-2	中断处理失败，请求关闭中断
MS_IRQ_HANDLED_DIS	-1	中断处理成功，请求关闭中断
MS_IRQ_HANDLED	0	中断处理成功
MS_IRQ_FAILED	1	中断处理失败

ms_isr_t

中断服务函数、中断底半部工作函数类型：

```
typedef ms_isr_ret_t (*ms_isr_t)(ms_ptr_t arg);
```

以下是一个中断服务函数、中断底半部工作函数定义的示例：

```
static ms_isr_ret_t xxx_isr(ms_ptr_t arg)
{
    // do some thing
    return MS_IRQ_HANDLED;
}
```

中断相关 API

下表展示了中断管理相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_int_enter		●
ms_int_exit		●
ms_int_nesting		●
ms_int_enable		●
ms_int_disable		●
ms_int_is_enable		●
ms_int_install		●
ms_int_uninstall		●
ms_int_handle		●
ms_int_add_tasklet		●

ms_int_enter()

- **描述** 中断进入
- **函数原型**

```
ms_err_t ms_int_enter(void);
```

- **参数** 无
- **返回值** MS-RTOS 内核错误码
- **注意事项** 只能在中断入口函数中调用，需要与 `ms_int_exit` 配对使用
- **示例** 见 `ms_int_exit()`

ms_int_exit()

- **描述** 中断退出
- **函数原型**

```
ms_err_t ms_int_exit(void);
```

- **参数** 无
- **返回值** MS-RTOS 内核错误码
- **注意事项** 只能在中断入口函数中调用，需要与 `ms_int_enter` 配对使用
- **示例**


```
void xxx_int_handle(void)
{
    // enter interrupt
    ms_int_enter();

    // do some thing

    // exit interrupt
    ms_int_exit();
}
```

ms_int_nesting()

- **描述** 获得中断嵌套的层次
- **函数原型**

```
ms_uint8_t ms_int_nesting(void);
```

- **参数** 无
- **返回值** 中断嵌套的层次
- **注意事项** 在中断入口函数或中断服务函数中调用才有意义
- **示例**

```
void xxx_int_handle(void)
{
    ms_uint8_t level;

    // enter interrupt
    ms_int_enter();

    level = ms_int_nesting();

    // do some thing

    // exit interrupt
    ms_int_exit();
}
```

ms_int_enable()

- **描述** 使能指定的中断
- **函数原型**

```
ms_err_t ms_int_enable(ms_irq_t irq);
```


- **参数**

输入/输出	参数	描述
[in]	irq	中断号

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
void xxx(void)
{
    // do some thing
    ms_int_enable(10U);
    // do some thing
}
```

ms_int_disable()

- **描述** 屏蔽指定的中断
- **函数原型**

```
ms_err_t ms_int_disable(ms_irq_t irq);
```

- **参数**

输入/输出	参数	描述
[in]	irq	中断号

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
void xxx(void)
{
    // do some thing
    ms_int_disable(10U);
    // do some thing
}
```

ms_int_is_enable()

- **描述** 判断指定的中断是否使能
- **函数原型**


```
ms_err_t ms_int_is_enable(ms_irq_t irq, ms_bool_t *penable);
```

- **参数**

输入/输出	参数	描述
[in]	irq	中断号
[out]	penable	中断是否使能

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
void xxx(void)
{
    ms_bool_t enable;

    ms_int_is_enable(10U, &enable);

    // do some thing
}
```

ms_int_install()

- **描述** 为指定的中断安装中断服务函数
- **函数原型**

```
ms_err_t ms_int_install(ms_irq_t irq, ms_isr_t isr, ms_ptr_t arg);
```

- **参数**

输入/输出	参数	描述
[in]	irq	中断号
[in]	isr	中断服务函数指针，不能为空指针
[in]	arg	中断服务函数参数，可以为空指针

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在内核初始化前调用
- **示例**


```
static ms_isr_ret_t xxx_isr(ms_ptr arg)
{
    // do some thing
    // clear interrupt pend

    return MS_IRQ_HANDLED;
}

void xxx(void)
{
    // do some thing

    ms_int_install(10U, xxx_isr, MS_NULL);
    ms_int_enable(10U);

    // do some thing
}
```

ms_int_uninstall()

- **描述** 卸载指定中断的中断服务函数
- **函数原型**

```
ms_err_t ms_int_uninstall(ms_irq_t irq);
```

- **参数**

输入/输出	参数	描述
[in]	irq	中断号

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在内核初始化前调用
- **示例**

```
void xxx(void)
{
    // do some thing

    ms_int_disable(10U);
    ms_int_uninstall(10U);

    // do some thing
}
```

ms_int_handle()

- **描述** 响应指定的中断，调用指定中断的中断服务函数
- **函数原型**

```
ms_err_t ms_int_handle(ms_irq_t irq);
```

- **参数**

输入/输出	参数	描述
[in]	irq	中断号

- **返回值** MS-RTOS 内核错误码
- **注意事项** 只能在 CPU 的中断入口函数中调用
- **示例**

```
void int_ctrl_int_handle(void)
{
    ms_int32_t irq;

    // enter interrupt
    ms_int_enter();

    // process all interrupt request
    while ((irq = int_ctrl_get_request()) >= 0) {
        ms_int_handle(irq);
    }

    // exit interrupt
    ms_int_exit();
}
```

ms_int_add_tasklet()

- **描述** 为指定的中断增加一个中断底半部工作函数
- **函数原型**

```
ms_err_t ms_int_add_tasklet(ms_irq_t irq, ms_isr_t job, ms_ptr_t arg);
```

- **参数**

输入/输出	参数	描述
[in]	irq	中断号
[in]	job	中断底半部工作函数指针，不能为空指针

输入/输出	参数	描述
[in]	arg	中断底半部工作函数参数，可以为空指针

- **返回值** MS-RTOS 内核错误码
- **注意事项** 只能在中断入口函数或中断服务函数中调用
- **示例**

```
static ms_isr_ret_t xxx_isr_tasklet(ms_ptr arg)
{
    // do some thing
    // clear interrupt pend

    // will enable irq
    return MS_IRQ_HANDLED;
}

static ms_isr_ret_t xxx_isr(ms_ptr arg)
{
    // do some thing

    // add a tasklet
    ms_int_add_tasklet(10U, xxx_isr_tasklet, MS_NULL);

    // will disable irq
    return MS_IRQ_HANDLED_DIS;
}

void xxx(void)
{
    // do some thing

    ms_int_install(10U, xxx_isr, MS_NULL);
    ms_int_enable(10U);

     // do some thing
}
```

5 MS-RTOS 时间管理

> 本章将介绍 MS-RTOS 时间管理接口的使用。

时间相关类型

类型	描述
ms_timeval_t	时间变量类型（1970 年 1 月 1 日 0 时 0 分 0 秒到现在的时间）
struct timeval	posix 时间变量类型（1970 年 1 月 1 日 0 时 0 分 0 秒到现在的时间）
ms_tms_t	tms 类型
struct tms	posix tms 类型
ms_tick_t	32 位嘀嗒类型，一般用于表示等待的超时时间
ms_tick64_t	64 位嘀嗒类型，一般用于表示内核心跳时间
ms_time_slice_t	16 位的时间片类型，以嘀嗒为单位，有效范围 0 - 65534，0 为同优先级先来先服务调度策略，MS_TIME_SLICE_INVALID 为无效的时间片
ms_rtc_time_t	RTC 时间类型
time_t	1970 年 1 月 1 日 0 时 0 分 0 秒到现在的秒数

ms_timeval_t 与 struct timeval

时间变量类型：

```
typedef struct timeval {
    time_t      tv_sec;     /* seconds */
    suseconds_t tv_usec;    /* and microseconds */
} ms_timeval_t;
```

参数	说明
tv_sec	秒数
tv_usec	微秒数

ms_tms_t 与 struct tms

时间类型：

```
typedef struct tms {
    clock_t tms_utime;      /* user time */
    clock_t tms_stime;      /* system time */
    clock_t tms_cutime;     /* user time, children */
    clock_t tms_cstime;     /* system time, children */
} ms_tms_t;
```

参数	说明
tms_utime	在执行用户空间代码上的时间总量
tms_stime	在执行内核空间代码上的时间总量
tms_cutime	子进程在执行用户空间代码上的时间总量
tms_cstime	子进程在执行内核空间代码上的时间总量

ms_tick_t

32 位嘀嗒类型一般用于表示等待的超时时间，等待的超时时间除了可以使用数值外，也可以使用以下的宏：

宏	值	含义
MS_TIMEOUT_NO_WAIT	0	不等待
MS_TIMEOUT_FOREVER	4294967295	一直等待

一个嘀嗒的时间长度取决于 `MS_CFG_KERN_TICK_HZ` 配置项，如果 `MS_CFG_KERN_TICK_HZ` 为 100，则一个嘀嗒为 10 毫秒，如果 `MS_CFG_KERN_TICK_HZ` 为 1000，则一个嘀嗒为 1 毫秒。

ms_rtc_time_t

RTC 时间类型：

```
typedef struct {
#define MS_RTC_YEAR_BASE            2000U
    ms_uint8_t  year;       /* base 2000 */

#define MS_RTC_MONTH_JANUARY        1U
#define MS_RTC_MONTH_FEBRUARY       2U
#define MS_RTC_MONTH_MARCH          3U
#define MS_RTC_MONTH_APRIL          4U
#define MS_RTC_MONTH_MAY            5U
#define MS_RTC_MONTH_JUNE           6U
#define MS_RTC_MONTH_JULY           7U
#define MS_RTC_MONTH_AUGUST         8U
#define MS_RTC_MONTH_SEPTEMBER      9U
#define MS_RTC_MONTH_OCTOBER        10U
#define MS_RTC_MONTH_NOVEMBER       11U
```

```
#define MS_RTC_MONTH_DECEMBER       12U
    ms_uint8_t  month;

    ms_uint8_t  date;       /* 1-31 */

#define MS_RTC_WEEKDAY_MONDAY       1U
#define MS_RTC_WEEKDAY_TUESDAY      2U
#define MS_RTC_WEEKDAY_WEDNESDAY    3U
#define MS_RTC_WEEKDAY_THURSDAY     4U
#define MS_RTC_WEEKDAY_FRIDAY       5U
#define MS_RTC_WEEKDAY_SATURDAY     6U
#define MS_RTC_WEEKDAY_SUNDAY       7U
    ms_uint8_t  weekday;

    ms_uint8_t  hour;       /* 0-23 */
    ms_uint8_t  minute;     /* 0-59 */
    ms_uint8_t  second;     /* 0-59 */
} ms_rtc_time_t;
```

参数	说明
year	年，基于公元 2000 年，即 0 表示公元 2000 年，最大到公元 2255 年
month	月， 1~12
date	日， 1~31
weekday	星期几，1~7
hour	时，0-23
minute	分，0~59
second	秒，0~59

时间管理相关 API

下表展示了时间管理相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_time_get	●	●
ms_time_get_ms	●	●
ms_time_set		●
ms_time_tick_to_ms	●	●
ms_time_ms_to_tick	●	●
ms_time_tick_hz	●	●

API	用户空间	内核空间
ms_gettimeofday	●	●
ms_settimeofday	●	●
ms_is_leap_year	●	●
ms_hms_time_valid	●	●
ms_weekday_valid	●	●
ms_weekday_calc	●	●
ms_weekday_string	●	●
ms_rtc_time_valid	●	●
ms_rtc_time_to_timeval	●	●
ms_timeval_to_rtc_time	●	●

ms_time_get()

- **描述** 获得内核心跳时间(以嘀嗒为单位)
- **函数原型**

```
ms_tick64_t ms_time_get(void);
```

- **参数** 无
- **返回值** 内核心跳时间(以嘀嗒为单位)
- **注意事项** 无
- **示例**

```
int main(int argc, char *argv[])
{
    ms_tick64_t time_tick = ms_time_get();

    // do some thing

    return 0;
}
```

ms_time_get_ms()

- **描述** 获得内核心跳时间(以毫秒为单位)
- **函数原型**

```
ms_uint64_t ms_time_get_ms(void);
```

- **参数** 无
- **返回值** 内核心跳时间(以毫秒为单位)
- **注意事项** 无
- **示例**

```
int main(int argc, char *argv[])
{
    ms_uint64_t time_ms = ms_time_get_ms();

    // do some thing

    return 0;
}
```

ms_time_set()

- **描述** 设置内核心跳时间(以嘀嗒为单位)
- **函数原型**

```
void ms_time_set(ms_tick64_t tick);
```

- **参数**

输入/输出	参数	描述
[in]	tick	内核心跳时间(以嘀嗒为单位)

- **返回值** 无
- **注意事项** 无
- **示例**

```
void xxx(void)
{
    ms_time_set(600U);
}
```

ms_time_tick_to_ms()

- **描述** 嘀嗒数转毫秒数
- **函数原型**

```
ms_uint32_t ms_time_tick_to_ms(ms_tick_t tick);
```

- **参数**

输入/输出	参数	描述
[in]	tick	嘀嗒数

- **返回值** 毫秒数
- **注意事项** 无
- **示例**

```
int main(int argc, char *argv[])
{
    ms_uint32_t ms = ms_time_tick_to_ms(100U);

    // do some thing

    return 0;
}
```

ms_time_ms_to_tick()

- **描述** 毫秒数转嘀嗒数
- **函数原型**

```
ms_tick_t ms_time_ms_to_tick(ms_uint32_t ms);
```

- **参数**

输入/输出	参数	描述
[in]	ms	毫秒数

- **返回值** 嘀嗒数
- **注意事项** 无
- **示例**

```
int main(int argc, char *argv[])
{
    ms_tick_t tick = ms_time_ms_to_tick(100U);

    // do some thing
```

```
    return 0;
}
```

ms_time_tick_hz()

- **描述** 获得内核心跳的频率
- **函数原型**

```
ms_uint16_t ms_time_tick_hz(void);
```

- **参数** 无
- **返回值** 内核心跳的频率
- **注意事项** 无
- **示例**

```
int main(int argc, char *argv[])
{
    ms_uint16_t tick_hz = ms_time_tick_hz();

    // do some thing

    return 0;
}
```

ms_gettimeofday()

- **描述** 获得当前精确时间（1970 年 1 月 1 日到现在的时间）
- **函数原型**

```
ms_err_t ms_gettimeofday(ms_timeval_t *tv);
```

- **参数**

输入/输出	参数	描述
[out]	tv	当前精确时间

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
int main(int argc, char *argv[])
{
```

```
    ms_timeval_t tv;

    ms_gettimeofday(&tv);

    // do some thing

    return 0;
}
```

ms_settimeofday()

- **描述** 设置当前精确时间（1970 年 1 月 1 日到现在的时间）
- **函数原型**

```
ms_err_t ms_settimeofday(const ms_timeval_t *tv);
```

- **参数**

输入/输出	参数	描述
[in]	tv	当前精确时间

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
int main(int argc, char *argv[])
{
    ms_timeval_t tv;

    // do some thing

    ms_settimeofday(&tv);

    return 0;
}
```

ms_is_leap_year()

- **描述** 判断输入的年是否为闰年
- **函数原型**

```
ms_bool_t ms_is_leap_year(ms_uint16_t year);
```

- **参数**

输入/输出	参数	描述
[in]	year	年，基于 0

- **返回值** MS_TRUE：是闰年，MS_FALSE：不是闰年
- **注意事项** 无
- **示例**

```
int main(int argc, char *argv[])
{
    ms_is_leap_year(2000U);

    return 0;
}
```

ms_hms_time_valid()

- **描述** 判断输入的时分秒是否有效
- **函数原型**

```
ms_bool_t ms_hms_time_valid(ms_uint8_t hour, ms_uint8_t min, ms_uint8_t sec);
```

- **参数**

输入/输出	参数	描述
[in]	hour	时
[in]	min	分
[in]	sec	秒

- **返回值** MS_TRUE：有效，MS_FALSE：无效
- **注意事项** 无
- **示例**

```
int main(int argc, char *argv[])
{
    ms_hms_time_valid(10U, 10U, 10U);

    return 0;
}
```

ms_weekday_valid()

- **描述** 判断输入的星期几是否为有效
- **函数原型**

```
ms_bool_t ms_weekday_valid(ms_uint8_t weekday);
```

- **参数**

输入/输出	参数	描述
[in]	weekday	星期几数值

- **返回值** MS_TRUE：有效，MS_FALSE：无效
- **注意事项** 无
- **示例**

```
int main(int argc, char *argv[])
{
    ms_weekday_valid(1U);

    return 0;
}
```

ms_weekday_calc()

- **描述** 计算某天是星期几
- **函数原型**

```
ms_uint8_t ms_weekday_calc(ms_uint16_t year, ms_uint8_t month, ms_uint8_t date);
```

- **参数**

输入/输出	参数	描述
[in]	year	年，基于 0，必须 >= 2000
[in]	month	月
[in]	date	日

- **返回值** 日期对应的星期几
- **注意事项** 无
- **示例**


```
int main(int argc, char *argv[])
{
    ms_weekday_calc(2000U, 1U, 1U);

    return 0;
}
```

ms_weekday_string()

- **描述** 获得星期几对应的字符串
- **函数原型**

```
const char *ms_weekday_string(ms_uint8_t weekday);
```

- **参数**

输入/输出	参数	描述
[in]	weekday	星期几

- **返回值** 星期几对应的字符串
- **注意事项** 无
- **示例**

```
int main(int argc, char *argv[])
{
    ms_printf(ms_weekday_string(1U));

    return 0;
}
```

ms_rtc_time_valid()

- **描述** 判断输入的 RTC 时间是否有效
- **函数原型**

```
ms_bool_t ms_rtc_time_valid(const ms_rtc_time_t *rtc_time);
```

- **参数**

输入/输出	参数	描述
[in]	rtc_time	RTC 时间

- **返回值** MS_TRUE：有效，MS_FALSE：无效
- **注意事项** 无
- **示例** 无

ms_rtc_time_to_timeval()

- **描述** 将 RTC 时间转换为 timeval 结构时间
- **函数原型**

```
ms_err_t ms_rtc_time_to_timeval(const ms_rtc_time_t *rtc_time, ms_timeval_t *tv);
```

- **参数**

输入/输出	参数	描述
[in]	rtc_time	RTC 时间
[out]	tv	timeval 结构时间

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_timeval_to_rtc_time()

- **描述** 将 timeval 结构时间转换为 RTC 时间
- **函数原型**

```
ms_err_t ms_timeval_to_rtc_time(const ms_timeval_t *tv, ms_rtc_time_t *rtc_time);
```

- **参数**

输入/输出	参数	描述
[in]	tv	timeval 结构时间
[out]	rtc_time	RTC 时间

- **返回值** MS-RTOS 内核错误码
- **注意事项** timeval 结构时间表示的时间必须大于 2000 年 1 月 1 日 0 时 0 分 0 秒
- **示例** 无

6 MS-RTOS 线程

> 本章将介绍 MS-RTOS 线程相关接口的使用。

线程相关数据类型

类型	描述
ms_thread_opt_t	线程选项类型
ms_thread_status_t	线程运行状态类型
ms_thread_entry_t	线程入口函数类型
ms_thread_stat_t	线程状态信息结构类型

ms_thread_opt_t

线程的选项为以下宏的组合：

宏	含义
MS_THREAD_OPT_SUPER	特权态的线程
MS_THREAD_OPT_USER	用户态的线程
MS_THREAD_OPT_FPU_EN	线程使能 FPU
MS_THREAD_OPT_REENT_EN	线程使能 C 库可重入结构

MS_THREAD_OPT_SUPER 与 MS_THREAD_OPT_USER 互斥使用

ms_thread_status_t

线程的运行状态为以下的宏：

宏	描述
MS_THREAD_STATUS_INVALID	无效的状态
MS_THREAD_STATUS_DEAD	僵死态
MS_THREAD_STATUS_READY	就绪态
MS_THREAD_STATUS_SUSPEND	挂起态
MS_THREAD_STATUS_SLEEP	休眠态
MS_THREAD_STATUS_WAIT_SEMC	等待计数信号量状态
MS_THREAD_STATUS_WAIT_MUTEX	等待互斥量状态

宏	描述
MS_THREAD_STATUS_WAIT_MQUEUE	等待消息队列状态
MS_THREAD_STATUS_WAIT_MEMPOOL	等待内存池状态
MS_THREAD_STATUS_WAIT_EVENTSET	等待事件标志组状态
MS_THREAD_STATUS_WAIT_COND	等待条件变量状态
MS_THREAD_STATUS_WAIT_RWLOCK_R	等待读写锁可读状态
MS_THREAD_STATUS_WAIT_RWLOCK_W	等待读写锁可写状态

如果线程以超时的方式等待一个线程间通信 IPC 对象，则线程的运行状态为 `MS_THREAD_STATUS_WAIT_???` | `MS_THREAD_STATUS_SLEEP`

ms_thread_entry_t

线程入口函数类型无返回值，有一个 `ms_ptr_t` 类型参数：

```
typedef void (*ms_thread_entry_t)(ms_ptr_t arg);
```

以下是一个线程的入口函数定义的示例：

```
static void xxx_thread(ms_ptr_t arg)
{
    // do some thing
}
```

ms_thread_stat_t

线程的状态信息结构类型：

```
typedef struct {
    const char *       name;
    ms_thread_status_t status;
    ms_time_slice_t    time_slice;
    ms_prio_t          prio;
} ms_thread_stat_t;
```

参数	说明
name	线程的名字
status	线程的运行状态
time_slice	线程的时间片（创建时指定的值）

参数	说明
prio	线程的优先级

线程相关 API

下表展示了线程相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_thread_create	●	●
ms_thread_init	●	●
ms_thread_self	●	●
ms_thread_kill	●	●
ms_thread_exit	●	●
ms_thread_suspend	●	●
ms_thread_resume	●	●
ms_thread_yield	●	●
ms_thread_sleep	●	●
ms_thread_sleep_ms	●	●
ms_thread_sleep_s	●	●
ms_thread_sleep_hms	●	●
ms_thread_stat	●	●
ms_thread_errno_ptr	●	●
ms_thread_get_errno	●	●
ms_thread_set_errno	●	●

ms_thread_create()

- **描述** 创建一个就绪态的线程
- **函数原型**

```
ms_err_t ms_thread_create(const char *name, ms_thread_entry_t entry, ms_ptr_t arg,
                          ms_size_t stk_size, ms_prio_t prio, ms_time_slice_t time_slice,
                          ms_thread_opt_t opt, ms_handle_t *tid);
```

- **参数**

输入/输出	参数	描述
[in]	name	线程的名字，不能为空指针
[in]	entry	线程的进入点函数
[in]	arg	线程的进入点函数的参数
[in]	stk_size	线程的堆栈空间大小
[in]	prio	线程的优先级
[in]	time_slice	线程的时间片
[in]	opt	线程的选项
[out]	tid	线程的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核锁定期间调用。 如果一个线程创建时没有指定使用 FPU，但运行过程中去执行 FPU 指令，将产生异常。 用户空间创建线程时将自动加入 `MS_THREAD_OPT_USER` 选项。
- **示例**

```
static ms_handle_t test_tid;

static void test_thread(void *arg)
{
    // do some thing
}

int main(int argc, char *argv[])
{
    ms_thread_create("test", test_thread, MS_NULL,
                     4096U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test_tid);
    ms_thread_sleep_s(1);

    return 0;
}
```

ms_thread_init()

- **描述** 创建一个暂停态的线程
- **函数原型**

```
ms_err_t ms_thread_init(const char *name, ms_thread_entry_t entry, ms_ptr_t arg,
                        ms_size_t stk_size, ms_prio_t prio, ms_time_slice_t time_slice,
                        ms_thread_opt_t opt, ms_handle_t *tid);
```

- **参数**

输入/输出	参数	描述
[in]	name	线程的名字，不能为空指针
[in]	entry	线程的进入点函数
[in]	arg	线程的进入点函数的参数
[in]	stk_size	线程的堆栈空间大小
[in]	prio	线程的优先级
[in]	time_slice	线程的时间片
[in]	opt	线程的选项
[out]	tid	线程的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核锁定期间调用
- **示例**

```
static ms_handle_t test_tid;

static void test_thread(void *arg)
{
    // do some thing
}

int main(int argc, char *argv[])
{
    ms_thread_init("test", test_thread, MS_NULL,
                   4096U, 16U, 0U,
                   MS_THREAD_OPT_USER, &test_tid);
    // do some thing

    return 0;
}
```

ms_thread_self()

- **描述** 获得当前线程的 ID
- **函数原型**

```
ms_handle_t ms_thread_self(void);
```

- **参数** 无

- **返回值** 当前线程的 ID
- **注意事项** 不能在中断中调用，不能在内核启动前调用
- **示例**

```
static ms_handle_t test_tid;

static void test_thread(void *arg)
{
    ms_handle_t tid = ms_thread_self();
    // do some thing
}

int main(int argc, char *argv[])
{
    ms_thread_create("test", test_thread, MS_NULL,
                     4096U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test_tid);
    ms_thread_sleep_s(1);
    return 0;
}
```

ms_thread_kill()

- **描述** 杀死一个线程
- **函数原型**

```
ms_err_t ms_thread_kill(ms_handle_t tid);
```

- **参数**

输入/输出	参数	描述
[in]	tid	线程的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核启动前调用，不能在内核锁定期间调用
- **示例**

```
static ms_handle_t test0_tid, test1_tid;

static void test0_thread(void *arg)
{
    ms_thread_sleep_s(1);
    ms_thread_kill(test1_tid);
}
```



```
static void test1_thread(void *arg)
{
    ms_thread_sleep_s(2);
}

int main(int argc, char *argv[])
{
    ms_thread_create("test0", test0_thread, MS_NULL,
                     4096U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test0_tid);
    ms_thread_create("test1", test1_thread, MS_NULL,
                     4096U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test1_tid);
    ms_thread_sleep_s(3);
    return 0;
}
```

ms_thread_exit()

- **描述** 当前线程主动退出（结束运行）
- **函数原型**

```
ms_err_t ms_thread_exit(void);
```

- **参数** 无
- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核启动前调用，不能在内核锁定期间调用
- **示例**

```
static ms_handle_t test_tid;

static void test_thread(void *arg)
{
    ms_thread_exit();
}

int main(int argc, char *argv[])
{
    ms_thread_create("test", test_thread, MS_NULL,
                     4096U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test_tid);
    ms_thread_sleep_s(1);
    return 0;
}
```

ms_thread_suspend()

- **描述** 暂停一个线程的运行

- **函数原型**

```
ms_err_t ms_thread_suspend(ms_handle_t tid);
```

- **参数**

输入/输出	参数	描述
[in]	tid	线程的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用
- **示例**

```
static ms_handle_t test0_tid, test1_tid;

static void test0_thread(void *arg)
{
    ms_thread_sleep_s(1);
    ms_thread_suspend(test1_tid);
}

static void test1_thread(void *arg)
{
    while (1) {
        // do some thing
    }
}

int main(int argc, char *argv[])
{
    ms_thread_create("test0", test0_thread, MS_NULL,
                     4096U, 15U, 0U,
                     MS_THREAD_OPT_USER, &test0_tid);
    ms_thread_create("test1", test1_thread, MS_NULL,
                     4096U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test1_tid);
    ms_thread_sleep_s(3);
    return 0;
}
```

ms_thread_resume()

- **描述** 恢复一个线程的运行
- **函数原型**

```
ms_err_t ms_thread_resume(ms_handle_t tid);
```


- **参数**

输入/输出	参数	描述
[in]	tid	线程的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
static ms_handle_t test0_tid, test1_tid;

static void test0_thread(void *arg)
{
    ms_thread_sleep_s(1);
    ms_thread_resume(test1_tid);
}

static void test1_thread(void *arg)
{
    ms_thread_sleep_s(1);
}

int main(int argc, char *argv[])
{
    ms_thread_create("test0", test0_thread, MS_NULL,
                     4096U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test0_tid);
    ms_thread_init("test1", test1_thread, MS_NULL,
                   4096U, 16U, 0U,
                   MS_THREAD_OPT_USER, &test1_tid);
    ms_thread_sleep_s(3);
    return 0;
}
```

ms_thread_yield()

- **描述** 当前线程让出 CPU 使用权
- **函数原型**

```
ms_err_t ms_thread_yield(void);
```

- **参数** 无
- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核启动前调用，不能在内核锁定期间调用
- **示例**

```c
static ms_handle_t test_tid;

static void test_thread(void *arg)
{
    ms_thread_yield();
}

int main(int argc, char *argv[])
{
    ms_thread_create("test", test_thread, MS_NULL,
                     4096U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test_tid);
    ms_thread_sleep_s(1);
    return 0;
}
```

ms_thread_sleep()

- **描述** 当前线程休眠指定的嘀嗒数
- **函数原型**

```c
ms_err_t ms_thread_sleep(ms_tick_t tick);
```

- **参数**

输入/输出	参数	描述
[in]	tick	休眠的嘀嗒数

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核启动前调用，不能在内核锁定期间调用
- **示例**

```c
static ms_handle_t test_tid;

static void test_thread(void *arg)
{
    ms_thread_sleep(10U);
}

int main(int argc, char *argv[])
{
    ms_thread_create("test", test_thread, MS_NULL,
                     4096U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test_tid);
    while (1) {
        ms_thread_suspend(ms_thread_self());
```

```
    }
    return 0;
}
```

ms_thread_sleep_ms()

- **描述** 当前线程休眠指定的毫秒数
- **函数原型**

```
ms_err_t ms_thread_sleep_ms(ms_uint32_t ms);
```

- **参数**

输入/输出	参数	描述
[in]	ms	休眠的毫秒数

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核启动前调用，不能在内核锁定期间调用
- **示例**

```
static ms_handle_t test_tid;

static void test_thread(void *arg)
{
    ms_thread_sleep_ms(10U);
}

int main(int argc, char *argv[])
{
    ms_thread_create("test", test_thread, MS_NULL,
                     4096U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test_tid);
    while (1) {
        ms_thread_suspend(ms_thread_self());
    }
    return 0;
}
```

ms_thread_sleep_s()

- **描述** 当前线程休眠指定的秒数
- **函数原型**

```
ms_err_t ms_thread_sleep_s(ms_uint32_t sec);
```


- **参数**

输入/输出	参数	描述
[in]	s	休眠的秒数

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核启动前调用，不能在内核锁定期间调用
- **示例**

```
static ms_handle_t test_tid;

static void test_thread(void *arg)
{
    ms_thread_sleep_s(1U);
}

int main(int argc, char *argv[])
{
    ms_thread_create("test", test_thread, MS_NULL,
                     4096U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test_tid);
    while (1) {
        ms_thread_suspend(ms_thread_self());
    }
    return 0;
}
```

ms_thread_sleep_hms()

- **描述** 当前线程休眠指定的时分秒数
- **函数原型**

```
ms_err_t ms_thread_sleep_hms(ms_uint32_t hour, ms_uint32_t min, ms_uint32_t sec);
```

- **参数**

输入/输出	参数	描述
[in]	hour	休眠的小时数
[in]	min	休眠的分钟数
[in]	sec	休眠的秒数

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核启动前调用，不能在内核锁定期间调用

- **示例**

```
static ms_handle_t test_tid;

static void test_thread(void *arg)
{
    ms_thread_sleep_hms(1U, 0U, 0U);
}

int main(int argc, char *argv[])
{
    ms_thread_create("test", test_thread, MS_NULL,
                     4096U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test_tid);
    while (1) {
        ms_thread_suspend(ms_thread_self());
    }
    return 0;
}
```

ms_thread_stat()

- **描述** 获得指定线程的状态信息
- **函数原型**

```
ms_err_t ms_thread_stat(ms_handle_t tid, ms_thread_stat_t *stat);
```

- **参数**

输入/输出	参数	描述
[in]	tid	线程的 ID
[out]	stat	线程的状态信息

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
static ms_handle_t test0_tid, test1_tid;

static void test0_thread(void *arg)
{
    ms_thread_stat_t stat;

    ms_thread_stat(test1_tid, &stat);

    // do some thing
```



```
}

static void test1_thread(void *arg)
{
    while (1) {
        // do some thing
    }
}

int main(int argc, char *argv[])
{
    ms_thread_create("test0", test0_thread, MS_NULL,
                     4096U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test0_tid);
    ms_thread_init("test1", test1_thread, MS_NULL,
                   4096U, 16U, 0U,
                   MS_THREAD_OPT_USER, &test1_tid);
    while (1) {
        ms_thread_suspend(ms_thread_self());
    }
    return 0;
}
```

ms_thread_errno_ptr()

- **描述** 获得当前线程 errno 的指针
- **函数原型**

```
int *ms_thread_errno_ptr(void);
```

- **参数** 无
- **返回值** 当前线程 errno 的指针
- **注意事项** 无
- **示例** 无

ms_thread_get_errno()

- **描述** 获得当前线程的 errno
- **函数原型**

```
int  ms_thread_get_errno(void);
```

- **参数** 无
- **返回值** 当前线程的 errno
- **注意事项** 无
- **示例** 无

ms_thread_set_errno()

- **描述** 设置当前线程的 errno
- **函数原型**

```
void ms_thread_set_errno(int _errno);
```

- **参数**

输入/输出	参数	描述
[in]	_errno	错误码

- **返回值** 无
- **注意事项** 无
- **示例** 无

7 MS-RTOS 二值信号量

> 本章将介绍 MS-RTOS 二值信号量的使用。

二值信号量相关数据类型

类型	描述
ms_semb_stat_t	二值信号量的状态信息结构类型

ms_semb_stat_t

二值信号量的状态信息结构类型：

```
typedef struct {
    ms_bool_t value;
} ms_semb_stat_t;
```

参数	说明
value	二值信号量的当前值

二值信号量相关的 API

下表展示了二值信号量相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_semb_create	●	●
ms_semb_destroy	●	●
ms_semb_wait	●	●
ms_semb_trywait	●	●
ms_semb_post	●	●
ms_semb_stat	●	●

ms_semb_create()

- **描述** 创建一个二值信号量
- **函数原型**

```
ms_err_t ms_semb_create(const char *name, ms_bool_t init_value, ms_ipc_opt_t opt,
```



```
                        ms_handle_t *sembid);
```

- **参数**

输入/输出	参数	描述
[in]	name	二值信号量的名字，不能为空指针
[in]	init_value	二值信号量的初始值
[in]	opt	二值信号量的选项
[out]	sembid	二值信号量的 ID，不能为空指针

opt 参数可以取如下的值：

宏	含义
MS_WAIT_TYPE_PRIO	按优先级高低规则等待
MS_WAIT_TYPE_FIFO	按先来先服务规则等待

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用
- **示例**

```
static ms_handle_t semb_id;

int main(int argc, char *argv[])
{
    ms_semb_create("semb", MS_TRUE, MS_WAIT_TYPE_PRIO, &semb_id);

    return 0;
}
```

ms_semb_destroy()

- **描述** 销毁一个二值信号量
- **函数原型**

```
ms_err_t ms_semb_destroy(ms_handle_t sembid);
```

- **参数**

输入/输出	参数	描述
[in]	sembid	二值信号量的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用
- **示例**

```
static ms_handle_t semb_id;

int main(int argc, char *argv[])
{
    ms_semb_create("semb", MS_TRUE, MS_WAIT_TYPE_PRIO, &semb_id);

    // do some thing

    ms_semb_destroy(semb_id);

    return 0;
}
```

ms_semb_wait()

- **描述** 等待一个二值信号量
- **函数原型**

```
ms_err_t ms_semb_wait(ms_handle_t sembid, ms_tick_t timeout);
```

- **参数**

输入/输出	参数	描述
[in]	sembid	二值信号量的 ID
[in]	timeout	等待的超时时间

- **返回值** MS-RTOS 内核错误码
- **注意事项** 中断中或内核锁定期间调用，timeout 必须为 `MS_TIMEOUT_NO_WAIT`，不能在内核启动前调用
- **示例**

```
static ms_handle_t semb_id;

int main(int argc, char *argv[])
{
    ms_semb_create("semb", MS_TRUE, MS_WAIT_TYPE_PRIO, &semb_id);

    ms_semb_wait(semb_id, 1000U);

    // do some thing

    ms_semb_destroy(semb_id);
```



```
    return 0;
}
```

ms_semb_trywait()

- **描述** 尝试等待一个二值信号量
- **函数原型**

```
ms_err_t ms_semb_trywait(ms_handle_t sembid);
```

- **参数**

输入/输出	参数	描述
[in]	sembid	二值信号量的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_semb_post()

- **描述** 发送一个二值信号量
- **函数原型**

```
ms_err_t ms_semb_post(ms_handle_t sembid);
```

- **参数**

输入/输出	参数	描述
[in]	sembid	二值信号量的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
static ms_handle_t test_tid;
static ms_handle_t semb_id;

static void test_thread(void *arg)
{
    ms_semb_wait(semb_id, MS_TIMEOUT_FOREVER);
```

```
    // semb come
    // do some thing
}

int main(int argc, char *argv[])
{
    ms_semb_create("semb", MS_FALSE, MS_WAIT_TYPE_PRIO, &semb_id);

    ms_thread_create("test", test_thread, MS_NULL,
                     4096U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test_tid);

    ms_thread_sleep_s(1U);

    // post semb
    ms_semb_post(semb_id);

    ms_thread_sleep_s(1U);

    return 0;
}
```

ms_semb_stat()

- **描述** 获得二值信号量的状态信息
- **函数原型**

```
ms_err_t ms_semb_stat(ms_handle_t sembid, ms_semb_stat_t *stat);
```

- **参数**

输入/输出	参数	描述
[in]	sembid	二值信号量的 ID
[out]	stat	二值信号量的状态信息

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
static ms_handle_t semb_id;

int main(int argc, char *argv[])
{
    ms_semb_stat_t stat;

    ms_semb_create("semb", MS_FALSE, MS_WAIT_TYPE_PRIO, &semb_id);
```



```
    ms_semb_stat(semb_id, &stat);
    // do some thing

    return 0;
}
```


8 MS-RTOS 计数信号量

> 本章将介绍 MS-RTOS 计数信号量的使用。

计数信号量相关数据类型

类型	描述
ms_semc_stat_t	计数信号量的状态信息结构类型

ms_semc_stat_t

计数信号量的状态信息结构类型：

```
typedef struct {
    ms_uint32_t value;
} ms_semc_stat_t;
```

参数	说明
value	计数信号量的当前值

计数信号量相关 API

下表展示了计数信号量相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_semc_create	●	●
ms_semc_destroy	●	●
ms_semc_wait	●	●
ms_semc_trywait	●	●
ms_semc_post	●	●
ms_semc_stat	●	●

ms_semc_create()

- **描述** 创建一个计数信号量
- **函数原型**

```
ms_err_t ms_semc_create(const char *name, ms_uint32_t init_value, ms_uint32_t max_value,
```



```
                        ms_ipc_opt_t opt, ms_handle_t *semcid);
```

- **参数**

输入/输出	参数	描述
[in]	name	计数信号量的名字，不能为空指针
[in]	init_value	计数信号量的初始值，必须 <= max_value
[in]	max_value	计数信号量的最大值
[in]	opt	计数信号量的选项
[out]	semcid	计数信号量的 ID，不能为空指针

opt 参数可以取如下的值：

宏	含义
MS_WAIT_TYPE_PRIO	按优先级高低规则等待
MS_WAIT_TYPE_FIFO	按先来先服务规则等待

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用
- **示例**

```
static ms_handle_t semc_id;

int main(int argc, char *argv[])
{
    ms_semc_create("semc", 0U, 100U, MS_WAIT_TYPE_PRIO, &semc_id);

    return 0;
}
```

ms_semc_destroy()

- **描述** 销毁一个计数信号量
- **函数原型**

```
ms_err_t ms_semc_destroy(ms_handle_t semcid);
```

- **参数**

输入/输出	参数	描述
[in]	semcid	计数信号量的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用
- **示例**

```
static ms_handle_t semc_id;

int main(int argc, char *argv[])
{
    ms_semc_create("semc", 0U, 100U, MS_WAIT_TYPE_PRIO, &semc_id);

    // do some thing

    ms_semc_destroy(semc_id);

    return 0;
}
```

ms_semc_wait()

- **描述** 等待一个计数信号量
- **函数原型**

```
ms_err_t ms_semc_wait(ms_handle_t semcid, ms_tick_t timeout);
```

- **参数**

输入/输出	参数	描述
[in]	semcid	计数信号量的 ID
[in]	timeout	等待的超时时间

- **返回值** MS-RTOS 内核错误码
- **注意事项** 中断中或内核锁定期间调用，timeout 必须为 `MS_TIMEOUT_NO_WAIT`，不能在内核启动前调用
- **示例**

```
static ms_handle_t semc_id;

int main(int argc, char *argv[])
{
    ms_semc_create("semc", 1U, 100U, MS_WAIT_TYPE_PRIO, &semc_id);
```



```
    ms_semc_wait(semc_id, 1000U);

    // do some thing

    ms_semc_destroy(semc_id);

    return 0;
}
```

ms_semc_trywait()

- **描述** 尝试等待一个计数信号量
- **函数原型**

```
ms_err_t ms_semc_trywait(ms_handle_t semcid);
```

- **参数**

输入/输出	参数	描述
[in]	semcid	计数信号量的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_semc_post()

- **描述** 发送一个计数信号量
- **函数原型**

```
ms_err_t ms_semc_post(ms_handle_t semcid);
```

- **参数**

输入/输出	参数	描述
[in]	semcid	计数信号量的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
static ms_handle_t test_tid;
static ms_handle_t semc_id;
```



```
static void test_thread(void *arg)
{
    ms_semc_wait(semc_id, MS_TIMEOUT_FOREVER);

    // semc come
    // do some thing
}

int main(int argc, char *argv[])
{
    ms_semc_create("semc", 0U, 100U, MS_WAIT_TYPE_PRIO, &semc_id);

    ms_thread_create("test", test_thread, MS_NULL,
                     4096U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test_tid);

    ms_thread_sleep_s(1U);

    // post semc
    ms_semc_post(semc_id);

    ms_thread_sleep_s(1U);

    return 0;
}
```

ms_semc_stat()

- **描述** 获得计数信号量的状态信息
- **函数原型**

```
ms_err_t ms_semc_stat(ms_handle_t semcid, ms_semc_stat_t *stat);
```

- **参数**

输入/输出	参数	描述
[in]	semcid	计数信号量的 ID
[out]	stat	计数信号量的状态信息

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
static ms_handle_t semc_id;

int main(int argc, char *argv[])
```



```
{
    ms_semc_stat_t stat;

    ms_semc_create("semc", 0U, 100U, MS_WAIT_TYPE_PRIO, &semc_id);

    ms_semc_stat(semc_id, &stat);
    // do some thing

    return 0;
}
```


9 MS-RTOS 互斥量

> 本章将介绍 MS-RTOS 互斥量的使用。

互斥量相关 API

下表展示了互斥量相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_mutex_create	●	●
ms_mutex_destroy	●	●
ms_mutex_lock	●	●
ms_mutex_trylock	●	●
ms_mutex_unlock	●	●

ms_mutex_create()

- **描述** 创建一个互斥量
- **函数原型**

```
ms_err_t ms_mutex_create(const char *name, ms_ipc_opt_t opt, ms_handle_t *mutexid);
```

- **参数**

输入/输出	参数	描述
[in]	name	互斥量的名字，不能为空指针
[in]	opt	互斥量的选项
[out]	mutexid	互斥量的 ID，不能为空指针

opt 参数可以取如下的值：

宏	含义
MS_WAIT_TYPE_PRIO	按优先级高低规则等待
MS_WAIT_TYPE_FIFO	按先来先服务规则等待

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用

- **示例**

```
static ms_handle_t mutex_id;

int main(int argc, char *argv[])
{
    ms_mutex_create("mutex", MS_WAIT_TYPE_PRIO, &mutex_id);

    return 0;
}
```

ms_mutex_destroy()

- **描述** 销毁一个互斥量
- **函数原型**

```
ms_err_t ms_mutex_destroy(ms_handle_t mutexid);
```

- **参数**

输入/输出	参数	描述
[in]	mutexid	互斥量的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用
- **示例**

```
static ms_handle_t mutex_id;

int main(int argc, char *argv[])
{
    ms_mutex_create("mutex", MS_WAIT_TYPE_PRIO, &mutex_id);

    // do some thing

    ms_mutex_destroy(mutexid);

    return 0;
}
```

ms_mutex_lock()

- **描述** 锁住一个互斥量
- **函数原型**


```
ms_err_t ms_mutex_lock(ms_handle_t mutexid, ms_tick_t timeout);
```

- **参数**

输入/输出	参数	描述
[in]	mutexid	互斥量的 ID
[in]	timeout	等待的超时时间

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，在内核锁定期间调用，timeout 必须为 `MS_TIMEOUT_NO_WAIT`，不能在内核启动前调用，与 `ms_mutex_unlock` 配对使用
- **示例** 见 `ms_mutex_unlock`

ms_mutex_trylock()

- **描述** 尝试锁住一个互斥量
- **函数原型**

```
ms_err_t ms_mutex_trylock(ms_handle_t mutexid);
```

- **参数**

输入/输出	参数	描述
[in]	mutexid	互斥量的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核启动前调用，与 `ms_mutex_unlock` 配对使用
- **示例** 见 `ms_mutex_unlock`

ms_mutex_unlock()

- **描述** 解锁一个互斥量
- **函数原型**

```
ms_err_t ms_mutex_unlock(ms_handle_t mutexid);
```

- **参数**

输入/输出	参数	描述
[in]	mutexid	互斥量的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核启动前调用，与 `ms_mutex_lock` 配对使用
- **示例**

```c
static ms_handle_t mutex_id;

int main(int argc, char *argv[])
{
    ms_mutex_create("mutex", MS_WAIT_TYPE_PRIO, &mutex_id);

    ms_mutex_lock(mutex_id, MS_TIMEOUT_FOREVER);

    // do some thing

    ms_mutex_unlock(mutex_id);

    ms_mutex_destroy(mutexid);

    return 0;
}
```


10 MS-RTOS 条件变量

> 本章将介绍 MS-RTOS 条件变量的使用。
>
> 条件变量只能与互斥量配合使用，如果您还不清楚互斥量，建议先查阅互斥量章节。

条件变量相关 API

下表展示了条件变量相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_cond_create	●	●
ms_cond_destroy	●	●
ms_cond_wait	●	●
ms_cond_trywait	●	●
ms_cond_signal	●	●
ms_cond_broadcast	●	●

ms_cond_create()

- **描述** 创建一个条件变量
- **函数原型**

```
ms_err_t ms_cond_create(const char *name, ms_ipc_opt_t opt, ms_handle_t *condid);
```

- **参数**

输入/输出	参数	描述
[in]	name	条件变量的名字，不能为空指针
[in]	opt	条件变量的选项
[out]	condid	条件变量的 ID，不能为空指针

opt 参数可以取如下的值：

宏	含义
MS_WAIT_TYPE_PRIO	按优先级高低规则等待
MS_WAIT_TYPE_FIFO	按先来先服务规则等待

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用
- **示例**

```
static ms_handle_t cond_id;

int main(int argc, char *argv[])
{
    ms_cond_create("cond", MS_WAIT_TYPE_PRIO, &cond_id);

    return 0;
}
```

ms_cond_destroy()

- **描述** 销毁一个条件变量
- **函数原型**

```
ms_err_t ms_cond_destroy(ms_handle_t condid);
```

- **参数**

输入/输出	参数	描述
[in]	condid	条件变量的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用
- **示例**

```
static ms_handle_t cond_id;

int main(int argc, char *argv[])
{
    ms_cond_create("cond", MS_WAIT_TYPE_PRIO, &cond_id);

    ms_thread_sleep_s(1U);

    ms_cond_destroy(cond_id);

    return 0;
}
```

ms_cond_wait()

- **描述** 等待一个条件变量

- **函数原型**

```
ms_err_t ms_cond_wait(ms_handle_t condid, ms_handle_t mutexid, ms_tick_t timeout);
```

- **参数**

输入/输出	参数	描述
[in]	condid	条件变量的 ID
[in]	mutexid	互斥量的 ID
[in]	timeout	等待的超时时间

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用
- **示例**

```
static ms_handle_t cond_id;
static ms_handle_t mutex_id;

int main(int argc, char *argv[])
{
    ms_cond_create("cond", MS_WAIT_TYPE_PRIO, &cond_id);
    ms_mutex_create("mutex", MS_WAIT_TYPE_PRIO, &mutex_id);

    ms_cond_wait(cond_id, mutex_id, 1000U);

    return 0;
}
```

ms_cond_trywait()

- **描述** 尝试等待一个条件变量
- **函数原型**

```
ms_err_t ms_cond_trywait(ms_handle_t condid, ms_handle_t mutexid);
```

- **参数**

输入/输出	参数	描述
[in]	condid	条件变量的 ID
[in]	mutexid	互斥量的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用
- **示例** 无

ms_cond_signal()

- **描述** 发送一个条件变量
- **函数原型**

```
ms_err_t ms_cond_signal(ms_handle_t condid);
```

- **参数**

输入/输出	参数	描述
[in]	condid	条件变量的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
static ms_handle_t test_tid;
static ms_handle_t cond_id;
static ms_handle_t mutex_id;

static void test_thread(void *arg)
{
    // lock the resource pool
    ms_mutex_lock(mutex_id, MS_TIMEOUT_FOREVER);

    // do some thing
    // no resource, wait condition variable
    ms_cond_wait(cond_id, mutex_id, MS_TIMEOUT_FOREVER);
    // get resource, do some thing

    // unlock the resource pool
    ms_mutex_unlock(mutex_id);
}

int main(int argc, char *argv[])
{
    ms_cond_create("cond", MS_WAIT_TYPE_PRIO, &cond_id);
    ms_mutex_create("mutex", MS_WAIT_TYPE_PRIO, &mutex_id);

    ms_thread_create("test", test_thread, MS_NULL,
                     1024U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test_tid);
```



```
    ms_thread_sleep_s(1U);

    // lock the resource pool
    ms_mutex_lock(mutex_id, MS_TIMEOUT_FOREVER);

    // produce resource, signal condition variable
    ms_cond_signal(cond_id);

    // unlock the resource pool
    ms_mutex_unlock(mutex_id);

    ms_thread_sleep_s(1U);

    return 0;
}
```

ms_cond_broadcast()

- **描述** 广播一个条件变量
- **函数原型**

```
ms_err_t ms_cond_broadcast(ms_handle_t condid);
```

- **参数**

输入/输出	参数	描述
[in]	condid	条件变量的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
#include <ms_rtos.h>

static ms_handle_t test0_tid;
static ms_handle_t test1_tid;
static ms_handle_t cond_id;
static ms_handle_t mutex_id;

static void test_thread0(void *arg)
{
    // lock the resource pool
    ms_mutex_lock(mutex_id, MS_TIMEOUT_FOREVER);

    // do some thing
    // no resource, wait condition variable
    ms_cond_wait(cond_id, mutex_id, MS_TIMEOUT_FOREVER);
```



```
    // get resource, do some thing

    // unlock the resource pool
    ms_mutex_unlock(mutex_id);
}

static void test_thread1(void *arg)
{
    // lock the resource pool
    ms_mutex_lock(mutex_id, MS_TIMEOUT_FOREVER);

    // do some thing
    // no resource, wait condition variable
    ms_cond_wait(cond_id, mutex_id, MS_TIMEOUT_FOREVER);
    // get resource, do some thing

    // unlock the resource pool
    ms_mutex_unlock(mutex_id);
}

int main(int argc, char *argv[])
{
    ms_cond_create("cond", MS_WAIT_TYPE_PRIO, &cond_id);
    ms_mutex_create("mutex", MS_WAIT_TYPE_PRIO, &mutex_id);

    ms_thread_create("test0", test_thread0, MS_NULL,
                     1024U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test0_tid);

    ms_thread_create("test1", test_thread1, MS_NULL,
                     1024U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test1_tid);

    ms_thread_sleep_s(1U);

    // lock the resource pool
    ms_mutex_lock(mutex_id, MS_TIMEOUT_FOREVER);

    // produce resource, broadcast condition variable
    ms_cond_broadcast(cond_id);

    // unlock the resource pool
    ms_mutex_unlock(mutex_id);

    ms_thread_sleep_s(1U);

    return 0;
}
```


11 MS-RTOS 读写锁

> 本章将介绍 MS-RTOS 读写锁的使用。

读写锁相关 API

下表展示了读写锁相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_rwlock_create	●	●
ms_rwlock_destroy	●	●
ms_rwlock_lock_read	●	●
ms_rwlock_trylock_read	●	●
ms_rwlock_lock_write	●	●
ms_rwlock_trylock_write	●	●
ms_rwlock_unlock	●	●

ms_rwlock_create()

- **描述** 创建一个读写锁
- **函数原型**

```
ms_err_t ms_rwlock_create(const char *name, ms_ipc_opt_t opt,
                          ms_handle_t *rwlockid);
```

- **参数**

输入/输出	参数	描述
[in]	name	读写锁的名字，不能为空指针
[in]	opt	读写锁的选项
[out]	rwlockid	读写锁的 ID，不能为空指针

opt 参数可以取如下的值：

宏	含义
MS_WAIT_TYPE_PRIO	按优先级高低规则等待

宏	含义
MS_WAIT_TYPE_FIFO	按先来先服务规则等待

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用
- **示例**

```
static ms_handle_t rwlock_id;

int main(int argc, char *argv[])
{
    ms_rwlock_create("rwlock", MS_WAIT_TYPE_PRIO, &rwlock_id);

    return 0;
}
```

ms_rwlock_destroy()

- **描述** 销毁一个读写锁
- **函数原型**

```
ms_err_t ms_rwlock_destroy(ms_handle_t rwlockid);
```

- **参数**

输入/输出	参数	描述
[in]	rwlockid	读写锁的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用
- **示例**

```
static ms_handle_t rwlock_id;

int main(int argc, char *argv[])
{
    ms_rwlock_create("rwlock", MS_WAIT_TYPE_PRIO, &rwlock_id);

    // do some thing

    ms_rwlock_destroy(rwlockid);
```



```
    return 0;
}
```

ms_rwlock_lock_read()

- **描述** 以读的方式锁住一个读写锁
- **函数原型**

```
ms_err_t ms_rwlock_lock_read(ms_handle_t rwlockid, ms_tick_t timeout);
```

- **参数**

输入/输出	参数	描述
[in]	rwlockid	读写锁的 ID
[in]	timeout	等待的超时时间

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，在内核锁定期间调用，timeout 必须为 `MS_TIMEOUT_NO_WAIT`，不能在内核启动前调用，与 `ms_rwlock_unlock` 配对使用
- **示例** 见 `ms_rwlock_unlock()`

ms_rwlock_trylock_read()

- **描述** 尝试以读的方式锁住一个读写锁
- **函数原型**

```
ms_err_t ms_rwlock_trylock_read(ms_handle_t rwlockid);
```

- **参数**

输入/输出	参数	描述
[in]	rwlockid	读写锁的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_rwlock_lock_write()

- **描述** 以写的方式锁住一个读写锁
- **函数原型**


```
ms_err_t ms_rwlock_lock_write(ms_handle_t rwlockid, ms_tick_t timeout);
```

- **参数**

输入/输出	参数	描述
[in]	rwlockid	读写锁的 ID
[in]	timeout	等待的超时时间

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，在内核锁定期间调用，timeout 必须为 `MS_TIMEOUT_NO_WAIT`，不能在内核启动前调用，与 `ms_rwlock_unlock` 配对使用
- **示例** 见 `ms_rwlock_unlock()`

ms_rwlock_trylock_write()

- **描述** 尝试以写的方式锁住一个读写锁
- **函数原型**

```
ms_err_t ms_rwlock_trylock_write(ms_handle_t rwlockid);
```

- **参数**

输入/输出	参数	描述
[in]	rwlockid	读写锁的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_rwlock_unlock()

- **描述** 解锁一个读写锁
- **函数原型**

```
ms_err_t ms_rwlock_unlock(ms_handle_t rwlockid);
```

- **参数**

输入/输出	参数	描述
[in]	rwlockid	读写锁的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核启动前调用
- **示例**

```
static ms_handle_t rwlock_id;

int main(int argc, char *argv[])
{
    ms_rwlock_create("rwlock", MS_WAIT_TYPE_PRIO, &rwlock_id);

    ms_rwlock_lock_write(rwlock_id, MS_TIMEOUT_FOREVER);
    // do some thing

    ms_rwlock_unlock(rwlock_id);

    ms_rwlock_destroy(rwlockid);

    return 0;
}
```


12 MS-RTOS 事件标志组

> 本章将介绍 MS-RTOS 事件标志组的使用。

事件标志组相关数据类型

类型	描述
ms_eventset_post_opt_t	事件标志组的发送选项类型
ms_eventset_wait_opt_t	事件标志组的等待选项类型
ms_eventset_stat_t	事件标志组的状态信息结构类型

ms_eventset_post_opt_t

事件标志组的发送选项为以下的宏：

宏	含义
MS_EVENTSET_OPT_SET	设置事件
MS_EVENTSET_OPT_CLEAR	清除事件

MS_EVENTSET_OPT_SET 与 MS_EVENTSET_OPT_CLEAR 互斥使用；

ms_eventset_wait_opt_t

事件标志组的等待选项为以下的宏的组合：

宏	含义
MS_EVENTSET_OPT_ALL	等待所指定的事件标志集合全部得到满足
MS_EVENTSET_OPT_ANY	等待所指定的事件标志集合任意一个得到满足
MS_EVENTSET_OPT_CONSUME	消费掉获取的事件

MS_EVENTSET_OPT_ALL 与 MS_EVENTSET_OPT_ANY 互斥使用；

ms_eventset_stat_t

事件标志组的状态信息结构类型：

```
typedef struct {
    ms_uint32_t value;
} ms_eventset_stat_t;
```


参数	说明
value	事件标志组的当前值

事件标志组相关 API

下表展示了事件标志组相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_eventset_create	●	●
ms_eventset_destroy	●	●
ms_eventset_wait	●	●
ms_eventset_trywait	●	●
ms_eventset_post	●	●
ms_eventset_stat	●	●
ms_eventset_set	●	●
ms_eventset_clear	●	●
ms_eventset_reset	●	●

ms_eventset_create()

- **描述** 创建一个事件标志组
- **函数原型**

```
ms_err_t ms_eventset_create(const char *name, ms_uint32_t init_value,
                            ms_ipc_opt_t opt, ms_handle_t *eventsetid);
```

- **参数**

输入/输出	参数	描述
[in]	name	事件标志组的名字，不能为空指针
[in]	init_value	事件标志组的初始值
[in]	opt	事件标志组的选项
[out]	eventsetid	事件标志组的 ID，不能为空指针

opt 参数可以取如下的值：

宏	含义
MS_WAIT_TYPE_PRIO	按优先级高低规则等待
MS_WAIT_TYPE_FIFO	按先来先服务规则等待

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用
- **示例**

```
static ms_handle_t eventset_id;

int main(int argc, char *argv[])
{
    ms_eventset_create("eventset", 0U, MS_WAIT_TYPE_PRIO, &eventset_id);

    return 0;
}
```

ms_eventset_destroy()

- **描述** 销毁一个事件标志组
- **函数原型**

```
ms_err_t ms_eventset_destroy(ms_handle_t eventsetid);
```

- **参数**

输入/输出	参数	描述
[in]	eventsetid	事件标志组的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用
- **示例**

```
static ms_handle_t eventset_id;

int main(int argc, char *argv[])
{
    ms_eventset_create("eventset", 0U, MS_WAIT_TYPE_PRIO, &eventset_id);

    ms_thread_sleep_s(1U);

    ms_eventset_destroy(eventset_id);
```

```
    return 0;
}
```

ms_eventset_wait()

- **描述** 等待一个事件标志组
- **函数原型**

```
ms_err_t ms_eventset_wait(ms_handle_t eventsetid, ms_uint32_t eventset,
                          ms_eventset_wait_opt_t opt, ms_tick_t timeout,
                          ms_uint32_t *result);
```

- **参数**

输入/输出	参数	描述
[in]	eventsetid	事件标志组的 ID
[in]	eventset	等待的事件标志集合
[in]	opt	事件标志组的等待选项
[in]	timeout	等待的超时时间
[out]	result	事件标志组结果，如果不关心，可以为空指针

- **返回值** MS-RTOS 内核错误码
- **注意事项** 中断中或内核锁定期间调用，timeout 必须为 `MS_TIMEOUT_NO_WAIT`，不能在内核启动前调用
- **示例**

```
static ms_handle_t eventset_id;

int main(int argc, char *argv[])
{
    ms_eventset_create("eventset", 0U, MS_WAIT_TYPE_PRIO, &eventset_id);

    ms_eventset_wait(eventset_id, MS_BIT(0U), MS_EVENTSET_OPT_ALL, 1000U, MS_NULL);

    return 0;
}
```

ms_eventset_trywait()

- **描述** 尝试等待一个事件标志组
- **函数原型**


```
ms_err_t ms_eventset_trywait(ms_handle_t eventsetid, ms_uint32_t eventset,
                             ms_eventset_wait_opt_t opt, ms_uint32_t *result);
```

- **参数**

输入/输出	参数	描述
[in]	eventsetid	事件标志组的 ID
[in]	eventset	尝试等待的事件标志集合
[in]	opt	事件标志组的等待选项
[out]	result	事件标志组结果，如果不关心，可以为空指针

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_eventset_post()

- **描述** 发送一个事件标志组
- **函数原型**

```
ms_err_t ms_eventset_post(ms_handle_t eventsetid, ms_uint32_t eventset,
                          ms_eventset_post_opt_t opt);
```

- **参数**

输入/输出	参数	描述
[in]	eventsetid	事件标志组的 ID
[in]	eventset	发送的事件标志集合
[in]	opt	事件标志组的发送选项

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
static ms_handle_t test_tid;
static ms_handle_t eventset_id;

static void test_thread(void *arg)
{
    ms_eventset_wait(eventset_id, MS_BIT(0U), MS_EVENTSET_OPT_ALL, MS_TIMEOUT_FOREVER,
```

```
                    MS_NULL);

    // event 0 come
    // do some thing
}

int main(int argc, char *argv[])
{
    ms_eventset_create("eventset", 0U, MS_WAIT_TYPE_PRIO, &eventset_id);

    ms_thread_create("test", test_thread, MS_NULL,
                     1024U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test_tid);

    ms_thread_sleep_s(1U);

    // post event 0
    ms_eventset_post(eventset_id, MS_BIT(0U), MS_EVENTSET_OPT_SET);

    ms_thread_sleep_s(1U);

    return 0;
}
```

ms_eventset_stat()

- **描述** 获得事件标志组的状态信息
- **函数原型**

```
ms_err_t ms_eventset_stat(ms_handle_t eventsetid, ms_eventset_stat_t *stat);
```

- **参数**

输入/输出	参数	描述
[in]	eventsetid	事件标志组的 ID
[out]	stat	事件标志组的状态信息

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
static ms_handle_t eventset_id;

int main(int argc, char *argv[])
{
    ms_eventset_stat_t stat;
```

```
    ms_eventset_create("eventset", MS_BIT(0U), MS_WAIT_TYPE_PRIO, &eventset_id);

    ms_eventset_stat(eventset_id, &stat);
    // do some thing
    return 0;
}
```

ms_eventset_set()

- **描述** 设置事件标志组中的指定事件，等价于以下操作：

```
ms_eventset_post(eventsetid, eventset, MS_EVENTSET_OPT_SET)
```

- **函数原型**

```
ms_err_t ms_eventset_set(ms_handle_t eventsetid, ms_uint32_t eventset);
```

- **参数**

输入/输出	参数	描述
[in]	eventsetid	事件标志组的 ID
[in]	eventset	需要设置的事件标志集合

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_eventset_clear()

- **描述** 清除事件标志组中的指定事件，等价于以下操作：

```
ms_eventset_post(eventsetid, eventset, MS_EVENTSET_OPT_CLEAR)
```

- **函数原型**

```
ms_err_t ms_eventset_clear(ms_handle_t eventsetid, ms_uint32_t eventset);
```

- **参数**

输入/输出	参数	描述
[in]	eventsetid	事件标志组的 ID

输入/输出	参数	描述
[in]	eventset	需要清除的事件标志集合

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_eventset_reset()

- **描述** 复位事件标志组（清除事件标志组中的所有事件），等价于以下操作：

```
ms_eventset_post(eventsetid, ((ms_uint32_t)-1), MS_EVENTSET_OPT_CLEAR)
```

- **函数原型**

```
ms_err_t ms_eventset_reset(ms_handle_t eventsetid);
```

- **参数**

输入/输出	参数	描述
[in]	eventsetid	事件标志组的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

13 MS-RTOS 消息队列

> 本章将介绍 MS-RTOS 消息队列的使用。

消息队列相关数据类型

类型	描述
ms_mqueue_stat_t	消息队列的状态信息结构类型

ms_eventset_stat_t

消息队列的状态信息结构类型：

```
typedef struct {
    ms_ptr_t    msg_buf;
    ms_uint16_t msg_count;
} ms_mqueue_stat_t;
```

参数	说明
msg_buf	消息缓冲区的基地址
msg_count	消息队列中消息数目

消息队列相关 API

下表展示了消息队列相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_mqueue_create	●	●
ms_mqueue_destroy	●	●
ms_mqueue_wait	●	●
ms_mqueue_trywait	●	●
ms_mqueue_post	●	●
ms_mqueue_trypost	●	●
ms_mqueue_post_overwrite	●	●
ms_mqueue_post_front	●	●
ms_mqueue_trypost_front	●	●

API	用户空间	内核空间
ms_mqueue_post_front_overwrite	●	●
ms_mqueue_flush	●	●
ms_mqueue_stat	●	●
ms_mqueue_recv	●	●
ms_mqueue_send	●	●

ms_mqueue_create()

- **描述** 创建一个消息队列
- **函数原型**

```
ms_err_t ms_mqueue_create(const char *name, ms_ptr_t msg_buf, ms_uint16_t n_msg,
                          ms_uint16_t msg_size, ms_ipc_opt_t opt,
                          ms_handle_t *mqueueid);
```

- **参数**

输入/输出	参数	描述
[in]	name	消息队列的名字，不能为空指针
[in]	msg_buf	消息缓冲区的基地址
[in]	n_msg	消息缓冲区能存放多少则消息
[in]	msg_size	每则消息的大小
[in]	opt	消息队列的选项
[out]	mqueueid	消息队列的 ID，不能为空指针

opt 参数可以取如下的值：

宏	含义
MS_WAIT_TYPE_PRIO	按优先级高低规则等待
MS_WAIT_TYPE_FIFO	按先来先服务规则等待

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用
- **示例**


```c
static ms_handle_t mqueue_id;

static ms_uint8_t mqueue[16U * 4U];

int main(int argc, char *argv[])
{
    ms_mqueue_create("mqueue", mqueue, 16U, 4U, MS_WAIT_TYPE_PRIO, &mqueue_id);

    return 0;
}
```

ms_mqueue_destroy()

- **描述** 销毁一个消息队列
- **函数原型**

```c
ms_err_t ms_mqueue_destroy(ms_handle_t mqueueid);
```

- **参数**

输入/输出	参数	描述
[in]	mqueueid	消息队列的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用
- **示例**

```c
static ms_handle_t mqueue_id;

static ms_uint8_t mqueue[16U * 4U];

int main(int argc, char *argv[])
{
    ms_mqueue_create("mqueue", mqueue, 16U, 4U, MS_WAIT_TYPE_PRIO, &mqueue_id);

    // do some thing

    ms_mqueue_destroy(mqueue_id);

    return 0;
}
```

ms_mqueue_wait()

- **描述** 从消息队列中取出一则消息，也可以使用接口别名 `ms_mqueue_recv`

- **函数原型**

```
ms_err_t ms_mqueue_wait(ms_handle_t mqueueid, ms_ptr_t msg, ms_tick_t timeout);
```

- **参数**

输入/输出	参数	描述
[in]	mqueueid	消息队列的 ID
[out]	msg	用于接收消息的缓冲区
[in]	timeout	消息队列空时需要等待的超时时间

- **返回值** MS-RTOS 内核错误码
- **注意事项** 中断中或内核锁定期间调用，timeout 必须为 `MS_TIMEOUT_NO_WAIT`，不能在内核启动前调用
- **示例**

```
static ms_handle_t mqueue_id;

static ms_uint8_t mqueue[16U * 4U];

int main(int argc, char *argv[])
{
    ms_uint32_t msg;

    ms_mqueue_create("mqueue", mqueue, 16U, 4U, MS_WAIT_TYPE_PRIO, &mqueue_id);

    ms_mqueue_wait(mqueue_id, &msg, 1000U);

    ms_mqueue_destroy(mqueue_id);

    return 0;
}
```

ms_mqueue_trywait()

- **描述** 尝试从消息队列中取出一则消息
- **函数原型**

```
ms_err_t ms_mqueue_trywait(ms_handle_t mqueueid, ms_ptr_t msg);
```

- **参数**

输入/输出	参数	描述
[in]	mqueueid	消息队列的 ID

输入/输出	参数	描述
[out]	msg	用于接收消息的缓冲区

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_mqueue_post()

- **描述** 投递一则消息到消息队列，也可以使用接口别名 `ms_mqueue_send`
- **函数原型**

```
ms_err_t ms_mqueue_post(ms_handle_t mqueueid, ms_ptr_t msg, ms_tick_t timeout);
```

- **参数**

输入/输出	参数	描述
[in]	mqueueid	消息队列的 ID
[in]	msg	消息的缓冲区
[in]	timeout	消息队列满时需要等待的超时时间

- **返回值** MS-RTOS 内核错误码
- **注意事项** 中断中、内核锁定期间或内核启动前调用，timeout 必须为 `MS_TIMEOUT_NO_WAIT`
- **示例**

```
static ms_handle_t test_tid;
static ms_handle_t mqueue_id;

static ms_uint8_t mqueue[16U * 4U];

static void test_thread(void *arg)
{
    ms_uint32_t msg;

    ms_mqueue_wait(mqueue_id, &msg, MS_TIMEOUT_FOREVER);

    // msg come
    // do some thing
}

int main(int argc, char *argv[])
{
    ms_uint32_t msg;

    ms_mqueue_create("mqueue", mqueue, 16U, 4U, MS_WAIT_TYPE_PRIO, &mqueue_id);
```

```
    ms_thread_create("test", test_thread, MS_NULL,
                     4096U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test_tid);

    ms_thread_sleep_s(1U);

    // post msg
    msg = 0xabcdabcdUL;
    ms_mqueue_post(mqueue_id, &msg, MS_TIMEOUT_FOREVER);

    ms_thread_sleep_s(1U);

    return 0;
}
```

ms_mqueue_trypost()

- **描述** 尝试投递一则消息到消息队列
- **函数原型**

```
ms_err_t ms_mqueue_trypost(ms_handle_t mqueueid, ms_ptr_t msg);
```

- **参数**

输入/输出	参数	描述
[in]	mqueueid	消息队列的 ID
[in]	msg	消息的缓冲区

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_mqueue_post_overwrite()

- **描述** 投递一则消息到消息队列，如果队列满，则覆写队列末尾的一则消息
- **函数原型**

```
ms_err_t ms_mqueue_post_overwrite(ms_handle_t mqueueid, ms_const_ptr_t msg);
```

- **参数**

输入/输出	参数	描述
[in]	mqueueid	消息队列的 ID

输入/输出	参数	描述
[in]	msg	消息的缓冲区

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_mqueue_post_front()

- **描述** 投递一则消息到消息队列的前面
- **函数原型**


```
ms_err_t ms_mqueue_post_front(ms_handle_t mqueueid, ms_const_ptr_t msg, ms_tick_t timeout);
```

- **参数**

输入/输出	参数	描述
[in]	mqueueid	消息队列的 ID
[in]	msg	消息的缓冲区
[in]	timeout	消息队列满时需要等待的超时时间

- **返回值** MS-RTOS 内核错误码
- **注意事项** 中断中、内核锁定期间或内核启动前调用，timeout 必须为 `MS_TIMEOUT_NO_WAIT`
- **示例**


```
static ms_handle_t test_tid;
static ms_handle_t mqueue_id;

static ms_uint8_t mqueue[16U * 4U];

static void test_thread(void *arg)
{
    ms_uint32_t msg;

    ms_mqueue_wait(mqueue_id, &msg, MS_TIMEOUT_FOREVER);

    // msg come
    // do some thing
}

int main(int argc, char *argv[])
{
    ms_uint32_t msg;
```



```
    ms_mqueue_create("mqueue", mqueue, 16U, 4U, MS_WAIT_TYPE_PRIO, &mqueue_id);

    ms_thread_create("test", test_thread, MS_NULL,
                     4096U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test_tid);

    ms_thread_sleep_s(1U);

    // post msg
    msg = 0xabcdabcdUL;
    ms_mqueue_post_front(mqueue_id, &msg, MS_TIMEOUT_FOREVER);

    ms_thread_sleep_s(1U);

    return 0;
}
```

ms_mqueue_trypost_front()

- **描述** 尝试投递一则消息到消息队列的前面
- **函数原型**

```
ms_err_t ms_mqueue_trypost_front(ms_handle_t mqueueid, ms_ptr_t msg);
```

- **参数**

输入/输出	参数	描述
[in]	mqueueid	消息队列的 ID
[in]	msg	消息的缓冲区

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_mqueue_post_front_overwrite()

- **描述** 投递一则消息到消息队列的前面，如果队列满，则覆写最前面的一则消息
- **函数原型**

```
ms_err_t ms_mqueue_post_overwrite(ms_handle_t mqueueid, ms_const_ptr_t msg);
```

- **参数**

输入/输出	参数	描述
[in]	mqueueid	消息队列的 ID
[in]	msg	消息的缓冲区

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_mqueue_flush()

- **描述** 清空消息队列
- **函数原型**

```
ms_err_t ms_mqueue_flush(ms_handle_t mqueueid);
```

- **参数**

输入/输出	参数	描述
[in]	mqueueid	消息队列的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
static ms_handle_t test_tid;
static ms_handle_t mqueue_id;

static ms_uint8_t mqueue[16U * 4U];

static void test_thread(void *arg)
{
    ms_uint32_t msg;

    // flush msg
    ms_mqueue_flush(mqueue_id);

    ms_mqueue_wait(mqueue_id, &msg, MS_TIMEOUT_FOREVER);

    // msg come, msg will be 0xaa55aa55
    // do some thing
}

int main(int argc, char *argv[])
{
    ms_uint32_t msg;
```



```
    ms_mqueue_create("mqueue", mqueue, 16U, 4U, MS_WAIT_TYPE_PRIO, &mqueue_id);

    // post msg 0xabcdabcd
    msg = 0xabcdabcdUL;
    ms_mqueue_post(mqueue_id, &msg, MS_TIMEOUT_FOREVER);

    ms_thread_create("test", test_thread, MS_NULL,
                     4096U, 16U, 0U,
                     MS_THREAD_OPT_USER, &test_tid);

    ms_thread_sleep_s(1U);

    // post msg 0xaa55aa55
    msg = 0xaa55aa55UL;
    ms_mqueue_post(mqueue_id, &msg, MS_TIMEOUT_FOREVER);
    ms_thread_sleep_s(1U);

    return 0;
}
```

ms_mqueue_stat()

- **描述** 获得消息队列的状态
- **函数原型**

```
ms_err_t ms_mqueue_stat(ms_handle_t mqueueid, ms_mqueue_stat_t *stat);
```

- **参数**

输入/输出	参数	描述
[in]	mqueueid	消息队列的 ID
[out]	stat	消息队列的状态

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
static ms_handle_t mqueue_id;

static ms_uint8_t mqueue[16U * 4U];

int main(int argc, char *argv[])
{
    ms_uint32_t msg;
    ms_mqueue_stat_t stat;
```

```
    ms_mqueue_create("mqueue", mqueue, 16U, 4U, MS_WAIT_TYPE_PRIO, &mqueue_id);

    // post msg
    msg = 0xabcdabcdUL;
    ms_mqueue_post(mqueue_id, &msg, MS_TIMEOUT_FOREVER);

    ms_mqueue_stat(mqueue_id, &stat);
    // do some thing

    ms_mqueue_destroy(mqueue_id);

    return 0;
}
```


14 MS-RTOS 内存池

> 本章将介绍 MS-RTOS 内存池的使用。

内存池相关数据类型

类型	描述
ms_mempool_stat_t	内存池的状态信息结构类型

ms_mempool_stat_t

内存池的状态信息结构类型：

```
typedef struct {
    ms_size_t free_count;
} ms_mempool_stat_t;
```

参数	说明
free_count	空闲内存块数目

内存池相关 API

下表展示了内存池相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_mempool_create	●	●
ms_mempool_destroy	●	●
ms_mempool_alloc	●	●
ms_mempool_tryalloc	●	●
ms_mempool_free	●	●
ms_mempool_stat	●	●

ms_mempool_create()

- **描述** 创建一个内存池
- **函数原型**

```
ms_err_t ms_mempool_create(const char *name, ms_ptr_t mem_base, ms_uint16_t n_blk,
                           ms_size_t blk_size, ms_ipc_opt_t opt,
```

```
                                ms_handle_t *mpoolid);
```

- **参数**

输入/输出	参数	描述
[in]	name	内存池的名字，不能为空指针
[in]	mem_base	内存基地址
[in]	n_blk	有多少个内存块
[in]	blk_size	每个内存块的大小
[in]	opt	内存池的选项
[out]	mpoolid	内存池的 ID，不能为空指针

opt 参数可以取如下的值：

宏	含义
MS_WAIT_TYPE_PRIO	按优先级高低规则等待
MS_WAIT_TYPE_FIFO	按先来先服务规则等待

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用
- **示例**

```
static ms_handle_t mempool_id;

static ms_uint8_t mempool[10U * 1024U];

int main(int argc, char *argv[])
{
    ms_mempool_create("mempool", mempool, 10U, 1024U, MS_WAIT_TYPE_PRIO, &mempool_id);

    return 0;
}
```

ms_mempool_destroy()

- **描述** 销毁一个内存池
- **函数原型**

```
ms_err_t ms_mempool_destroy(ms_handle_t mpoolid);
```

- **参数**

输入/输出	参数	描述
[in]	mpoolid	内存池的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用
- **示例**

```
static ms_handle_t mempool_id;

static ms_uint8_t mempool[10U * 1024U];

int main(int argc, char *argv[])
{
    ms_mempool_create("mempool", mempool, 10U, 1024U, MS_WAIT_TYPE_PRIO, &mempool_id);

    // do some thing

    ms_mempool_destroy(mempool_id);

    return 0;
}
```

ms_mempool_alloc()

- **描述** 从内存池中分配一个内存块
- **函数原型**

```
ms_ptr_t ms_mempool_alloc(ms_handle_t mpoolid, ms_tick_t timeout, ms_err_t *perr);
```

- **参数**

输入/输出	参数	描述
[in]	mpoolid	内存池的 ID
[in]	timeout	等待的超时时间
[out]	perr	MS-RTOS 内核错误码，不关心时可以为 MS_NULL

- **返回值** 内存块指针，失败时返回 MS_NULL
- **注意事项** 中断中、内核锁定期间或内核启动前调用，timeout 必须为 `MS_TIMEOUT_NO_WAIT`
- **示例**


```c
static ms_handle_t mempool_id;

static ms_uint8_t mempool[10U * 1024U];

int main(int argc, char *argv[])
{
    ms_ptr_t ptr;

    ms_mempool_create("mempool", mempool, 10U, 1024U, MS_WAIT_TYPE_PRIO, &mempool_id);

    ptr = ms_mempool_alloc(mempool_id, 1000U, MS_NULL);

    // do some thing

    ms_mempool_destroy(mempool_id);

    return 0;
}
```

ms_mempool_tryalloc()

- **描述** 从内存池中尝试分配一个内存块
- **函数原型**

```c
ms_ptr_t ms_mempool_tryalloc(ms_handle_t mpoolid, ms_err_t *perr);
```

- **参数**

输入/输出	参数	描述
[in]	mpoolid	内存池的 ID
[out]	perr	MS-RTOS 内核错误码，不关心时可以为 MS_NULL

- **返回值** 内存块指针，失败时返回 MS_NULL
- **注意事项** 无
- **示例** 无

ms_mempool_free()

- **描述** 释放一个内存块到内存池
- **函数原型**

```c
ms_err_t ms_mempool_free(ms_handle_t mpoolid, ms_ptr_t ptr);
```

- **参数**

输入/输出	参数	描述
[in]	mpoolid	内存池的 ID
[in]	ptr	内存块指针

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
static ms_handle_t mempool_id;

static ms_uint8_t mempool[10U * 1024U];

int main(int argc, char *argv[])
{
    ms_ptr_t ptr;

    ms_mempool_create("mempool", mempool, 10U, 1024U, MS_WAIT_TYPE_PRIO, &mempool_id);

    ptr = ms_mempool_alloc(mempool_id, 1000U, MS_NULL);

    // do some thing

    ms_mempool_free(mempool_id, ptr);

    ms_mempool_destroy(mempool_id);

    return 0;
}
```

ms_mempool_stat()

- **描述** 获得内存池的状态信息
- **函数原型**

```
ms_err_t ms_mempool_stat(ms_handle_t mpoolid, ms_mempool_stat_t *stat);
```

- **参数**

输入/输出	参数	描述
[in]	mpoolid	内存池的 ID
[out]	stat	内存池的状态信息

- **返回值** MS-RTOS 内核错误码

- **注意事项** 无
- **示例**

```c
static ms_handle_t mempool_id;

static ms_uint8_t mempool[10U * 1024U];

int main(int argc, char *argv[])
{
    ms_mempool_stat_t stat;

    ms_mempool_create("mempool", mempool, 10U, 1024U, MS_WAIT_TYPE_PRIO, &mempool_id);

    // do some thing

    ms_mempool_stat(mempool_id, &stat);

    ms_mempool_destroy(mempool_id);

    return 0;
}
```


15 MS-RTOS 内存堆

> 本章将介绍 MS-RTOS 内存堆的使用。

内存堆相关 API

下表展示了内存堆相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_kheap_sbrk		●
ms_kmalloc		●
ms_kzalloc		●
ms_kmalloc_align		●
ms_kfree		●
ms_krealloc		●
ms_kcalloc		●
ms_malloc	●	●
ms_zalloc	●	●
ms_malloc_align	●	●
ms_free	●	●
ms_realloc	●	●
ms_calloc	●	●

ms_kheap_sbrk()

- **描述** 给内核内存堆增加内存
- **函数原型**

```
ms_err_t ms_kheap_sbrk(ms_ptr_t base, ms_size_t size);
```

- **参数**

输入/输出	参数	描述
[in]	base	内存基地址，不能为空指针
[in]	size	内存大小，必须 >= 2048

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核锁定期间调用，最多增加一个区域
- **示例**

```
static ms_uint8_t mem[4096U];

int xxx(void)
{
    ms_kheap_sbrk(mem, sizeof(mem));

    return 0;
}
```

ms_kmalloc()

- **描述** 从内核内存堆中分配指定大小的内存
- **函数原型**

```
ms_ptr_t ms_kmalloc(ms_size_t size);
```

- **参数**

输入/输出	参数	描述
[in]	size	需要分配的内存大小，必须 > 0

- **返回值** 内存指针，失败时返回 MS_NULL
- **注意事项** 不能在中断中调用，不能在内核锁定期间调用
- **示例**

```
int xxx(void)
{
    ms_ptr_t ptr = ms_kmalloc(100U);

    // do some thing

    return 0;
}
```

ms_kzalloc()

- **描述** 从内核内存堆中分配指定大小的内存并清零
- **函数原型**


```
ms_ptr_t ms_kzalloc(ms_size_t size);
```

- **参数**

输入/输出	参数	描述
[in]	size	需要分配的内存大小，必须 > 0

- **返回值** 内存指针，失败时返回 MS_NULL
- **注意事项** 不能在中断中调用，不能在内核锁定期间调用
- **示例**

```
int xxx(void)
{
    ms_ptr_t ptr = ms_kzalloc(100U);

    // do some thing

    return 0;
}
```

ms_kfree()

- **描述** 释放内存到内核内存堆
- **函数原型**

```
ms_ptr_t ms_kfree(ms_ptr_t ptr);
```

- **参数**

输入/输出	参数	描述
[in]	ptr	内存指针，不能为空指针

- **返回值** 成功返回 MS_NULL，失败返回 ptr
- **注意事项** 不能在中断中调用，不能在内核锁定期间调用
- **示例**

```
int xxx(void)
{
    ms_ptr_t ptr = ms_kmalloc(100U);

    // do some thing
```



```
    ms_kfree(ptr);

    return 0;
}
```

ms_kmalloc_align()

- **描述** 从内核内存堆中分配指定大小指定对齐要求的内存
- **函数原型**

```
ms_ptr_t ms_kmalloc_align(ms_size_t size, ms_size_t align);
```

- **参数**

输入/输出	参数	描述
[in]	size	需要分配的内存大小，必须 > 0
[in]	align	需要对齐到多少个字节，必须 > 0

- **返回值** 成功返回内存指针，失败时返回 MS_NULL
- **注意事项** 不能在中断中调用，不能在内核锁定期间调用
- **示例**

```
int xxx(void)
{
    ms_ptr_t ptr = ms_kmalloc_align(100U, 32U);

    // do some thing

    ms_kfree(ptr);

    return 0;
}
```

ms_krealloc()

- **描述** 从内核内存堆中重新分配指定大小的内存
- **函数原型**

```
ms_ptr_t ms_krealloc(ms_ptr_t ptr, ms_size_t size);
```

- **参数**

输入/输出	参数	描述
[in]	ptr	内存指针，空指针时未分配
[in]	size	需要分配的内存大小，必须 > 0

- **返回值** 成功返回内存指针，失败时返回 MS_NULL
- **注意事项** 不能在中断中调用，不能在内核锁定期间调用
- **示例**

```
int xxx(void)
{
    ms_ptr_t ptr = ms_kmalloc(100U);

    // do some thing

    ptr = ms_krealloc(ptr, 200U);

    // do some thing

    ms_kfree(ptr);

    return 0;
}
```

ms_kcalloc()

- **描述** 从内核内存堆中分配 N 个指定大小的内存块，并清零
- **函数原型**

```
ms_ptr_t ms_kcalloc(ms_size_t n_blk, ms_size_t blk_size);
```

- **参数**

输入/输出	参数	描述
[in]	n_blk	N 个内存块，必须 > 0
[in]	blk_size	每个内存块的大小，必须 > 0

- **返回值** 成功返回内存指针，失败时返回 MS_NULL
- **注意事项** 不能在中断中调用，不能在内核锁定期间调用
- **示例**

```
int xxx(void)
{
```



```
    ms_ptr_t ptr = ms_kcalloc(10U, 100U);

    // do some thing

    ms_kfree(ptr);

    return 0;
}
```

ms_malloc()

- **描述** 从内存堆中分配指定大小的内存
- **函数原型**

```
ms_ptr_t ms_malloc(ms_size_t size);
```

- **参数**

输入/输出	参数	描述
[in]	size	需要分配的内存大小，必须 > 0

- **返回值** 成功返回内存指针，失败时返回 MS_NULL
- **注意事项** 不能在中断中调用，不能在内核锁定期间调用
- **示例**

```
int main(int argc, char *argv[])
{
    ms_ptr_t ptr = ms_malloc(100U);

    // do some thing

    return 0;
}
```

ms_zalloc()

- **描述** 从内存堆中分配指定大小的内存并清零
- **函数原型**

```
ms_ptr_t ms_zalloc(ms_size_t size);
```

- **参数**

输入/输出	参数	描述
[in]	size	需要分配的内存大小，必须 > 0

- **返回值** 成功返回内存指针，失败时返回 MS_NULL
- **注意事项** 不能在中断中调用，不能在内核锁定期间调用
- **示例**

```
int main(int argc, char *argv[])
{
    ms_ptr_t ptr = ms_zalloc(100U);

    // do some thing

    return 0;
}
```

ms_free()

- **描述** 释放内存到内存堆
- **函数原型**

```
ms_ptr_t ms_free(ms_ptr_t ptr);
```

- **参数**

输入/输出	参数	描述
[in]	ptr	内存指针，必须不为空指针

- **返回值** 成功返回 MS_NULL，失败返回 ptr
- **注意事项** 不能在中断中调用，不能在内核锁定期间调用
- **示例**

```
int main(int argc, char *argv[])
{
    ms_ptr_t ptr = ms_malloc(100U);

    // do some thing

    ms_free(ptr);

    return 0;
}
```


ms_malloc_align()

- **描述** 从内存堆中分配指定大小指定对齐要求的内存
- **函数原型**

```
ms_ptr_t ms_malloc_align(ms_size_t size, ms_size_t align);
```

- **参数**

输入/输出	参数	描述
[in]	size	需要分配的内存大小，必须 > 0
[in]	align	需要对齐到多少个字节，必须 > 0

- **返回值** 成功返回内存指针，失败时返回 MS_NULL
- **注意事项** 不能在中断中调用，不能在内核锁定期间调用
- **示例**

```
int main(int argc, char *argv[])
{
    ms_ptr_t ptr = ms_malloc_align(100U, 32U);

    // do some thing

    ms_free(ptr);

    return 0;
}
```

ms_realloc()

- **描述** 从内存堆中重新分配指定大小的内存
- **函数原型**

```
ms_ptr_t ms_realloc(ms_ptr_t ptr, ms_size_t size);
```

- **参数**

输入/输出	参数	描述
[in]	ptr	内存指针，空指针时未分配
[in]	size	需要分配的内存大小，必须 > 0

- **返回值** 成功返回内存指针，失败时返回 MS_NULL

- **注意事项** 不能在中断中调用，不能在内核锁定期间调用
- **示例**

```
int main(int argc, char *argv[])
{
    ms_ptr_t ptr = ms_malloc(100U);

    // do some thing

    ptr = ms_realloc(ptr, 200U);

    // do some thing

    ms_free(ptr);

    return 0;
}
```

ms_calloc()

- **描述** 从内存堆中分配 N 个指定大小的内存块，并清零
- **函数原型**

```
ms_ptr_t ms_calloc(ms_size_t n_blk, ms_size_t blk_size);
```

- **参数**

输入/输出	参数	描述
[in]	n_blk	N 个内存块，必须 > 0
[in]	blk_size	每个内存块的大小，必须 > 0

- **返回值** 成功返回内存指针，失败时返回 MS_NULL
- **注意事项** 不能在中断中调用，不能在内核锁定期间调用
- **示例**

```
int main(int argc, char *argv[])
{
    ms_ptr_t ptr = ms_calloc(10U, 100U);

    // do some thing

    ms_free(ptr);
```



```
    return 0;
}
```


16 MS-RTOS 软件定时器

> 本章将介绍 MS-RTOS 软件定时器的使用。

软件定时器相关数据类型

类型	描述
ms_timer_status_t	软件定时器的运行状态类型
ms_timer_opt_t	软件定时器的选项类型
ms_timer_callback_t	软件定时器的回调函数指针类型
ms_timer_stat_t	软件定时器的状态信息结构类型

ms_timer_status_t

软件定时器的运行状态为以下的宏：

宏	含义
MS_TIMER_STATUS_UNUSED	软件定时器已经被销毁
MS_TIMER_STATUS_STOP	软件定时器已经停止
MS_TIMER_STATUS_RUNNING	软件定时器正在运行

ms_timer_opt_t

软件定时器的选项为以下的宏：

宏	含义
MS_TIMER_OPT_ONE_SHOT	只启动一次
MS_TIMER_OPT_PERIODIC	周期性启动

ms_timer_callback_t

软件定时器回调函数指针类型无返回值，有一个 `ms_ptr_t` 类型参数：

```
typedef void (*ms_timer_callback_t)(ms_ptr_t arg);
```

以下是一个软件定时器回调函数定义的示例：

```
static void xxx_timer(ms_ptr_t arg)
{
```



```
    // do some thing
}
```

ms_timer_stat_t

软件定时器的状态信息结构类型：

```
typedef struct {
    ms_tick_t          remain;
    ms_timer_status_t  status;
} ms_timer_stat_t;
```

参数	说明
remain	软件定时器的到期剩余时间（以嘀嗒为单位）
status	软件定时器的工作状态

软件定时器相关 API

下表展示了软件定时器相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_timer_create	●	●
ms_timer_destroy	●	●
ms_timer_start	●	●
ms_timer_stop	●	●
ms_timer_stat	●	●

ms_timer_create()

- **描述** 创建一个软件定时器
- **函数原型**

```
ms_err_t ms_timer_create(const char *name, ms_timer_callback_t callback, ms_ptr_t arg,
                         ms_handle_t *timerid);
```

- **参数**

输入/输出	参数	描述
[in]	name	软件定时器的名字，不能为空指针

输入/输出	参数	描述
[in]	callback	回调函数指针，不能为空指针
[in]	arg	回调函数参数，可以为空指针
[out]	timerid	软件定时器的 ID，不能为空指针

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核锁定期间调用
- **示例**

```
static ms_handle_t timer_id;

static void timer_callback(ms_ptr_t arg)
{
    // do some thing
}

int main(int argc, char *argv[])
{
    ms_timer_create("timer", timer_callback, MS_NULL, &timer_id);

    // do some thing

    return 0;
}
```

ms_timer_destroy()

- **描述** 销毁一个软件定时器
- **函数原型**

```
ms_err_t ms_timer_destroy(ms_handle_t timerid);
```

- **参数**

输入/输出	参数	描述
[in]	timerid	软件定时器的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核锁定期间调用
- **示例**


```c
static ms_handle_t timer_id;

static void timer_callback(ms_ptr_t arg)
{
    // do some thing
}

int main(int argc, char *argv[])
{
    ms_timer_create("timer", timer_callback, MS_NULL, &timer_id);

    // do some thing

    ms_timer_destroy(timer_id);

    return 0;
}
```

ms_timer_start()

- **描述** 启动一个软件定时器
- **函数原型**

```c
ms_err_t ms_timer_start(ms_handle_t timerid, ms_tick_t delay, ms_tick_t period,
                        ms_timer_opt_t opt);
```

- **参数**

输入/输出	参数	描述
[in]	timerid	软件定时器的 ID
[in]	delay	首先延时多少个嘀嗒（单次触发定时器如果 delay > 0，将会忽略 period）
[in]	period	软件定时器的周期启动时间（以嘀嗒为单位）
[in]	opt	软件定时器的选项

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核锁定期间调用
- **示例**

```c
static ms_handle_t timer_id;

static void timer_callback(ms_ptr_t arg)
{
    ms_printf("in timer\n");
}
```

```c
int main(int argc, char *argv[])
{
    ms_timer_create("timer", timer_callback, MS_NULL, &timer_id);

    ms_timer_start(timer_id, 0U, 2000U, MS_TIMER_OPT_PERIODIC);

    ms_thread_sleep_s(10);

    // do some thing

    ms_timer_destroy(timer_id);

    return 0;
}
```

ms_timer_stop()

- **描述** 停止一个软件定时器
- **函数原型**

```c
ms_err_t ms_timer_stop(ms_handle_t timerid);
```

- **参数**

输入/输出	参数	描述
[in]	timerid	软件定时器的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核锁定期间调用
- **示例**

```c
static ms_handle_t timer_id;

static void timer_callback(ms_ptr_t arg)
{
    ms_printf("in timer\n");
}

int main(int argc, char *argv[])
{
    ms_timer_create("timer", timer_callback, MS_NULL, &timer_id);

    ms_timer_start(timer_id, 0U, 2000U, MS_TIMER_OPT_PERIODIC);

    ms_thread_sleep_s(10);
```

```
    ms_timer_stop(timer_id);

    // do some thing

    ms_timer_destroy(timer_id);

    return 0;
}
```

ms_timer_stat()

- **描述** 获得软件定时器的状态信息
- **函数原型**

```
ms_err_t ms_timer_stat(ms_handle_t timerid, ms_timer_stat_t *stat);
```

- **参数**

输入/输出	参数	描述
[in]	timerid	软件定时器的 ID
[out]	stat	软件定时器的状态信息

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核锁定期间调用
- **示例**

```
static ms_handle_t timer_id;

static void timer_callback(ms_ptr_t arg)
{
    ms_printf("in timer\n");
}

int main(int argc, char *argv[])
{
    ms_timer_stat_t stat;

    ms_timer_create("timer", timer_callback, MS_NULL, &timer_id);

    ms_timer_start(timer_id, 0U, 2000U, MS_TIMER_OPT_PERIODIC);

    ms_thread_sleep_s(10);

    ms_timer_stat(timer_id, &stat);

    ms_printf("timer remain time %d\n", stat.remain);
```



```
    // do some thing

    ms_timer_destroy(timer_id);

    return 0;
}
```


17 MS-RTOS 原子量

> 本章将介绍 MS-RTOS 原子量的使用。

原子量介绍

原子量的操作都是原子的（过去人们认为原子不可再分，故称为原子量），外部可以直接使用原子量的 API 操作原子量而无需加锁，可大大提高操作、访问的速度。

原子量相关数据类型

类型	描述
ms_atomic_t	原子量类型

ms_atomic_t

原子量的类型为 `ms_atomic_t`，原子量使用前需要定义，它可以是一个全局变量，也可以嵌入到其它数据类型当中（如结构体）作为一个成员变量。

```
typedef struct {
    volatile int counter;
} ms_atomic_t;
```

参数	说明
counter	原子量的当前值

原子量相关 API

下表展示了原子量相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_atomic_read	●	●
ms_atomic_set	●	●
ms_atomic_add	●	●
ms_atomic_sub	●	●
ms_atomic_and	●	●
ms_atomic_or	●	●
ms_atomic_xor	●	●

API	用户空间	内核空间
ms_atomic_inc	●	●
ms_atomic_dec	●	●
ms_atomic_cas	●	●

ms_atomic_read()

- **描述** 读原子量的值
- **函数原型**

```
int ms_atomic_read(const ms_atomic_t *v);
```

- **参数**

输入/输出	参数	描述
[in]	v	原子量指针

- **返回值** 原子量的值
- **注意事项** 无
- **示例** 无

ms_atomic_set()

- **描述** 设置原子量的值
- **函数原型**

```
void ms_atomic_set(ms_atomic_t *v, int i);
```

- **参数**

输入/输出	参数	描述
[in]	v	原子量指针
[in]	i	原子量的新值

- **返回值** 无
- **注意事项** 无
- **示例**


```c
static ms_atomic_t g_atomic;

int main(int argc, char *argv[])
{
    ms_atomic_set(&g_atomic, 1);
    ms_printf("value: %d\n", ms_atomic_read(&g_atomic));

    ms_atomic_set(&g_atomic, 10);
    ms_printf("value: %d\n", ms_atomic_read(&g_atomic));

    ms_atomic_set(&g_atomic, 100);
    ms_printf("value: %d\n", ms_atomic_read(&g_atomic));

    return 0;
}
```

ms_atomic_add()

- **描述** 原子量加上一个指定的值
- **函数原型**

```c
int ms_atomic_add(ms_atomic_t *v, int i);
```

- **参数**

输入/输出	参数	描述
[in]	v	原子量指针
[in]	i	需要加上的值

- **返回值** 原子量的新值
- **注意事项** 无
- **示例**

```c
static ms_atomic_t g_atomic;

int main(int argc, char *argv[])
{
    int value;

    ms_atomic_set(&g_atomic, 100);

    value = ms_atomic_add(&g_atomic, -10);
    ms_printf("value: %d\n", value);

    value = ms_atomic_add(&g_atomic, 20);
    ms_printf("value: %d\n", value);
```

```
    return 0;
}
```

ms_atomic_sub()

- **描述** 原子量减去一个指定的值
- **函数原型**

```
int ms_atomic_sub(ms_atomic_t *v, int i);
```

- **参数**

输入/输出	参数	描述
[in]	v	原子量指针
[in]	i	需要减去的值

- **返回值** 原子量的新值
- **注意事项** 无
- **示例**

```
static ms_atomic_t g_atomic;

int main(int argc, char *argv[])
{
    int value;

    ms_atomic_set(&g_atomic, 100);

    value = ms_atomic_sub(&g_atomic, -10);
    ms_printf("value: %d\n", value);

    value = ms_atomic_sub(&g_atomic, 20);
    ms_printf("value: %d\n", value);

    return 0;
}
```

ms_atomic_and()

- **描述** 原子量与上一个指定的值
- **函数原型**


```
int ms_atomic_and(ms_atomic_t *v, int i);
```

- **参数**

输入/输出	参数	描述
[in]	v	原子量指针
[in]	i	需要与上的值

- **返回值** 原子量的新值
- **注意事项** 无
- **示例**

```
static ms_atomic_t g_atomic;

int main(int argc, char *argv[])
{
    int value;

    ms_atomic_set(&g_atomic, 0x12345678);

    value = ms_atomic_and(&g_atomic, 0x000000FF);
    ms_printf("value: %x\n", value);

    value = ms_atomic_and(&g_atomic, 0x0000FF00);
    ms_printf("value: %x\n", value);

    value = ms_atomic_and(&g_atomic, 0x00FF0000);
    ms_printf("value: %x\n", value);

    value = ms_atomic_and(&g_atomic, 0xFF000000);
    ms_printf("value: %x\n", value);

    return 0;
}
```

ms_atomic_or()

- **描述** 原子量或上一个指定的值
- **函数原型**

```
int ms_atomic_or(ms_atomic_t *v, int i);
```

- **参数**

输入/输出	参数	描述
[in]	v	原子量指针
[in]	i	需要或上的值

- **返回值** 原子量的新值
- **注意事项** 无
- **示例**

```
static ms_atomic_t g_atomic;

int main(int argc, char *argv[])
{
    int value;

    ms_atomic_set(&g_atomic, 0x12345678);

    value = ms_atomic_or(&g_atomic, 0x000000FF);
    ms_printf("value: %x\n", value);

    value = ms_atomic_or(&g_atomic, 0x0000FF00);
    ms_printf("value: %x\n", value);

    value = ms_atomic_or(&g_atomic, 0x00FF0000);
    ms_printf("value: %x\n", value);

    value = ms_atomic_or(&g_atomic, 0xFF000000);
    ms_printf("value: %x\n", value);

    return 0;
}
```

ms_atomic_xor()

- **描述** 原子量异或上一个指定的值
- **函数原型**

```
int ms_atomic_xor(ms_atomic_t *v, int i);
```

- **参数**

输入/输出	参数	描述
[in]	v	原子量指针
[in]	i	需要异或上的值

- **返回值** 原子量的新值
- **注意事项** 无
- **示例** 无

ms_atomic_inc()

- **描述** 原子量自增
- **函数原型**

```
int ms_atomic_inc(ms_atomic_t *v);
```

- **参数**

输入/输出	参数	描述
[in]	v	原子量指针

- **返回值** 原子量的新值
- **注意事项** 无
- **示例**

```
static ms_atomic_t g_atomic;

int main(int argc, char *argv[])
{
    int value;

    ms_atomic_set(&g_atomic, 100);

    value = ms_atomic_inc(&g_atomic);
    ms_printf("value: %d\n", value);

    value = ms_atomic_inc(&g_atomic);
    ms_printf("value: %d\n", value);

    return 0;
}
```

ms_atomic_dec()

- **描述** 原子量自减
- **函数原型**

```
int ms_atomic_dec(ms_atomic_t *v);
```

- **参数**

输入/输出	参数	描述
[in]	v	原子量指针

- **返回值** 原子量的新值
- **注意事项** 无
- **示例**

```
static ms_atomic_t g_atomic;

int main(int argc, char *argv[])
{
    int value;

    ms_atomic_set(&g_atomic, 100);

    value = ms_atomic_dec(&g_atomic);
    ms_printf("value: %d\n", value);

    value = ms_atomic_dec(&g_atomic);
    ms_printf("value: %d\n", value);

    return 0;
}
```

ms_atomic_cas()

- **描述** 原子量比较和交换（Compare-and-swap，CAS）
- **函数原型**

```
int ms_atomic_cas(ms_atomic_t *v, int old_value, int new_value);
```

- **参数**

输入/输出	参数	描述
[in]	v	原子量指针
[in]	old_value	原子量的旧值
[in]	new_value	原子量的新值

- **返回值** 原子量的旧值，如果返回的原子量旧值与传入的原子量旧值相等，则说明操作成功，传入的原子量新值被成功写入；如果不相等，则说明操作失败，传入的原子量新值没有成功写入
- **注意事项** 无

- **示例** 无

18 MS-RTOS 进程

> 本章将介绍 MS-RTOS 进程相关接口的使用。

进程相关数据类型

类型	描述
ms_pid_t	进程 ID
ms_sigset_t	信号集
ms_sighandler_t	信号处理函数

ms_sigset_t

信号集类型：

```
typedef ms_ulong_t ms_sigset_t;
```

以下是 MS-RTOS 预定义的两个信号，用户可用 3 ~ 31 编号的信号：

参数	值	说明
MS_SINGAL_SHUTDOWN	1	关机信号
MS_SINGAL_POWER_FAIL	2	掉电信号

ms_sighandler_t

信号处理函数类型有一个信号编号的参数，无返回值：

```
typedef void (*ms_sighandler_t)(int sig);
```

以下是一个信号处理函数定义的示例：

```
static void shutdown_handler(int sig)
{
    // do some thing
}
```

进程相关 API

下表展示了进程相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_process_mem_sbrk		●
ms_process_create		●
ms_process_self	●	●
ms_process_kill	●	●
ms_process_exit	●	
ms_process_find	●	●
ms_process_signal	●	
ms_process_sigprocmask	●	
ms_process_sigqueue	●	●
ms_process_sigbroadcast	●	●

ms_process_mem_sbrk()

- **描述** 添加一个进程内存区域
- **函数原型**

```
ms_err_t ms_process_mem_sbrk(ms_addr_t base, ms_size_t size);
```

- **参数**

输入/输出	参数	描述
[in]	base	内存基地址
[in]	size	内存大小

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用
- **示例** 无

ms_process_create()

- **描述** 创建一个进程
- **函数原型**

```
ms_err_t ms_process_create(const char *name, ms_const_ptr_t app_base, ms_size_t mem_size,
                           ms_size_t stk_size, ms_prio_t prio, ms_time_slice_t time_slice,
                           ms_uint32_t argc, char *argv[],
```



```
                        const ms_printf_func_t debug_printf,
                        ms_pid_t *pid);
```

- **参数**

输入/输出	参数	描述
[in]	name	进程的名字，不能为空指针
[in]	app_base	APP 镜像的地址
[in]	mem_size	进程内存的大小
[in]	stk_size	进程主线程的堆栈大小
[in]	prio	进程主线程的优先级
[in]	time_slice	进程主线程的时间片
[in]	argc	字符串指针参数数目
[in]	argv	字符串指针参数数组
[in]	debug_printf	调试信息打印函数指针，非空指针为调试模式
[out]	pid	进程的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核锁定期间调用
- **示例**

```
void xxx(void)
{
    ms_err_t err = ms_process_create("process", (ms_ptr_t)0x20040000UL, 65536, 4096U, 9U, 0U,
                                     0U, MS_NULL,
                                     MS_NULL, MS_NULL);

    // do some thing
}
```

ms_process_self()

- **描述** 获得当前进程的 ID
- **函数原型**

```
ms_pid_t ms_process_self(void);
```

- **参数** 无

- **返回值** 当前进程的 ID
- **注意事项** 不能在中断中调用
- **示例**

```
int main(int argc, char *argv[])
{
    ms_pid_t pid = ms_process_self();

    // do some thing

    return 0;
}
```

ms_process_kill()

- **描述** 杀死一个指定的进程
- **函数原型**

```
ms_err_t ms_process_kill(ms_pid_t pid);
```

- **参数**

输入/输出	参数	描述
[in]	pid	进程的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用，不能在内核锁定期间调用
- **示例**

```
int main(int argc, char *argv[])
{
    // do some thing

    ms_process_kill(ms_process_self());

    return 0;
}
```

ms_process_exit()

- **描述** 进程主动退出
- **函数原型**


```
ms_err_t ms_process_exit(void);
```

- **参数** 无
- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用
- **示例**

```
int main(int argc, char *argv[])
{
    // do some thing

    ms_process_exit();

    return 0;
}
```

ms_process_find()

- **描述** 通过名字查找进程
- **函数原型**

```
ms_err_t ms_process_find(const char *name, ms_pid_t *pid);
```

- **参数**

输入/输出	参数	描述
[in]	name	进程的名字
[out]	pid	进程的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 不能在中断中调用
- **示例** 无

ms_process_signal()

- **描述** 安装指定信号的处理函数
- **函数原型**

```
ms_err_t ms_process_signal(int sig, ms_sighandler_t sighandler);
```

- **参数**

输入/输出	参数	描述
[in]	sig	信号的编号
[in]	sighandler	信号的处理函数

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_process_sigprocmask()

- **描述** 设置信号屏蔽掩码
- **函数原型**

```
ms_err_t ms_process_sigprocmask(const ms_sigset_t *nset, ms_sigset_t *oset);
```

- **参数**

输入/输出	参数	描述
[in]	nset	新的信号屏蔽掩码
[out]	oset	旧的信号屏蔽掩码

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_process_sigqueue()

- **描述** 给指定进程发信号
- **函数原型**

```
ms_err_t ms_process_sigqueue(ms_pid_t pid, int sig, ms_tick_t timeout);
```

- **参数**

输入/输出	参数	描述
[in]	pid	进程的 ID
[in]	sig	信号的编号
[in]	timeout	超时时间

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_process_sigbroadcast()

- **描述** 给所有进程发信号
- **函数原型**

```
ms_err_t ms_process_sigbroadcast(int sig, ms_tick_t timeout);
```

- **参数**

输入/输出	参数	描述
[in]	sig	信号的编号
[in]	timeout	超时时间

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

19 MS-RTOS 文件 IO

> 本章将介绍 MS-RTOS 文件 IO 相关接口的使用。

文件操作编程接口

MS-RTOS IO 子系统管理了 MS-RTOS 上的各种各样的设备，同时隔离了不同文件系统上层与底层接口的差异，上层可以使用统一的类 posix 的 API 操作设备和读写文件、目录、设备。

文件 IO 相关数据类型

类型	描述
ms_mode_t	文件模式类型
mode_t	posix 文件模式类型
ms_off_t	文件偏移量类型
off_t	posix 文件偏移量类型
ms_dev_t	设备号类型
dev_t	posix 设备号类型
ms_stat_t	文件状态类型
struct stat	posix 文件状态类型
ms_fd_set_t	文件描述符集类型
fd_set	posix 文件描述符集类型
ms_dirent_t	目录项类型
struct dirent	posix 目录项类型
MS_DIR	目录流类型
DIR	posix 目录流类型
ms_statvfs_t	文件系统状态类型
ms_nfds_t	文件描述符集数量类型
nfds_t	posix 文件描述符集数量类型
ms_pollevent_t	poll 事件类型
ms_pollfd_t	poll 文件描述符类型
struct pollfd	posix poll 文件描述符类型

ms_stat_t 与 struct stat

ms_stat_t 与 struct stat 类型定义如下：

```
typedef struct stat {
    dev_t       st_dev;     /* ID of device containing file */
    ino_t       st_ino;     /* inode number */
    mode_t      st_mode;    /* protection */
    nlink_t     st_nlink;   /* number of hard links */
    uid_t       st_uid;     /* user ID of owner */
    gid_t       st_gid;     /* group ID of owner  */
    dev_t       st_rdev;    /* device ID (if special file) */
    off_t       st_size;    /* total size, in bytes */
    time_t      st_atime;   /* time of last access */
    long        st_spare1;
    time_t      st_mtime;   /* time of last modification */
    long        st_spare2;
    time_t      st_ctime;   /* time of last status change */
    long        st_spare3;
    blksize_t   st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t    st_blocks;  /* number of blocks allocated */
    long        st_spare4[2];
} ms_stat_t;
```

参数	说明
st_dev	文件所在的设备的 ID
st_ino	inode 号
st_mode	文件的访问模式
st_nlink	文件的硬链接数量
st_uid	文件拥有者的用户 ID
st_gid	文件拥有者的组 ID
st_rdev	特殊文件的设备 ID
st_size	文件总大小
st_atime	文件最后一次访问的时间
st_mtime	文件最后一次修改的时间
st_ctime	文件最后一次状态改变的时间
st_blksize	文件系统 IO 传输的块大小
st_blocks	文件占用的文件系统块数量

ms_fd_set_t 与 fd_set

ms_fd_set_t 与 fd_set 可用以下的宏来进行操作：

类型	描述
FD_SET(n, p)	把文件描述符设置到文件描述符集
FD_CLR(n, p)	把文件描述符从文件描述符集中清除
FD_ISSET(n, p)	判断文件描述符是否在文件描述符集中设置
FD_ZERO(p)	清零文件描述符集

ms_dirent_t 与 struct dirent

ms_dirent_t 与 struct dirent 类型定义如下：

```
typedef struct dirent {
    char            d_name[MS_IO_NAME_BUF_SIZE];

#define DT_UNKNOWN  0U
#define DT_FIFO     1U
#define DT_CHR      2U
#define DT_DIR      4U
#define DT_BLK      6U
#define DT_REG      8U
#define DT_LNK      10U
#define DT_SOCK     12U
#define DT_WHT      14U
    ms_uint8_t      d_type;
} ms_dirent_t;
```

宏	值	含义
DT_UNKNOWN	0	未知设备类型
DT_FIFO	1	FIFO 设备类型
DT_CHR	2	字符设备类型
DT_DIR	4	目录类型
DT_BLK	6	块设备类型
DT_REG	8	普通文件类型
DT_LNK	10	符号链接文件类型
DT_SOCK	12	socket 文件设备类型
DT_WHT	14	whiteout

ms_statvfs_t

ms_statvfs_t 类型定义如下：

```
typedef struct ms_statvfs {
    ms_uint32_t     f_bsize;    /* Optimal transfer block size */
    ms_uint32_t     f_frsize;   /* Allocation unit size */
    ms_uint32_t     f_blocks;   /* Size of FS in f_frsize units */
    ms_uint32_t     f_bfree;    /* Number of free blocks */
    ms_uint32_t     f_files;    /* Number of avail files */
    ms_uint32_t     f_ffree;    /* Number of free files */
    const char     *f_dev;      /* Block device path */
    const char     *f_mnt;      /* Mount point path */
    const char     *f_fsname;   /* File system name */
} ms_statvfs_t;
```

参数	说明
f_bsize	最佳的传输块大小
f_frsize	分配单元大小
f_blocks	分配单元数量
f_bfree	空闲块数量
f_files	有效的文件数
f_ffree	空闲的文件数
f_dev	块设备路径
f_mnt	挂载点路径
f_fsname	文件系统名称

ms_pollevent_t

ms_pollevent_t 类型定义如下：

宏	值	含义
POLLIN	0x01U	普通或优先级带数据可读（等效于 POLLRDNORM | POLLRDBAND）
POLLRDNORM	0x01U	普通数据可读
POLLRDBAND	0x01U	优先级带数据可读
POLLPRI	0x01U	高优先级数据可读

宏	值	含义
POLLOUT	0x02U	普通数据可写
POLLWRNORM	0x02U	普通数据可写，等价于 POLLOUT
POLLWRBAND	0x02U	优先级带数据可写
POLLERR	0x04U	发生错误
POLLHUP	0x08U	发生挂起
POLLNVAL	0x10U	描述字不是一个打开的文件

ms_pollfd_t 与 struct pollfd

ms_pollfd_t 与 struct pollfd 类型定义如下：

```
typedef struct pollfd {
    /*
     * Standard fields
     */
    int             fd;             /*  The descriptor being polled         */
    ms_pollevent_t  events;         /*  The input event flags               */
    ms_pollevent_t  revents;        /*  The output event flags              */

    /*
     * Non-standard fields used internally by MS-RTOS
     */
    ms_handle_t     semcid;         /*  Semaphore used to post output event */
    ms_ptr_t        priv;           /*  For use by drivers                  */
} ms_pollfd_t;
```

参数	说明
fd	等待的文件描述符
events	指定的等待事件
revents	等待到的事件

文件 IO 相关 API

原生文件 IO 相关 API

下表展示了文件 IO 相关 API 在两个权限空间下是否可用：

API	功能	用户空间	内核空间
当前工作目录			
ms_io_chdir	改变当前工作目录	●	●
ms_io_getcwd	获得当前工作目录	●	●
文件			
ms_io_creat	创建文件	●	●
ms_io_open	打开文件	●	●
ms_io_close	关闭文件	●	●
ms_io_fcntl	控制文件描述符	●	●
ms_io_fstat	获得文件状态	●	●
ms_io_isatty	判断是否为一个 TTY 设备	●	●
ms_io_fsync	回写文件 CACHE 到磁盘	●	●
ms_io_fdatasync	回写文件数据 CACHE 到磁盘	●	●
ms_io_ftruncate	文件截断	●	●
ms_io_ioctl	IO 控制	●	●
ms_io_dup	复制文件描述符	●	●
ms_io_dup2	复制文件描述符到指定的文件描述符	●	●
ms_io_lseek	调整文件读写指针	●	●
ms_io_read	读文件	●	●
ms_io_write	写文件	●	●
ms_io_tell	获得文件当前读写指针	●	●
ms_io_poll	等待文件集事件	●	●
ms_io_select	等待文件集事件	●	●
文件系统			
ms_io_link	链接文件	●	●
ms_io_rename	文件重命名	●	●
ms_io_stat	获得文件状态	●	●
ms_io_lstat	获得文件状态	●	●

API	功能	用户空间	内核空间
ms_io_statvfs	获得文件系统状态	●	●
ms_io_unlink	删除文件	●	●
ms_io_mkdir	创建目录	●	●
ms_io_rmdir	删除目录	●	●
ms_io_access	判断文件是否可以访问	●	●
ms_io_truncate	截断文件	●	●
ms_io_sync	回写文件系统 CACHE 到磁盘	●	●
ms_io_mkfs	格式化磁盘		●
ms_io_mount	挂载磁盘		●
ms_io_mount_ex	挂载磁盘		●
ms_io_unmount	卸载挂载点		●
目录流			
ms_io_opendir	打开目录	●	●
ms_io_closedir	关闭目录	●	●
ms_io_readdir_r	读目录项	●	●
ms_io_rewinddir	重置目录流读指针	●	●
ms_io_seekdir	调整目录流读指针	●	●
ms_io_telldir	获得目录流当前读指针	●	●

ms_io_chdir()

- **描述** 切换当前工作目录
- **函数原型**

```
int ms_io_chdir(const char *path);
```

- **参数**

输入/输出	参数	描述
[in]	path	需要切换到的目录路径

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无

- **示例** 无

ms_io_getcwd()

- **描述** 获得当前工作目录路径
- **函数原型**

```
char *ms_io_getcwd(char *buf, ms_size_t size);
```

- **参数**

输入/输出	参数	描述
[out]	buf	当前工作目录路径缓冲区
[in]	size	当前工作目录路径缓冲区大小

- **返回值** 成功返回当前工作目录路径缓冲区的首地址，失败返回 MS_NULL
- **注意事项** 无
- **示例** 无

ms_io_creat()

- **描述** 创建文件
- **函数原型**

```
int ms_io_creat(const char *path, ms_mode_t mode);
```

- **参数**

输入/输出	参数	描述
[in]	path	需要创建的文件路径
[in]	mode	文件模式

- **返回值** 成功返回文件描述符，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_open()

- **描述** 打开文件
- **函数原型**


```
int ms_io_open(const char *path, int oflag, ms_mode_t mode);
```

- **参数**

输入/输出	参数	描述
[in]	path	需要打开的文件路径
[in]	oflag	文件打开标志
[in]	mode	文件模式（创建文件时有用）

- **oflag 文件打开标志**

宏	描述
O_RDONLY	只读打开
O_WRONLY	只写打开
O_RDWR	可读可写打开
O_APPEND	追加方式打开
O_CREAT	文件不存在则创建，需要第三个参数 mode 设置文件权限
O_TRUNC	如果文件已存在,并且以只写或可读可写方式打开， 则将其长度截断（Truncate）为 0 字节。
O_EXCL	如果同时指定了 O_CREAT，并且文件已存在,则出错返回
O_SYNC	对文件的写操作将按照同步的要求完成

- **返回值** 成功返回文件描述符，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_close()

- **描述** 关闭文件
- **函数原型**

```
int ms_io_close(int fd);
```

- **参数**

输入/输出	参数	描述
[in]	fd	文件描述符

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_fcntl()

- **描述** 控制文件描述符
- **函数原型**

```
int ms_io_fcntl(int fd, int cmd, int arg);
```

- **参数**

输入/输出	参数	描述
[in]	fd	文件描述符
[in]	cmd	控制命令
[in]	arg	控制命令的参数

- **返回值** 成功时根据参数 cmd 的不同而返回不同的值，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_fstat()

- **描述** 获得文件的状态
- **函数原型**

```
int ms_io_fstat(int fd, ms_stat_t *buf);
```

- **参数**

输入/输出	参数	描述
[in]	fd	文件描述符
[out]	buf	文件状态

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无

- **示例** 无

ms_io_isatty()

- **描述** 判断文件是否为一个 TTY 设备
- **函数原型**

```
int ms_io_isatty(int fd);
```

- **参数**

输入/输出	参数	描述
[in]	fd	文件描述符

- **返回值** 1：是 TTY 设备，0：不是 TTY 设备
- **注意事项** 无
- **示例** 无

ms_io_fsync()

- **描述** 回写文件 CACHE 到磁盘
- **函数原型**

```
int ms_io_fsync(int fd);
```

- **参数**

输入/输出	参数	描述
[in]	fd	文件描述符

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_fdatasync()

- **描述** 回写文件数据 CACHE 到磁盘
- **函数原型**

```
int ms_io_fdatasync(int fd);
```

- **参数**

输入/输出	参数	描述
[in]	fd	文件描述符

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_ftruncate()

- **描述** 截断文件（修改文件长度）
- **函数原型**

```
int ms_io_ftruncate(int fd, ms_off_t len);
```

- **参数**

输入/输出	参数	描述
[in]	fd	文件描述符
[in]	len	需要截断为的文件长度

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_ioctl()

- **描述** IO 控制
- **函数原型**

```
int ms_io_ioctl(int fd, int cmd, ms_ptr_t arg);
```

- **参数**

输入/输出	参数	描述
[in]	fd	文件描述符
[in]	cmd	控制的命令
[in]	arg	控制的命令参数

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无

- **示例** 无

ms_io_dup()

- **描述** 复制文件描述符
- **函数原型**

```
int ms_io_dup(int fd);
```

- **参数**

输入/输出	参数	描述
[in]	fd	文件描述符

- **返回值** 成功返回新的文件描述符，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_dup2()

- **描述** 复制文件描述符到指定的文件描述符
- **函数原型**

```
int ms_io_dup2(int fd, int to);
```

- **参数**

输入/输出	参数	描述
[in]	fd	文件描述符
[in]	to	需要复制到的文件描述符

- **返回值** 成功返回 to 描述符，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_lseek()

- **描述** 调整文件读写指针
- **函数原型**

```
ms_off_t ms_io_lseek(int fd, ms_off_t offset, int whence);
```


- **参数**

输入/输出	参数	描述
[in]	fd	文件描述符
[in]	offset	偏移量
[in]	whence	定位基准

- **返回值** 成功返回新的文件读写指针，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_read()

- **描述** 读文件
- **函数原型**

```
ms_ssize_t ms_io_read(int fd, ms_ptr_t buf, ms_size_t len);
```

- **参数**

输入/输出	参数	描述
[in]	fd	文件描述符
[out]	buf	数据缓冲区
[in]	len	需要读取的长度

- **返回值** 返回成功读取到的字节数，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_write()

- **描述** 写文件
- **函数原型**

```
ms_ssize_t ms_io_write(int fd, ms_const_ptr_t buf, ms_size_t len);
```

- **参数**

输入/输出	参数	描述
[in]	fd	文件描述符

输入/输出	参数	描述
[in]	buf	需要写入的数据
[in]	len	需要写入的长度

- **返回值** 返回成功写入的字节数，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_tell()

- **描述** 获得文件当前读写指针
- **内联函数实现** 等价于如下的 `ms_io_lseek` 调用

```
static MS_FORCE_INLINE ms_off_t ms_io_tell(int fd)
{
    return ms_io_lseek(fd, 0, SEEK_CUR);
}
```

- **参数**

输入/输出	参数	描述
[in]	fd	文件描述符

- **返回值** 成功返回文件当前读写指针，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_poll()

- **描述** poll 等待文件集事件
- **函数原型**

```
int ms_io_poll(ms_pollfd_t *fds, ms_nfds_t nfds, int timeout);
```

- **参数**

输入/输出	参数	描述
[in]	fds	poll 文件描述符数组
[in]	nfds	fds 数组的元素个数
[in]	timeout	超时值

- **返回值** 成功返回等到的描述符数量，超时返回 0，失败返回 -1，并设置 errno

- **注意事项** 无
- **示例** 无

ms_io_select()

- **描述** select 等待文件集事件
- **函数原型**

```
int ms_io_select(int maxfd, ms_fd_set_t *readfds, ms_fd_set_t *writefds,
                 ms_fd_set_t *errorfds, ms_timeval_t *timeout);
```

- **参数**

输入/输出	参数	描述
[in]	maxfd	文件描述符列表中最大文件描述符加 1
[in]	readfds	读文件描述符集
[in]	writefds	写文件描述符集
[in]	errorfds	异常文件描述符集
[in]	timeout	等到超时时间

- **返回值** 成功返回等到的文件描述符数量，超时返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_link()

- **描述** 链接文件
- **函数原型**

```
int ms_io_link(const char *path1, const char *path2);
```

- **参数**

输入/输出	参数	描述
[in]	path1	文件路径
[in]	path2	需要创建的链接文件

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_rename()

- **描述** 文件重命名（移动文件）
- **函数原型**

```
int ms_io_rename(const char *old, const char *_new);
```

- **参数**

输入/输出	参数	描述
[in]	old	旧的文件路径
[in]	_new	新的文件路径

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_stat()

- **描述** 获得文件状态
- **函数原型**

```
int ms_io_stat(const char *path, ms_stat_t *buf);
```

- **参数**

输入/输出	参数	描述
[in]	path	文件路径
[out]	buf	文件状态

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_lstat()

- **描述** 获得文件状态，如果文件是符号链接，则获得符号链接本身的状态
- **函数原型**

```
int ms_io_lstat(const char *path, ms_stat_t *buf);
```

- **参数**

输入/输出	参数	描述
[in]	path	文件路径
[out]	buf	文件状态

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_statvfs()

- **描述** 获得文件系统状态
- **函数原型**

```
int ms_io_statvfs(const char *path, ms_statvfs_t *buf);
```

- **参数**

输入/输出	参数	描述
[in]	path	挂载点的路径
[out]	buf	文件系统状态

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_unlink()

- **描述** 删除文件
- **函数原型**

```
int ms_io_unlink(const char *path);
```

- **参数**

输入/输出	参数	描述
[in]	path	文件路径

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无

- **示例** 无

ms_io_mkdir()

- **描述** 创建目录
- **函数原型**

```
int ms_io_mkdir(const char *path, ms_mode_t mode);
```

- **参数**

输入/输出	参数	描述
[in]	path	需要创建的目录路径
[in]	mode	目录模式

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_rmdir()

- **描述** 删除目录
- **函数原型**

```
int ms_io_rmdir(const char *path);
```

- **参数**

输入/输出	参数	描述
[in]	path	需要删除的目录路径

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_access()

- **描述** 判断文件是否可以以指定的访问模式来访问
- **函数原型**

```
int ms_io_access(const char *path, int amode);
```


- **参数**

输入/输出	参数	描述
[in]	path	文件路径
[in]	amode	访问模式

- **返回值** 可以访问返回 0，不能访问返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_truncate()

- **描述** 截断文件（修改文件长度）
- **函数原型**

```
int ms_io_truncate(const char *path, ms_off_t len);
```

- **参数**

输入/输出	参数	描述
[in]	path	文件路径
[in]	len	需要截断为的文件长度

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_sync()

- **描述** 回写文件系统 CACHE 到磁盘
- **函数原型**


```
int ms_io_sync(void);
```

- **参数** 无
- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_mkfs()

- **描述** 格式化磁盘
- **函数原型**

```
ms_err_t ms_io_mkfs(const char *path, ms_const_ptr_t param);
```

- **参数**

输入/输出	参数	描述
[in]	path	挂载点路径
[in]	param	格式化参数

- **返回值** MS-RTOS 错误码
- **注意事项** 仅内核态可用
- **示例** 无

ms_io_mount()

- **描述** 挂载磁盘
- **函数原型**

```
ms_err_t ms_io_mount(const char *mnt_path, const char *dev_path, const char *fs_name, ms_const
_ptr_t param);
```

- **参数**

输入/输出	参数	描述
[in]	mnt_path	挂载点路径
[in]	dev_path	设备路径
[in]	fs_name	文件系统名称
[in]	param	挂载参数

- **返回值** MS-RTOS 错误码
- **注意事项** 仅内核态可用
- **示例** 无

ms_io_mount_ex()

- **描述** 挂载磁盘
- **函数原型**


```
ms_err_t ms_io_mount_ex(const char *mnt_path, const char *dev_path, const char *fs_name, ms_const_ptr_t param,
                        ms_callback_t on_umount, ms_ptr_t on_umount_arg);
```

- **参数**

输入/输出	参数	描述
[in]	mnt_path	挂载点路径
[in]	dev_path	设备路径
[in]	fs_name	文件系统名称
[in]	param	挂载参数
[in]	on_umount	卸载挂载点时的回调函数
[in]	on_umount_arg	卸载挂载点时的回调函数参数

- **返回值** MS-RTOS 错误码
- **注意事项** 仅内核态可用
- **示例** 无

ms_io_unmount()

- **描述** 卸载挂载点
- **函数原型**

```
ms_err_t ms_io_unmount(const char *mnt_path, ms_const_ptr_t param);
```

- **参数**

输入/输出	参数	描述
[in]	mnt_path	挂载点路径
[in]	param	卸载参数

- **返回值** MS-RTOS 错误码
- **注意事项** 仅内核态可用
- **示例** 无

ms_io_opendir()

- **描述** 打开目录
- **函数原型**

```
MS_DIR *ms_io_opendir(const char *path);
```

- **参数**

输入/输出	参数	描述
[in]	path	需要打开的目录路径

- **返回值** 成功返回目录流指针，失败返回 MS_NULL
- **注意事项** 无
- **示例** 无

ms_io_closedir()

- **描述** 关闭目录流
- **函数原型**

```
int ms_io_closedir(MS_DIR *dir);
```

- **参数**

输入/输出	参数	描述
[in]	dir	目录流指针

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_readdir_r()

- **描述** 读目录项
- **函数原型**


```
int ms_io_readdir_r(MS_DIR *dir, ms_dirent_t *entry, ms_dirent_t **result);
```

- **参数**

输入/输出	参数	描述
[in]	dir	目录流指针
[out]	entry	目录项信息

输入/输出	参数	描述
[out]	result	指向 entry 地址或 MS_NULL

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** MS-RTOS 内核没有提供的不可重入函数 `ms_io_readdir`
- **示例** 无

ms_io_rewinddir()

- **描述** 重置目录流读指针
- **函数原型**

```
int ms_io_rewinddir(MS_DIR *dir);
```

- **参数**

输入/输出	参数	描述
[in]	dir	目录流指针

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_seekdir()

- **描述** 调整目录流读指针
- **函数原型**

```
int ms_io_seekdir(MS_DIR *dir, long loc);
```

- **参数**

输入/输出	参数	描述
[in]	dir	目录流指针
[in]	loc	需要调整到的位置

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_io_telldir()

- **描述** 获得目录流读指针
- **函数原型**

```
long ms_io_telldir(MS_DIR *dir);
```

- **参数**

输入/输出	参数	描述
[in]	dir	目录流指针

- **返回值** 目录流的当前读指针
- **注意事项** 无
- **示例** 无

posix 兼容 IO 相关 API

MS-RTOS 的 IO 子系统还提供了 posix 兼容的文件、目录 API（当然 MS-RTOS 原生的文件、目录 API 效率更高，推荐使用 MS-RTOS 原生的文件、目录 API）：

```
// 当前工作目录
int chdir(const char *path);
char *getcwd(char *buf, size_t size);

// 文件
int creat(const char *path, mode_t mode);
int open(const char *path, int oflag, ... /* mode */);
int close(int fd);
int fcntl(int fd, int cmd, ... /* arg */);
int fstat(int fd, struct stat *buf);
int isatty(int fd);
int fsync(int fd);
int fdatasync(int fd);
int ftruncate(int fd, off_t len);
int ioctl(int fd, int cmd, void *arg);
int dup(int fd);
int dup2(int fd, int to);
off_t lseek(int fd, off_t offset, int whence);
ssize_t read(int fd, void *buf, size_t len);
ssize_t write(int fd, const void *buf, size_t len);

// 多路 IO 复用
int poll(pollfd_t *fds, nfds_t nfds, int timeout);
int select(int maxfd, fd_set *readfds, fd_set *writefds,
           fd_set *errorfds, struct timeval *timeout);

// 文件系统
```



```
int link(const char *path1, const char *path2);
int rename(const char *old, const char *_new);
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
int unlink(const char *path);
int mkdir(const char *path, mode_t mode);
int rmdir(const char *path);
int access(const char *path, int amode);
int truncate(const char *path, off_t len);
void sync(void);

// 目录流
DIR *opendir(const char *path);
int closedir(DIR *dir);
int readdir_r(DIR *dir, struct dirent *entry, struct dirent **result);
struct dirent *readdir(DIR *dir); // 不可重入，不推荐使用
int rewinddir(DIR *dir);
int seekdir(DIR *dir, long loc);
long telldir(DIR *dir);
```


20 MS-RTOS 网络

> 本章将介绍 MS-RTOS 网络相关接口的使用。

网络编程接口

MS-RTOS 网络子系统屏蔽了不同网络实现（如 TCP/IP 协议栈和 ESP8266 等有线、无线通信模块）的差异，向上提供了统一的标准的 BSD/socket 套接字的网络编程 API，有效保证了上层物联网应用的跨平台能力，同时不同网络实现能够在同一套系统中共存。

网络相关 API

下表展示了网络操作 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_net_set_impl	●	●
ms_net_get_dns_server	●	●
ms_net_set_dns_server	●	●
gethostname	●	●
sethostname	●	●
htonl	●	●
htons	●	●
ntohl	●	●
ntohs	●	●
inet_addr	●	●
inet_aton	●	●
inet_ntoa	●	●
inet_ntoa_r	●	●
inet_ntop	●	●
inet_pton	●	●
gethostbyname	●	●
gethostbyname_r	●	●
getaddrinfo	●	●

API	用户空间	内核空间
freeaddrinfo	●	●
gai_strerror	●	●
if_indextoname	●	●
if_nametoindex	●	●
socket	●	●
accept	●	●
bind	●	●
shutdown	●	●
getpeername	●	●
getsockname	●	●
getsockopt	●	●
setsockopt	●	●
connect	●	●
listen	●	●
recv	●	●
recvfrom	●	●
recvmsg	●	●
send	●	●
sendto	●	●
sendmsg	●	●

socket 除了可以使用以上特有的接口操作外，还可以使用 read、write、ioctl、 fcntl、close、select、poll 等 posix 标准和 MS-RTOS 原生的文件接口操作。

ms_net_set_impl()

- **描述** 设置当前进程的网络实现，默认的网络实现为系统第一个注册的网络实现，系统已经注册的网络实现可通过 nets 命令查看
- **函数原型**

```
ms_err_t ms_net_set_impl(const char *name);
```

- **参数**

输入/输出	参数	描述
[in]	name	网络实现名称

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_net_get_dns_server()

- **描述** 获得指定的 DNS 服务器地址
- **函数原型**

```
int ms_net_get_dns_server(ms_uint8_t numdns, ip_addr_t *dnsserver);
```

- **参数**

输入/输出	参数	描述
[in]	numdns	DNS 服务器编号
[out]	dnsserver	DNS 服务器地址

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

ms_net_set_dns_server()

- **描述** 设置指定的 DNS 服务器地址
- **函数原型**

```
int ms_net_set_dns_server(ms_uint8_t numdns, const ip_addr_t *dnsserver);
```

- **参数**

输入/输出	参数	描述
[in]	numdns	DNS 服务器编号
[in]	dnsserver	DNS 服务器地址

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

gethostname()

- **描述** 获取主机名称
- **函数原型**

```
int gethostname(char *name, size_t len);
```

- **参数**

输入/输出	参数	描述
[out]	name	主机名称缓存区
[in]	len	主机名称缓存区的大小

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

sethostname()

- **描述** 设置主机名称
- **函数原型**

```
int sethostname(const char *name, size_t len);
```

- **参数**

输入/输出	参数	描述
[in]	name	主机名称
[in]	len	主机名称的最大长度

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

htonl()

- **描述** 长整数主机字节序转网络字节序
- **函数原型**

```
ms_uint32_t htonl(ms_uint32_t x);
```


- **参数**

输入/输出	参数	描述
[in]	x	需要转换的长整数

- **返回值** 转换后的长整数
- **注意事项** 无
- **示例** 无

htons()

- **描述** short 类型主机字节序转网络字节序
- **函数原型**

```
ms_uint16_t htons(ms_uint16_t x);
```

- **参数**

输入/输出	参数	描述
[in]	x	需要转换的 short 类型数

- **返回值** 转换后的 short 类型数
- **注意事项** 无
- **示例** 无

ntohl()

- **描述** 长整数网络字节序转主机字节序
- **函数原型**

```
ms_uint32_t ntohl(ms_uint32_t x);
```

- **参数**

输入/输出	参数	描述
[in]	x	需要转换的长整数

- **返回值** 转换后的长整数
- **注意事项** 无
- **示例** 无

ntohs()

- **描述** short 类型网络字节序转主机字节序
- **函数原型**

```
ms_uint16_t ntohs(ms_uint16_t x);
```

- **参数**

输入/输出	参数	描述
[in]	x	需要转换的 short 类型数

- **返回值** 转换后的 short 类型数
- **注意事项** 无
- **示例** 无

inet_addr()

- **描述** 点分十进制字符串转换为 32 位网络字节序二进制值
- **函数原型**

```
ms_uint32_t inet_addr(const char *cp);
```

- **参数**

输入/输出	参数	描述
[in]	cp	点分十进制地址，如 192.168.1.15

- **返回值** 成功时返回 32 位二进制的网络字节序地址，失败返回 INADDR_NONE
- **注意事项** 此函数存在一个问题，不能表示全部有效的 IP 地址（从 0.0.0.0 到 255.255.255.255）
- **示例** 无

inet_aton()

- **描述** 点分十进制字符串转换为 32 位网络字节序二进制值
- **函数原型**

```
int inet_aton(const char *cp, in_addr_t *addr);
```

- **参数**

输入/输出	参数	描述
[in]	cp	点分十进制地址，如 192.168.1.15
[out]	addr	用于保存网络字节序二进制值的缓存地址

- **返回值** 字符串有效时返回 1， 无效时返回 0
- **注意事项** 无
- **示例** 无

inet_ntoa()

- **描述** 32 位网络字节序二进制值转换为点分十进制字符串
- **函数原型**

```
char *inet_ntoa(const in_addr_t *addr);
```

- **参数**

输入/输出	参数	描述
[in]	addr	32 位网络字节序地址

- **返回值** 正确时返回字符串指针，错误时返回 MS_NULL
- **注意事项** 此函数是不可重入的，建议使用可重入版本 `inet_ntoa_r`
- **示例** 无

inet_ntoa_r()

- **描述** 32 位网络字节序二进制值转换为点分十进制字符串
- **函数原型**

```
char *inet_ntoa_r(const in_addr_t *addr, char *buf, int buflen);
```

- **参数**

输入/输出	参数	描述
[in]	addr	32 位网络字节序地址
[in]	buf	点分十进制字符串缓冲区
[in]	buflen	缓冲区长度

- **返回值** 正确时返回字符串指针，错误时返回 MS_NULL
- **注意事项** 无

- **示例** 无

inet_ntop()

- **描述** 将二进制数值转换为字符串表达式
- **函数原型**

```
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
```

- **参数**

输入/输出	参数	描述
[in]	af	必须是 AF_INET 或者 AF_INET6。其他地址簇不被支持，且返回错误
[in]	src	二进制数值
[in]	dst	用于保存转换后的地址字符串
[in]	size	dst 缓冲区的大小

- **返回值** 正确时返回地址串的指针，错误时返回 MS_NULL
- **注意事项** 无
- **示例** 无

inet_pton()

- **描述** 将字符串表达式转换为二进制数值
- **函数原型**

```
int  inet_pton(int af, const char *src, void *dst);
```

- **参数**

输入/输出	参数	描述
[in]	af	必须是 AF_INET 或者 AF_INET6。其他地址簇不被支持，且返回错误
[in]	src	地址字符串，如 IPv4 的 192.168.1.15
[in]	dst	用于保存二进制数值结果

- **返回值** 正确时返回 1， 错误时返回 -1，输入不是有效的表达格式时返回 0
- **注意事项** 无
- **示例** 无

gethostbyname()

- **描述** 用域名或者主机名获取地址
- **函数原型**

```
struct hostent *gethostbyname(const char *name);
```

- **参数**

输入/输出	参数	描述
[in]	name	域名或者主机名

- **返回值** 成功返回获取的网络地址，失败时返回 MS_NULL，并设置 errno
- **注意事项** 该函数是不可重入的，建议使用可重入版本 `gethostbyname_r`
- **示例** 无

gethostbyname_r()

- **描述** 用域名或者主机名获取地址
- **函数原型**

```
int  gethostbyname_r(const char *name, struct hostent *ret, char *buf,
                     size_t buflen, struct hostent **result, int *h_errnop);
```

- **参数**

输入/输出	参数	描述
[in]	name	域名或者主机名
[in]	ret	返回的网络地址
[out]	buf	结果缓冲区
[in]	buflen	结果缓冲区的长度
[out]	result	如果成功，指向 ret，如果失败为 MS_NULL
[out]	h_errnop	存储错误码

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

getaddrinfo()

- **描述** 获取地址信息
- **函数原型**

```
int  getaddrinfo(const char *nodename, const char *servname,
                 const struct addrinfo *hints, struct addrinfo **res);
```

- **参数**

输入/输出	参数	描述
[in]	nodename	地址字符串
[in]	servname	服务名
[in]	hints	输入地址信息
[out]	res	结果地址信息

- **返回值** 成功返回 0，失败返回非 0 值
- **注意事项** `getaddrinfo` 函数获取到的地址结构，需要使用 `freeaddrinfo` 函数进行释放
- **示例** 无

freeaddrinfo()

- **描述** 释放地址信息结构
- **函数原型**

```
void freeaddrinfo(struct addrinfo *ai);
```

- **参数**

输入/输出	参数	描述
[in]	ai	由 getaddrinfo 函数返回的地址信息结构

- **返回值** 无
- **注意事项** 无
- **示例** 无

gai_strerror()

- **描述** 获取网络错误码对应字符串
- **函数原型**

```
const char *gai_strerror(int error);
```

- **参数**

输入/输出	参数	描述
[in]	error	错误码

- **返回值** 错误类型字符串
- **注意事项** 无
- **示例** 无

if_indextoname()

- **描述** 根据网络接口索引获得网络接口名
- **函数原型**

```
char *if_indextoname(unsigned int ifindex, char *ifname);
```

- **参数**

输入/输出	参数	描述
[in]	ifindex	网络接口索引
[out]	ifname	网络接口名

- **返回值** 成功返回网络接口名，失败返回 MS_NULL
- **注意事项** 无
- **示例** 无

if_nametoindex()

- **描述** 根据网络接口名获得网络接口索引
- **函数原型**

```
unsigned int if_nametoindex(const char *ifname);
```

- **参数**

输入/输出	参数	描述
[in]	ifname	网络接口名

- **返回值** 网络接口索引
- **注意事项** 无
- **示例** 无

socket()

- **描述** 创建一个套接字
- **函数原型**

```
int socket(int domain, int type, int protocol);
```

- **参数**

输入/输出	参数	描述
[in]	domain	协议域
[in]	type	套接字类型
[in]	protocol	协议类型

- **返回值** 成功时返回套接字描述符，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

accept()

- **描述** 仅由 TCP 服务器调用，用于返回一个已完成的连接
- **函数原型**

```
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

- **参数**

输入/输出	参数	描述
[in]	s	套接字，socket 函数返回
[out]	addr	返回已连接对端的协议地址结构信息
[out]	addrlen	返回已连接协议地址结构大小

- **返回值** 成功返回非负已连接套接字描述符，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

bind()

- **描述** 把一个本地协议地址赋予一个套接字，协议地址的含义只取决于协议本身
- **函数原型**

```
int bind(int s, const struct sockaddr *name, socklen_t namelen);
```

- **参数**

输入/输出	参数	描述
[in]	s	套接字，socket 函数返回
[in]	name	一个指向特定协议域的 sockaddr 结构体类型指针
[in]	namelen	name 结构的长度

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

shutdown()

- **描述** 关闭一个套接字
- **函数原型**

```
int shutdown(int s, int how);
```

- **参数**

输入/输出	参数	描述
[in]	s	需要关闭的套接字
[in]	how	关闭的方式

how 的取值类型

参数	含义
SHUT_RD	关闭 socket 上的读功能，后续将不允许 socket 进行读操作。
SHUT_WR	关闭 socket 的写功能，后续将不允许 socket 进行写操作。
SHUT_RDWR	关闭 socket 的读写功能。

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

getpeername()

- **描述** 用于返回与某个套接字关联的远端协议地址
- **函数原型**

```
int getpeername(int s, struct sockaddr *name, socklen_t *namelen);
```

- **参数**

输入/输出	参数	描述
[in]	s	套接字，socket 接口返回
[out]	name	返回已连接对端的协议地址结构信息
[out]	namelen	用于返回已连接协议地址结构大小

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

getsockname()

- **描述** 用于返回与某个套接字关联的本地协议地址
- **函数原型**

```
int getsockname(int s, struct sockaddr *name, socklen_t *namelen);
```

- **参数**

输入/输出	参数	描述
[in]	s	套接字，socket 接口返回
[out]	name	返回本地的协议地址结构信息
[out]	namelen	用于返回本地地址结构大小

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

getsockopt()

- **描述** 返回套接字选项
- **函数原型**


```
int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);
```

- **参数**

输入/输出	参数	描述
[in]	s	套接字，socket 接口返回
[in]	level	选项等级
[in]	optname	选项名
[out]	optval	选项值
[in]	optlen	选项长度

套接字选项如下：

选项等级	选项名	说明	数据类型
SOL_SOCKET	SO_BROADCAST	运行发送广播数据报	int
SOL_SOCKET	SO_ERROR	获取待处理错误并消除	int
SOL_SOCKET	SO_KEEPALIVE	周期性测试连接是否存活	int
SOL_SOCKET	SO_LINGER	若有数据待发送则延迟关闭	struct linger
SOL_SOCKET	SO_DONTLINGER	关闭 SO_LINGER 选项	int
SOL_SOCKET	SO_RCVBUF	接收缓冲区大小	int
SOL_SOCKET	SO_RCVTIMEO	接收超时	struct timeval
SOL_SOCKET	SO_SNDTIMEO	发送超时	struct timeval
SOL_SOCKET	SO_REUSEADDR	允许重用本地地址	int
SOL_SOCKET	SO_REUSEPORT	允许重用本地端口	int
SOL_SOCKET	SO_TYPE	取得套接字类型	int
SOL_SOCKET	SO_CONTIMEO	连接超时	struct timeval
IPPROTO_IP	IP_TOS	服务类型和优先级	int
IPPROTO_IP	IP_TTL	存活时间	int
IPPROTO_IP	IP_MULTICAST_IF	指定外出接口	struct in_addr

选项等级	选项名	说明	数据类型
IPPROTO_IP	IP_MULTICAST_TTL	指定外出 TTL	unsigned char
IPPROTO_IP	IP_MULTICAST_LOOP	指定是否回馈	unsigned char
IPPROTO_IP	IP_ADD_MEMBERSHIP	加入多播组	struct in_mreq
IPPROTO_IP	IP_DROP_MEMBERSHIP	离开多播组	struct in_mreq
IPPROTO_TCP	TCP_KEEPALIVE	控测对方是否存活前连接闲置秒数	int
IPPROTO_TCP	TCP_KEEPIDLE	对一个连接探测前的允许时间	int
IPPROTO_TCP	TCP_KEEPINTVL	两个探测的时间间隔	int
IPPROTO_TCP	TCP_KEEPCNT	探测的最大次数	int
IPPROTO_IPV6	IPV6_V6ONLY	只允许 IPV6（SylixOS 不支持数据报通信）	int
IPPROTO_UDPLITE	UDPLITE_SEND_CSCOV	执行发送校验和	int
IPPROTO_UDPLITE	UDPLITE_RECV_CSCOV	执行接收校验和	int
IPPROTO_RAW	IPV6_CHECKSUM	IPV6 校验和	int

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

setsockopt()

- **描述** 设置套接字选项
- **函数原型**

```
int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);
```

- **参数**

输入/输出	参数	描述
[in]	s	套接字，socket 接口返回
[in]	level	选项等级

输入/输出	参数	描述
[in]	optname	选项名
[in]	optval	选项值
[in]	optlen	选项长度

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

connect()

- **描述** TCP 客户端建立与 TCP 服务器的连接
- **函数原型**

```
int connect(int s, const struct sockaddr *name, socklen_t namelen);
```

- **参数**

输入/输出	参数	描述
[in]	s	套接字，由 socket 接口返回
[in]	name	一个指向特定协议域的 sockaddr 结构体类型的指针
[in]	namelen	name 结构的长度

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

listen()

- **描述** 由 TCP 服务器调用，表示可以接受指向该套接字的连接请求
- **函数原型**

```
int listen(int s, int backlog);
```

- **参数**

输入/输出	参数	描述
[in]	s	套接字，由 socket 接口返回

输入/输出	参数	描述
[in]	backlog	表示对应套接字可以接受的最大连接数

- **返回值** 成功返回 0，失败返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

recv()

- **描述** 接收报文数据
- **函数原型**

```
ssize_t recv(int s, void *mem, size_t len, int flags);
```

- **参数**

输入/输出	参数	描述
[in]	s	套接字，由 socket 接口返回
[out]	mem	存储接收的数据
[in]	len	表示读取数据的字节长度
[in]	flags	指定消息类型

- **返回值** 成功时返回读到的数据字节数，失败时返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

recvfrom()

- **描述** 接收报文数据
- **函数原型**

```
ssize_t recvfrom(int s, void *mem, size_t len, int flags,
                 struct sockaddr *from, socklen_t *fromlen);
```

- **参数**

输入/输出	参数	描述
[in]	s	套接字，由 socket 接口返回
[out]	mem	存储接收的数据

输入/输出	参数	描述
[in]	len	表示读取数据的字节长度
[in]	flags	指定消息类型
[in]	from	用于表示 UDP 数据报发送者的地址协议
[in]	fromlen	指定 from 地址大小的指针

- **返回值** 成功时返回读到的数据字节数，失败时返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

recvmsg()

- **描述** 接收报文数据
- **函数原型**

```
ssize_t recvmsg(int s, struct msghdr *message, int flags);
```

- **参数**

输入/输出	参数	描述
[in]	s	套接字，由 socket 接口返回
[out]	message	接收到的数据结构
[in]	flags	指定消息类型

```
struct msghdr {
    void          *msg_name;       // protocol address
    socklen_t      msg_namelen;    // size of protocol address
    struct iovec  *msg_iov;        // scatter/gather array
    int            msg_iovlen;     // elements in msg_iov
    void          *msg_control;    // ancillary data (cmsghdr struct)
    socklen_t      msg_controllen; // length of ancillary data
    int            msg_flags;      // flags returned by recvmsg()
};
```

- **返回值** 成功时返回读到的数据字节数，失败时返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

send()

- **描述** 发送报文数据

- **函数原型**

```
ssize_t send(int s, const void *dataptr, size_t size, int flags);
```

- **参数**

输入/输出	参数	描述
[in]	s	套接字，由 socket 接口返回
[in]	dataptr	存储发送的数据
[in]	len	表示读取数据的字节长度
[in]	flags	指定消息类型

- **返回值** 成功时返回发送的数据字节数，失败时返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

sendto()

- **描述** 发送报文数据
- **函数原型**

```
ssize_t sendto(int s, const void *dataptr, size_t size, int flags,
               const struct sockaddr *to, socklen_t tolen);
```

- **参数**

输入/输出	参数	描述
[in]	s	套接字，由 socket 接口返回
[in]	dataptr	存储发送的数据
[in]	size	表示发送数据的字节长度
[in]	flags	指定消息类型
[in]	to	用于表示 UDP 数据报发送者的地址协议
[in]	tolen	指定 to 地址大小的指针

- **返回值** 成功时返回发送的数据字节数，失败时返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

sendmsg()

- **描述** 发送报文数据
- **函数原型**

```
ssize_t sendmsg(int s, const struct msghdr *message, int flags);
```

- **参数**

输入/输出	参数	描述
[in]	s	套接字，由 socket 接口返回
[in]	message	发送的数据结构
[in]	flags	指定消息类型

- **返回值** 成功时返回发送的数据字节数，失败时返回 -1，并设置 errno
- **注意事项** 无
- **示例** 无

TCP 回显服务器示例

以下的示例建立了一个 TCP 回显服务器，它监听 8101 端口，PC 机可以使用网络工具，建立一个 TCP 客户端，连接它的 8101 端口，然后发送数据，将收到发送的原样数据：

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>

#define __TCP_ECHO_PORT_SERVER          8101                            /* 服务器端口号
        */
#define __TCP_ECHO_BUFF_SIZE_SERVER     257                             /* 服务器接收缓冲区大小
       */

static char             cRecvBuff[__TCP_ECHO_BUFF_SIZE_SERVER] = {0};   /* 接收缓冲区
        */

int main (int argc, char *argv[])
{
    int                 iRet        = -1;
    int                 sockFd      = -1;
    int                 sockFdNew   = -1;
    socklen_t           uiAddrLen    = sizeof(struct sockaddr_in);      /* 地址结构大小
         */
    register ssize_t    sstRecv     = 0;                               /* 接收到的数据长度
        */
    struct sockaddr_in  sockaddrinLocal;                                /* 本地地址
         */
    struct sockaddr_in  sockaddrinRemote;                               /* 远端地址
```

```
     */

    fprintf(stdout, "TCP echo server start.\r\n");

    sockFd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sockFd < 0) {
        fprintf(stderr, "TCP echo server socket error.\r\n");
        return  (-1);
    }

    /*
     * 初始化本地地址结构
     */
    memset(&sockaddrinLocal, 0, sizeof(sockaddrinLocal));
    sockaddrinLocal.sin_len         = sizeof(struct sockaddr_in);
    sockaddrinLocal.sin_family      = AF_INET;                          /* 地址族
         */
    sockaddrinLocal.sin_addr.s_addr = INADDR_ANY;
    sockaddrinLocal.sin_port        = htons(__TCP_ECHO_PORT_SERVER);    /* 绑定服务器端口
        */
    iRet = bind(sockFd, (struct sockaddr *)&sockaddrinLocal, sizeof(sockaddrinLocal));
                                                                        /* 绑定本地地址与端口
        */
    if (iRet < 0) {                                                     /* 绑定操作失败
         */
        close(sockFd);                                                  /* 关闭已经创建的 socke
t        */
        fprintf(stderr, "TCP echo server bind error.\r\n");
        return  (-1);                                                   /* 错误返回
         */
    }

    listen(sockFd, 2);

    sockFdNew = accept(sockFd, (struct sockaddr *)&sockaddrinRemote, &uiAddrLen);
    if (sockFdNew < 0) {
        close(sockFd);                                                  /* 关闭已经创建的 socke
t        */
        fprintf(stderr, "TCP echo server accept error.\r\n");
        return  (-1);                                                   /* 错误返回
         */
    }

    for (;;) {
        memset(&cRecvBuff[0], 0, __TCP_ECHO_BUFF_SIZE_SERVER);          /* 清空接收缓冲区
        */

        sstRecv = read(sockFdNew, (void *)&cRecvBuff[0], __TCP_ECHO_BUFF_SIZE_SERVER);
                                                                        /* 从远端接收数据
        */
        if (sstRecv <= 0) {                                             /* 接收数据失败
         */
            if ((errno != ETIMEDOUT) && (errno != EWOULDBLOCK)) {       /* 非超时与非阻塞
        */
```



```
                close(sockFdNew);                                   /* 关闭已经连接的 socke
t       */
                fprintf(stderr, "TCP echo server recvfrom error.\r\n");
                return  (-1);                                       /* 错误返回
        */
            }

            continue;                                               /* 超时或非阻塞后重新运
行      */
        }

        cRecvBuff[sstRecv] = 0;
        printf("I recv %s\r\n", cRecvBuff);
                                                                    /* 将回射数据发回远端
        */
        write(sockFdNew, (const void *)&cRecvBuff[0], sstRecv);
    }

    return  (0);
}
```

UDP 多 IP 回显服务器示例

```
#include <ms_rtos.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>

#define __UDP_ECHO_PORT_SERVER          8101                        /* 服务器端口号
        */
#define __UDP_ECHO_BUFF_SIZE_SERVER     257                         /* 服务器接收缓冲区大小
        */
#define __UDP_ECHO_TIMEOUT_SERVER       5

static char             cRecvBuff[__UDP_ECHO_BUFF_SIZE_SERVER];     /* 接收缓冲区
        */

int main (int argc, char **argv)
{
    int                 iRet;
    int                 sockFd1;
    int                 sockFd2;
    int                 sockFd3;
    int                 maxFd;
    socklen_t           uiAddrLen;                                  /* 地址结构大小
        */
    register ssize_t    sstRecv;                                    /* 接收到的数据长度
        */
    struct sockaddr_in  sockaddrinLocal;                            /* 本地地址
        */
    struct sockaddr_in  sockaddrinRemote;                           /* 远端地址
        */
```



```
    fd_set              rfds;
    struct timeval      tv;
    char                ipAddrString[sizeof("255.255.255.255")];

    /*
     * 创建 socket 1
     */
    sockFd1 = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sockFd1 < 0) {
        fprintf(stderr, "UDP echo server socket error, line %d.\r\n", __LINE__);
        return  (-1);
    }

    /*
     * 设置网络接口 st0 的 mac 地址
     */
    {
        struct ifreq ifreq;

        bzero(&ifreq, sizeof(struct ifreq));

        strcpy(ifreq.ifr_name, "st0");

        ifreq.ifr_hwaddr.sa_data[0] = 0x11;
        ifreq.ifr_hwaddr.sa_data[1] = 0x22;
        ifreq.ifr_hwaddr.sa_data[2] = 0x33;
        ifreq.ifr_hwaddr.sa_data[3] = 0x44;
        ifreq.ifr_hwaddr.sa_data[4] = 0x55;
        ifreq.ifr_hwaddr.sa_data[5] = 0x66;
        ioctl(sockFd1, SIOCSIFHWADDR, &ifreq);

        /*
         * 获取网络接口 st0 的 mac 地址
         */
        ioctl(sockFd1, SIOCGIFHWADDR, &ifreq);
        ms_printf("MAC addr: %02x:%02x:%02x:%02x:%02x:%02x\n",
                  (ms_uint8_t)ifreq.ifr_hwaddr.sa_data[0],
                  (ms_uint8_t)ifreq.ifr_hwaddr.sa_data[1],
                  (ms_uint8_t)ifreq.ifr_hwaddr.sa_data[2],
                  (ms_uint8_t)ifreq.ifr_hwaddr.sa_data[3],
                  (ms_uint8_t)ifreq.ifr_hwaddr.sa_data[4],
                  (ms_uint8_t)ifreq.ifr_hwaddr.sa_data[5]);
    }

    /*
     * 设置网络接口 st0 的 ip 地址
     */
    {
        struct ifreq ifreq;
        struct sockaddr_in *psockaddrin;

        bzero(&ifreq, sizeof(struct ifreq));

        strcpy(ifreq.ifr_name, "st0");
```



```
    psockaddrin = (struct sockaddr_in *)&(ifreq.ifr_addr);
    psockaddrin->sin_len         = sizeof(struct sockaddr_in);
    psockaddrin->sin_family      = AF_INET;
    psockaddrin->sin_addr.s_addr = inet_addr("192.168.1.33");
    ioctl(sockFd1, SIOCSIFADDR, &ifreq);

    /*
     * 获取网络接口 st0 的 ip 地址
     */
    ioctl(sockFd1, SIOCGIFADDR, &ifreq);
    inet_ntoa_r(psockaddrin->sin_addr, ipAddrString, sizeof(ipAddrString));
    ms_printf("IP addr: %s\n", ipAddrString);
}

/*
 * 为 st0 增加一个 ip, 新增加的网络接口名为 mi0
 */
{
    struct ifaliasreq ifalias;
    struct sockaddr_in *psockaddrin;

    bzero(&ifalias, sizeof(struct ifaliasreq));

    strcpy(ifalias.ifra_name, "st0");

    /*
     * 设置 ip
     */
    psockaddrin = (struct sockaddr_in *)&(ifalias.ifra_addr);
    psockaddrin->sin_len         = sizeof(struct sockaddr_in);
    psockaddrin->sin_family      = AF_INET;
    psockaddrin->sin_addr.s_addr = inet_addr("192.168.1.100");

    /*
     * 设置子网掩码
     */
    psockaddrin = (struct sockaddr_in *)&(ifalias.ifra_mask);
    psockaddrin->sin_len         = sizeof(struct sockaddr_in);
    psockaddrin->sin_family      = AF_INET;
    psockaddrin->sin_addr.s_addr = inet_addr("255.255.255.0");

    ioctl(sockFd1, SIOCAIFADDR, &ifalias);
}

/*
 * socket 1 绑定服务器端口
 */
memset(&sockaddrinLocal, 0, sizeof(sockaddrinLocal));
sockaddrinLocal.sin_len         = sizeof(struct sockaddr_in);
sockaddrinLocal.sin_family      = AF_INET;                          /* 地址族
     */
sockaddrinLocal.sin_addr.s_addr = inet_addr("192.168.1.33");        /* 绑定到 192.168.1.33
     */
```



```
    sockaddrinLocal.sin_port        = htons(__UDP_ECHO_PORT_SERVER);    /* 绑定服务器端口
        */
    iRet = bind(sockFd1, (struct sockaddr *)&sockaddrinLocal, sizeof(sockaddrinLocal));
    if (iRet < 0) {                                                     /* 绑定操作失败
         */
        fprintf(stderr, "UDP echo server bind error, line %d.\r\n", __LINE__);
        return  (-1);                                                   /* 错误返回
         */
    }

    /*
     * 创建 socket 2
     */
    sockFd2 = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sockFd2 < 0) {
        fprintf(stderr, "UDP echo server socket error, line %d.\r\n", __LINE__);
        return  (-1);
    }

    /*
     * socket 2 绑定服务器端口
     */
    memset(&sockaddrinLocal, 0, sizeof(sockaddrinLocal));
    sockaddrinLocal.sin_len         = sizeof(struct sockaddr_in);
    sockaddrinLocal.sin_family      = AF_INET;                          /* 地址族
         */
    sockaddrinLocal.sin_addr.s_addr = inet_addr("192.168.1.100");       /* 绑定到 192.168.1.10
0         */
    sockaddrinLocal.sin_port        = htons(__UDP_ECHO_PORT_SERVER + 1);/* 绑定服务器端口
        */
    iRet = bind(sockFd2, (struct sockaddr *)&sockaddrinLocal, sizeof(sockaddrinLocal));
    if (iRet < 0) {                                                     /* 绑定操作失败
         */
        fprintf(stderr, "UDP echo server bind error, line %d.\r\n", __LINE__);
        return  (-1);                                                   /* 错误返回
         */
    }

    /*
     * 创建 socket 3
     */
    sockFd3 = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sockFd3 < 0) {
        fprintf(stderr, "UDP echo server socket error, line %d.\r\n", __LINE__);
        return  (-1);
    }

    /*
     * socket 3 绑定服务器端口
     */
    memset(&sockaddrinLocal, 0, sizeof(sockaddrinLocal));
    sockaddrinLocal.sin_len         = sizeof(struct sockaddr_in);
    sockaddrinLocal.sin_family      = AF_INET;                          /* 地址族
         */
```



```
    sockaddrinLocal.sin_addr.s_addr = INADDR_ANY;                    /* 绑定到任意 IP 地址
        */
    sockaddrinLocal.sin_port        = htons(__UDP_ECHO_PORT_SERVER + 2);/* 绑定服务器端口
        */
    iRet = bind(sockFd3, (struct sockaddr *)&sockaddrinLocal, sizeof(sockaddrinLocal));
    if (iRet < 0) {                                                  /* 绑定操作失败
         */
        fprintf(stderr, "UDP echo server bind error, line %d.\r\n", __LINE__);
        return  (-1);                                                /* 错误返回
         */
    }

    /*
     * 计算最大的文件描述符
     */
    maxFd = MS_MAX(sockFd1, sockFd2);
    maxFd = MS_MAX(maxFd, sockFd3);

    while (1) {

        /*
         * 将文件描述符加入到可读文件描述符集
         */
        FD_ZERO(&rfds);
        FD_SET(sockFd1, &rfds);
        FD_SET(sockFd2, &rfds);
        FD_SET(sockFd3, &rfds);

        /*
         * 等待至少一个 socket 可读，并设置等待的超时时间
         */
        tv.tv_sec  = __UDP_ECHO_TIMEOUT_SERVER;
        tv.tv_usec = 0;

        iRet = select(maxFd + 1, &rfds, MS_NULL, MS_NULL, &tv);
        if (iRet < 0) {
            /*
             * 出错
             */
            fprintf(stderr, "UDP echo server select error.\r\n");

        } else if (iRet == 0) {
            /*
             * 超时
             */
            fprintf(stderr, "UDP echo server select timeout.\r\n");

        } else {
            /*
             * 至少一个 socket 可读
             */
            if (FD_ISSET(sockFd1, &rfds)) {
                /*
                 * 读 socket 1
```



```
                 */
                uiAddrLen = sizeof(sockaddrinRemote);
                 sstRecv = recvfrom(sockFd1, cRecvBuff, sizeof(cRecvBuff), 0, (struct sockaddr
*)&sockaddrinRemote, &uiAddrLen);
                if (sstRecv > 0) {
                    /*
                     * 打印远端 IP 和端口及接收到的数据
                     */
                    cRecvBuff[sstRecv] = 0;
                       inet_ntoa_r(sockaddrinRemote.sin_addr, ipAddrString, sizeof(ipAddrStrin
g));
                     printf("I recv %s from %s:%u\r\n", cRecvBuff, ipAddrString, ntohs(sockaddr
inRemote.sin_port));

                    /*
                     * 回显
                     */
                     if (sendto(sockFd1, cRecvBuff, sstRecv, 0, (struct sockaddr *)&sockaddrinR
emote, sizeof(sockaddrinRemote)) != sstRecv) {
                         fprintf(stderr, "UDP echo server send error, line %d, errno %d.\r\n",
__LINE__, errno);
                    }
                } else {
                    fprintf(stderr, "UDP echo server read error, line %d.\r\n", __LINE__);
                }

            } else if (FD_ISSET(sockFd2, &rfds)) {
                /*
                 * 读 socket 2
                 */
                uiAddrLen = sizeof(sockaddrinRemote);
                 sstRecv = recvfrom(sockFd2, cRecvBuff, sizeof(cRecvBuff), 0, (struct sockaddr
*)&sockaddrinRemote, &uiAddrLen);
                if (sstRecv > 0) {
                    /*
                     * 打印远端 IP 和端口及接收到的数据
                     */
                    cRecvBuff[sstRecv] = 0;
                       inet_ntoa_r(sockaddrinRemote.sin_addr, ipAddrString, sizeof(ipAddrStrin
g));
                     printf("I recv %s from %s:%u\r\n", cRecvBuff, ipAddrString, ntohs(sockaddr
inRemote.sin_port));

                    /*
                     * 回显
                     */
                     if (sendto(sockFd2, cRecvBuff, sstRecv, 0, (struct sockaddr *)&sockaddrinR
emote, sizeof(sockaddrinRemote)) != sstRecv) {
                         fprintf(stderr, "UDP echo server send error, line %d, errno %d.\r\n",
__LINE__, errno);
                    }
                } else {
                    fprintf(stderr, "UDP echo server read error, line %d.\r\n", __LINE__);
                }
```



```
            } else if (FD_ISSET(sockFd3, &rfds)) {
                /*
                 * 读 socket 3
                 */
                uiAddrLen = sizeof(sockaddrinRemote);
                 sstRecv = recvfrom(sockFd3, cRecvBuff, sizeof(cRecvBuff), 0, (struct sockaddr
*)&sockaddrinRemote, &uiAddrLen);
                if (sstRecv > 0) {
                    /*
                     * 打印远端 IP 和端口及接收到的数据
                     */
                    cRecvBuff[sstRecv] = 0;
                       inet_ntoa_r(sockaddrinRemote.sin_addr, ipAddrString, sizeof(ipAddrStrin
g));
                     printf("I recv %s from %s:%u\r\n", cRecvBuff, ipAddrString, ntohs(sockaddr
inRemote.sin_port));

                    /*
                     * 回显
                     */
                     if (sendto(sockFd3, cRecvBuff, sstRecv, 0, (struct sockaddr *)&sockaddrinR
emote, sizeof(sockaddrinRemote)) != sstRecv) {
                         fprintf(stderr, "UDP echo server send error, line %d, errno %d.\r\n",
__LINE__, errno);
                    }
                } else {
                    fprintf(stderr, "UDP echo server read error, line %d.\r\n", __LINE__);
                }
            }
        }
    }

    close(sockFd1);
    close(sockFd2);
    close(sockFd3);

    return  (0);
}
```


21 MS-RTOS 管道

> 本章将介绍 MS-RTOS 管道的使用。

管道介绍

管道是一种半双工的通信方式，数据只能单向流动，MS-RTOS 的管道为命名管道，即在 `/dev` 目录下存在一个管道设备文件，写端以只写的方式调用 `ms_io_open` 函数打开该文件，读端以只读的方式打开该文件，返回管道的文件描述符，然后通过 MS-RTOS 的文件读写、操作接口读写、操作管道，完成进程间通信。

> **注意**
>
> 进程创建的管道如果已经完全被关闭（即写端、读端都关闭），管道设备文件会被删除。 而内核创建的管道，需要使用 `ms_io_unlink` 函数删除。

管道相关 API

下表展示了管道相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_pipe_drv_register		●
ms_pipe_dev_create	●	●
ms_pipe_dev_create_ex		●
ms_pipe_dev_write		●

ms_pipe_drv_register()

- **描述** 注册管道驱动
- **函数原型**

```
ms_err_t ms_pipe_drv_register(void);
```

- **参数** 无
- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_pipe_dev_create()

- **描述** 创建管道设备
- **函数原型**

```
ms_err_t ms_pipe_dev_create(const char *path, ms_size_t size);
```

- **参数**

输入/输出	参数	描述
[in]	path	管道设备文件的路径
[in]	size	管道的大小，必须 > 2

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_pipe_dev_create_ex()

- **描述** 创建管道设备，成功输出管道设备上下文供外部调用 `ms_pipe_dev_write` 函数
- **函数原型**

```
ms_err_t ms_pipe_dev_create_ex(const char *path, ms_size_t size, ms_ptr_t *ctx);
```

- **参数**

输入/输出	参数	描述
[in]	path	管道设备文件的路径
[in]	size	管道的大小，必须 > 2
[out]	ctx	管道设备上下文

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_pipe_dev_write()

- **描述** 写数据到管道设备
- **函数原型**

```
ms_ssize_t ms_pipe_dev_write(ms_ptr_t ctx, ms_const_ptr_t buf, ms_size_t len);
```

- **参数**

输入/输出	参数	描述
[in]	ctx	管道设备上下文
[in]	buf	需要写入的数据
[in]	len	需要写入的数据长度

- **返回值** 成功返回写入的数据长度，失败返回 -1
- **注意事项** 无
- **示例** 无

管道通信示例

写端进程创建并以只写方式打开管道设备文件：

```
#include <ms_rtos.h>
#include <stdlib.h>

int main (int argc, char **argv)
{
    int fd;

    if (ms_pipe_dev_create("/dev/pipe0", 128) < 0) {
        ms_printf("failed to create pipe\n");
        abort();
    }

    fd = ms_io_open("/dev/pipe0", O_WRONLY, 0666);

    while (1) {
        ms_io_write(fd, "hello", sizeof("hello") - 1);

        ms_thread_sleep_s(2);
    }

    ms_io_close(fd);

    return  (0);
}
```

读端进程以只读方式打开管道设备文件：

```
#include <ms_rtos.h>
#include <stdlib.h>

int main (int argc, char **argv)
{
    int fd;
    char buf[128];
    ms_ssize_t len;
```



```
    fd = ms_io_open("/dev/pipe0", O_RDONLY, 0666);

    while (1) {
        len = ms_io_read(fd, buf, sizeof(buf));

        if ((len < sizeof(buf)) && (len > 0)) {
            buf[len] = 0;
            ms_printf("i recv len %d %s\n", len, buf);
        } else {
            ms_printf("failed to read pipe!\n");
        }
    }

    ms_io_close(fd);

    return  (0);
}
```

管道支持 select 和 poll 操作:

```
#include <ms_rtos.h>
#include <stdlib.h>

int main (int argc, char **argv)
{
    int ret;
    int fd;
    char buf[128];
    ms_ssize_t len;
    ms_fd_set_t rfds;
    ms_timeval_t tv;

    fd = ms_io_open("/dev/pipe0", O_RDONLY, 0666);

    while (1) {
        FD_ZERO(&rfds);
        FD_SET(fd, &rfds);

        tv.tv_sec = 1;
        tv.tv_usec = 0;

        ret = ms_io_select(fd + 1, &rfds, NULL, NULL, &tv);
        if (ret > 0 && FD_ISSET(fd, &rfds)) {
            len = ms_io_read(fd, buf, sizeof(buf));
            if ((len < sizeof(buf)) && (len > 0)) {
                buf[len] = 0;
                ms_printf("i recv len %d %s\n", len, buf);
            } else {
                ms_printf("failed to read pipe!\n");
            }
        } else {
```



```
            ms_printf("failed to select pipe!\n");
        }
    }

    ms_io_close(fd);

    return  (0);
}
```

管道支持的 ioctl 命令

命令	描述	参数
MS_PIPE_CMD_GET_SIZE	获得管道的大小	ms_size_t 指针
MS_PIPE_CMD_GET_LEN	获得管道里的数据的长度	ms_size_t 指针
MS_PIPE_CMD_GET_SPACE	获得管道的剩余空间长度	ms_size_t 指针
MS_PIPE_CMD_FLUSH	清空管道数据	无
MS_PIPE_CMD_WAIT_EMPTY	等待管道数据被全部读出	等待超时时间（单位 ms），ms_uint32_t 指针
MS_PIPE_CMD_GET_R_TIMEOUT	获得读超时时间（单位 ms）	ms_uint32_t 指针
MS_PIPE_CMD_SET_R_TIMEOUT	设置读超时时间（单位 ms）	ms_uint32_t 指针
MS_PIPE_CMD_GET_W_TIMEOUT	获得写超时时间（单位 ms）	ms_uint32_t 指针
MS_PIPE_CMD_SET_W_TIMEOUT	设置写超时时间（单位 ms）	ms_uint32_t 指针

22 MS-RTOS 共享内存

> 本章将介绍 MS-RTOS 共享内存的使用。

共享内存介绍

共享内存是 MS-RTOS 多个进程都能访问的内存。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率不够高而专门设计的。它往往与其他通信机制（如读写锁）配合使用，来实现进程间的同步和通信。

共享内存相关 API

下表展示了共享内存相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_shm_drv_register		●
ms_shm_dev_create		●

ms_shm_drv_register()

- **描述** 注册共享内存驱动
- **函数原型**

```
ms_err_t ms_shm_drv_register(void);
```

- **参数** 无
- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_shm_dev_create()

- **描述** 创建共享内存设备
- **函数原型**

```
ms_err_t ms_shm_dev_create(const char *path, ms_addr_t base, ms_size_t size,
                           ms_bool_t mpu_protect);
```

- **参数**

输入/输出	参数	描述
[in]	path	共享内存设备文件的路径
[in]	base	共享内存的基地址
[in]	size	共享内存的大小
[in]	mpu_protect	是否使用 MPU 进行保护

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
    ms_shm_drv_register();
    ms_shm_dev_create("/dev/shm0", BSP_CFG_SHARED_RAM_BASE + 0 * 8192, 8192, MS_TRUE);
    ms_shm_dev_create("/dev/shm1", BSP_CFG_SHARED_RAM_BASE + 1 * 8192, 8192, MS_TRUE);
    ms_shm_dev_create("/dev/shm2", BSP_CFG_SHARED_RAM_BASE + 2 * 8192, 8192, MS_FALSE);
    ms_shm_dev_create("/dev/shm3", BSP_CFG_SHARED_RAM_BASE + 3 * 8192, 8192, MS_FLASE);
```

共享内存通信示例

写端进程打开共享内存设备文件，通过 `ms_io_ioctl` 获得共享内存的基地址和大小，并写入一个字符串：

```c
#include <ms_rtos.h>
#include <string.h>

int main (int argc, char **argv)
{
    ms_addr_t base;
    ms_size_t size;
    ms_handle_t rwlockid;
    int shmfd;

    shmfd = ms_io_open("/dev/shm0", O_WRONLY, 0666);

    ms_io_ioctl(shmfd, MS_SHM_CMD_GET_BASE, &base);
    ms_io_ioctl(shmfd, MS_SHM_CMD_GET_SIZE, &size);
    ms_io_ioctl(shmfd, MS_SHM_CMD_GET_RW_LOCK_ID, &rwlockid);

    ms_rwlock_lock_write(rwlockid, MS_TIMEOUT_FOREVER);
    strcpy((void *)base, "hello");
    ms_rwlock_unlock(rwlockid);

    ms_io_close(shmfd);

    return  (0);
```

```
}
```

读端进程也打开该共享内存设备文件，通过 `ms_io_ioctl` 获得共享内存的基地址和大小，并通过 `ms_printf` 函数打印存储于该共享内存的字符串：

```c
#include <ms_rtos.h>

int main (int argc, char **argv)
{
    ms_addr_t base;
    ms_size_t size;
    ms_handle_t rwlockid;
    int shmfd;

    shmfd = ms_io_open("/dev/shm0", O_RDONLY, 0666);

    ms_io_ioctl(shmfd, MS_SHM_CMD_GET_BASE, &base);
    ms_io_ioctl(shmfd, MS_SHM_CMD_GET_SIZE, &size);
    ms_io_ioctl(shmfd, MS_SHM_CMD_GET_RW_LOCK_ID, &rwlockid);

    ms_rwlock_lock_read(rwlockid, MS_TIMEOUT_FOREVER);
    ms_printf("%s\n", (char *)base);
    ms_rwlock_unlock(rwlockid);

    ms_io_close(shmfd);

    return  (0);
}
```

共享内存支持的 ioctl 命令

命令	描述	参数
MS_SHM_CMD_GET_BASE	获得共享内存的基地址	ms_addr_t 指针
MS_SHM_CMD_GET_SIZE	获得共享内存的大小	ms_size_t 指针
MS_SHM_CMD_GET_RW_LOCK_ID	获得共享内存的读写锁 ID	ms_handle_t 指针
MS_SHM_CMD_GET_W_SEQ	获得共享内存的写序列号	ms_uint32_t 指针
MS_SHM_CMD_W_COMMIT	写提交（写序列号将加一）	无

23 MS-RTOS 工具类函数

> 本章将介绍 MS-RTOS 工具类函数的使用。

工具类 API

下表展示了工具类函数在两个权限空间下是否可用：

API	用户空间	内核空间
stdin 标准输入文件		
ms_getc	●	●
ms_gets	●	●
stdout 标准输出文件		
ms_putc	●	●
ms_puts	●	●
ms_printf	●	●
字符串		
ms_snprintf	●	●
ms_strtol	●	●
ms_strtoul	●	●
ms_strtoull	●	●
ms_atoi	●	●
ms_atol	●	●
字符		
ms_isalnum	●	●
ms_isalpha	●	●
ms_iscntrl	●	●
ms_isdigit	●	●
ms_isgraph	●	●
ms_islower	●	●
ms_isprint	●	●

API	用户空间	内核空间
ms_ispunct	●	●
ms_isspace	●	●
ms_isupper	●	●
ms_isxdigit	●	●
ms_isblank	●	●
ms_isascii	●	●
ms_toascii	●	●
ms_tolower	●	●
ms_toupper	●	●
CRC		
ms_crc32	●	●
ms_file_crc32	●	●
其它		
ms_roundup_pow2_size	●	●

stdin 标准输入文件函数

MS-RTOS 提供了以下 stdin 标准输入文件函数，它们的实现比 C 库里的实现要小巧，可有效减少对 ROM 空间的消耗，建议采用以下的函数：

API	描述
char ms_getc(void)	从标准输入文件读取一个字符
ms_ssize_t ms_gets(char *buf, ms_size_t size)	从标准输入文件读取一个字符串

stdout 标准输出文件函数

MS-RTOS 提供了以下 stdout 标准输出文件函数，它们的实现比 C 库里的实现要小巧，可有效减少对 ROM 空间的消耗，建议采用以下的函数：

API	描述
void ms_putc(char ch)	打印一个字符到标准输出文件
void ms_puts(const char *string)	打印一个字符串到标准输出文件
void ms_printf(const char *fmt, ...)	打印格式化字符串到标准输出文件（不支持浮点）

字符串函数

MS-RTOS 提供了以下字符串函数，它们的实现比 C 库里的实现要小巧，可有效减少对 ROM 空间的消耗，建议采用以下的函数：

API	描述
int ms_snprintf(char *buf, ms_size_t buf_size, const char *fmt,...)	打印格式化字符串到缓冲区（不支持浮点）
ms_long_t ms_strtol(const char *cp, char **endp, ms_size_t base)	字符串按指定进制转换为有符号长整型
ms_ulong_t ms_strtoul(const char *cp, char **endp, ms_size_t base)	字符串按指定进制转换为无符号长整型
ms_ullong_t ms_strtoull(const char *cp, char **endp, ms_size_t base)	字符串按指定进制转换为无符号长长整型
int ms_atoi(const char *nptr)	将字符转换为整型
ms_long_t ms_atol(const char *nptr)	将字符转换为长整型

字符函数

MS-RTOS 也提供了 ctype.h 标准头文件的字符操作、判断函数，它们的实现比 C 库里的实现要小巧，可有效减少对 ROM 空间的消耗，建议采用以下的字符操作、判断函数：

API	描述
int ms_isalnum(int c)	判断字符是否为字母或者十进制数字
int ms_isalpha(int c)	判断字符是否为字母
int ms_iscntrl(int c)	判断字符是否为控制字符
int ms_isdigit(int c)	判断字符是否为十进制数字
int ms_isgraph(int c)	判断字符是否为图形字符
int ms_islower(int c)	判断字符是否为小写字母
int ms_isprint(int c)	判断字符是否为可打印字符
int ms_ispunct(int c)	判断字符是否为标点符号
int ms_isspace(int c)	判断字符是否为空白字符
int ms_isupper(int c)	判断字符是否为大写字母
int ms_isxdigit(int c)	判断字符是否为十六进制数字
int ms_isblank(int c)	判断字符是否为空白符
int ms_isascii(int c)	判断字符是否为 ASCII 字符

API	描述
int ms_toascii(int c)	将字符转换为 ASCII 字符
int ms_tolower(int c)	将小写字母转换为大写字母
int ms_toupper(int c)	将大写字母转换为小写字母

ms_crc32()

- **描述** 计算数据的 CRC32
- **函数原型**

```
ms_uint32_t ms_crc32(ms_uint32_t crc, ms_const_ptr_t buf, ms_size_t size);
```

- **参数**

输入/输出	参数	描述
[in]	crc	CRC32 初始值
[in]	buf	数据
[in]	size	数据的大小

- **返回值** 数据的 CRC32
- **注意事项** 无
- **示例**

```
int main(int argc, char *argv[])
{
    char data[128];
    ms_uint32_t crc;

    // do some thing

    crc = ms_crc32(0U, data, sizeof(data));

    // do some thing

    return 0;
}
```

ms_file_crc32()

- **描述** 计算指定文件的 CRC32
- **函数原型**


```
ms_err_t ms_file_crc32(const char *path, ms_uint32_t *crc);
```

- **参数**

输入/输出	参数	描述
[in]	path	文件路径
[out]	crc	文件的 CRC32

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
int main(int argc, char *argv[])
{
    ms_uint32_t crc;

    // do some thing

    ms_file_crc32("/nor/aaa.bin", &crc);

    // do some thing

    return 0;
}
```

ms_roundup_pow2_size()

- **描述** 向上圆整到 2 的次方大小
- **函数原型**

```
ms_uint32_t ms_roundup_pow2_size(ms_uint32_t size);
```

- **参数**

输入/输出	参数	描述
[in]	size	大小

- **返回值** 输入参数向上圆整到 2 的次方后的值
- **注意事项** 无
- **示例**


```c
int main(int argc, char *argv[])
{
    ms_uint32_t size;

    // do some thing

    size = ms_roundup_pow2_size(100);

    // do some thing

    return 0;
}
```


24 MS-RTOS FIFO

> 本章将介绍 MS-RTOS FIFO 的使用。

FIFO 介绍

FIFO 是一种先入先出的数据结构，MS-RTOS 的 FIFO 做到了单读单写免锁，如果需要多读或多写，请在 FIFO 操作的外部加锁（如互斥量）。

FIFO 相关数据类型

类型	描述
ms_fifo_t	FIFO 类型
ms_fifo_evt_cb_t	FIFO 的事件回调函数类型

ms_fifo_t

FIFO 的类型为 `ms_fifo_t`，FIFO 使用前需要定义，它可以是一个全局变量，也可以嵌入到其它数据类型当中（如结构体）作为一个成员变量。

```
typedef struct {
    ms_uint8_t        *buf;
    ms_size_t          size;
    ms_size_t          r;
    ms_size_t          w;
    ms_fifo_evt_cb_t   evt_cb;
} ms_fifo_t;
```

参数	说明
buf	FIFO 的基地址
size	FIFO 的大小
r	FIFO 的读指针
w	FIFO 的写指针
evt_cb	FIFO 的事件回调函数

FIFO 相关 API

下表展示了 FIFO 相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_fifo_init	●	●
ms_fifo_is_ready	●	●
ms_fifo_destroy	●	●
ms_fifo_set_evt_cb	●	●
ms_fifo_reset	●	●
ms_fifo_space	●	●
ms_fifo_len	●	●
ms_fifo_size	●	●
ms_fifo_is_empty	●	●
ms_fifo_is_full	●	●
ms_fifo_put	●	●
ms_fifo_get	●	●
ms_fifo_peek	●	●
ms_fifo_read_linear_blk	●	●
ms_fifo_skip	●	●
ms_fifo_write_linear_blk	●	●
ms_fifo_advance	●	●

ms_fifo_init()

- **描述** 初始化 FIFO
- **函数原型**

```
ms_err_t ms_fifo_init(ms_fifo_t *fifo, ms_ptr_t buf, ms_size_t size);
```

- **参数**

输入/输出	参数	描述
[in]	fifo	FIFO 的指针
[in]	buf	FIFO 的缓冲区的基地址，不能为空指针
[in]	size	FIFO 的缓冲区的大小

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_fifo_is_ready()

- **描述** 判断 FIFO 是否初始化好
- **函数原型**

```
ms_bool_t ms_fifo_is_ready(const ms_fifo_t *fifo);
```

- **参数**

输入/输出	参数	描述
[in]	fifo	FIFO 的指针

- **返回值** MS_TRUE：已经初始化好，MS_FALSE：未初始化好
- **注意事项** 无
- **示例** 无

ms_fifo_destroy()

- **描述** 销毁 FIFO
- **函数原型**

```
void ms_fifo_destroy(ms_fifo_t *fifo);
```

- **参数**

输入/输出	参数	描述
[in]	fifo	FIFO 的指针

- **返回值** 无
- **注意事项** 无
- **示例** 无

ms_fifo_set_evt_cb()

- **描述** 设置 FIFO 的事件回调函数
- **函数原型**


```
void ms_fifo_set_evt_cb(ms_fifo_t *fifo, ms_fifo_evt_cb_t evt_cb);
```

- **参数**

输入/输出	参数	描述
[in]	fifo	FIFO 的指针
[in]	evt_cb	FIFO 的事件回调函数指针

- **返回值** 无
- **注意事项** 无
- **示例** 无

ms_fifo_reset()

- **描述** 复位 FIFO
- **函数原型**

```
void ms_fifo_reset(ms_fifo_t *fifo);
```

- **参数**

输入/输出	参数	描述
[in]	fifo	FIFO 的指针

- **返回值** 无
- **注意事项** 无
- **示例** 无

ms_fifo_space()

- **描述** 获得 FIFO 的空闲空间长度
- **函数原型**

```
ms_size_t ms_fifo_space(const ms_fifo_t *fifo);
```

- **参数**

输入/输出	参数	描述
[in]	fifo	FIFO 的指针

- **返回值** FIFO 的空闲空间长度
- **注意事项** 无
- **示例** 无

ms_fifo_len()

- **描述** 获得 FIFO 里的数据长度
- **函数原型**

```
ms_size_t ms_fifo_len(const ms_fifo_t *fifo);
```

- **参数**

输入/输出	参数	描述
[in]	fifo	FIFO 的指针

- **返回值** 获得 FIFO 里的数据长度
- **注意事项** 无
- **示例** 无

ms_fifo_size()

- **描述** 获得 FIFO 的大小
- **函数原型**

```
ms_size_t ms_fifo_size(const ms_fifo_t *fifo);
```

- **参数**

输入/输出	参数	描述
[in]	fifo	FIFO 的指针

- **返回值** FIFO 的大小
- **注意事项** 无
- **示例** 无

ms_fifo_is_empty()

- **描述** 判断 FIFO 是否为空
- **函数原型**


```
ms_bool_t ms_fifo_is_empty(const ms_fifo_t *fifo);
```

- **参数**

输入/输出	参数	描述
[in]	fifo	FIFO 的指针

- **返回值** MS_TRUE：空，MS_FALSE：不空，有数据
- **注意事项** 无
- **示例** 无

ms_fifo_is_full()

- **描述** 判断 FIFO 是否为满
- **函数原型**

```
ms_bool_t ms_fifo_is_full(const ms_fifo_t *fifo);
```

- **参数**

输入/输出	参数	描述
[in]	fifo	FIFO 的指针

- **返回值** MS_TRUE：满，MS_FALSE：不满，有空位
- **注意事项** 无
- **示例** 无

ms_fifo_put()

- **描述** 将数据写入 FIFO
- **函数原型**

```
ms_size_t ms_fifo_put(ms_fifo_t *fifo, ms_const_ptr_t buf, ms_size_t len);
```

- **参数**

输入/输出	参数	描述
[in]	fifo	FIFO 的指针
[in]	buf	需要写入的数据，不能为空指针

输入/输出	参数	描述
[in]	len	需要写入的长度

- **返回值** 成功写入的长度
- **注意事项** 无
- **示例** 无

ms_fifo_get()

- **描述** 从 FIFO 读出数据
- **函数原型**

```
ms_size_t ms_fifo_get(ms_fifo_t *fifo, ms_ptr_t buf, ms_size_t len);
```

- **参数**

输入/输出	参数	描述
[in]	fifo	FIFO 的指针
[in]	buf	用于存放读出的数据的缓冲区，不能为空指针
[in]	len	需要读出的长度

- **返回值** 成功读出的长度
- **注意事项** 无
- **示例** 无

ms_fifo_peek()

- **描述** 从 FIFO 读出数据，但不调整读指针
- **函数原型**

```
ms_size_t ms_fifo_peek(const ms_fifo_t *fifo, ms_size_t skip_len, ms_ptr_t buf, ms_size_t le
n);
```

- **参数**

输入/输出	参数	描述
[in]	fifo	FIFO 的指针
[in]	skip_len	需要跳过读的数据长度
[in]	buf	用于存放读出的数据的缓冲区，不能为空指针

输入/输出	参数	描述
[in]	len	需要读出的长度

- **返回值** 成功读出的长度
- **注意事项** 无
- **示例** 无

ms_fifo_read_linear_blk()

- **描述** 获得 FIFO 线性读内存块地址和长度
- **函数原型**

```
ms_size_t ms_fifo_read_linear_blk(const ms_fifo_t *fifo, ms_const_ptr_t *pos);
```

- **参数**

输入/输出	参数	描述
[in]	fifo	FIFO 的指针
[in]	pos	线性读内存块地址

- **返回值** 线性读内存块长度
- **注意事项** 无
- **示例** 无

ms_fifo_skip()

- **描述** 跳过读指定长度的数据
- **函数原型**

```
ms_size_t ms_fifo_skip(ms_fifo_t *fifo, ms_size_t len);
```

- **参数**

输入/输出	参数	描述
[in]	fifo	FIFO 的指针
[in]	len	需要跳过读的数据长度

- **返回值** 成功跳过读的数据长度
- **注意事项** 无
- **示例** 无

ms_fifo_write_linear_blk()

- **描述** 获得 FIFO 线性写内存块地址和长度
- **函数原型**

```
ms_size_t ms_fifo_write_linear_blk(const ms_fifo_t *fifo, ms_ptr_t *pos);
```

- **参数**

输入/输出	参数	描述
[in]	fifo	FIFO 的指针
[in]	pos	线性写内存块地址

- **返回值** 线性写内存块长度
- **注意事项** 无
- **示例** 无

ms_fifo_advance()

- **描述** 提前预留指定长度的写空间
- **函数原型**

```
ms_size_t ms_fifo_advance(ms_fifo_t *fifo, ms_size_t len);
```

- **参数**

输入/输出	参数	描述
[in]	fifo	FIFO 的指针
[in]	len	需要提前预留写空间的长度

- **返回值** 成功提前预留写空间的长度
- **注意事项** 无
- **示例** 无

25 MS-RTOS 链表

> 本章将介绍 MS-RTOS 链表的使用。
>
> MS-RTOS 实现了一个双向循环链表，除了在 MS-RTOS 内部广泛使用外，MS-RTOS 的应用程序也可以使用该双向循环链表。

链表介绍

双向循环链表的链头和节点都是 ms_list_head_t 类型，它可以是一个全局变量，也可以嵌入到其它数据类型当中（如结构体）作为一个成员变量。只要得到节点的地址（节点的指针），就可以使用形如 MS_CONTAINER_OF(pnode, xxx_struct, xxx_member) 的宏来获得容器（如结构体）的地址（容器的指针）。

链表的遍历有两个宏： ms_list_for_each 和 ms_list_for_each_safe。

- ms_list_for_each_safe 是安全的，这里的安全指的是链表遍历过程如果有节点删除操作，链表遍历是安全的；
- ms_list_for_each 是不安全的。

可以根据需要选择合适的链表遍历宏。

> **注意**
>
> 以下的链表操作都不是线程安全的，如果需要线程安全，请在链表操作的外部加锁（如互斥量）。

链表相关数据类型

类型	描述
ms_list_head_t	双向链表节点类型

ms_list_head_t

MS-RTOS 中使用的通用双向链表节点类型为 ms_list_head_t，链表节点使用前需要定义，它可以是一个全局变量，也可以嵌入到其它数据类型当中（如结构体）作为一个成员变量。全局链表可以使用宏 MS_LIST_HEAD 进行定义，另外可以使用宏 MS_LIST_HEAD_INIT 对链表进行初始化。

```
typedef struct ms_list_head {
    struct ms_list_head *next;
    struct ms_list_head *prev;
} ms_list_head_t;
```

参数	说明
next	指向下一个节点
prev	指向前一个节点

链表相关 API

下表展示了链表相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
MS_LIST_HEAD	●	●
MS_LIST_HEAD_INIT	●	●
ms_list_is_empty	●	●
ms_list_is_head	●	●
ms_list_is_tail	●	●
ms_list_is_only_one	●	●
ms_list_add	●	●
ms_list_add_tail	●	●
ms_list_del	●	●
ms_list_del_init	●	●
ms_list_for_each	●	●
ms_list_for_each_safe	●	●

MS_LIST_HEAD()

- **描述** 定义一个链表
- **宏实现**

```
#define MS_LIST_HEAD(name) \
ms_list_head_t (name) = { &(name), &(name)}
```

- **参数**

输入/输出	参数	描述
[in]	name	链表变量名

- **返回值** 无
- **注意事项** 无
- **示例** 无

MS_LIST_HEAD_INIT()

- **描述** 初始化链表

- **宏实现**

```
#define MS_LIST_HEAD_INIT(p) \
    do {                     \
        (p)->next = (p);     \
        (p)->prev = (p);     \
    } while (0)
```

- **参数**

输入/输出	参数	描述
[in]	p	链表的指针

- **返回值** 无
- **注意事项** 无
- **示例** 无

ms_list_is_empty()

- **描述** 判断一个链表是否为空
- **函数原型**

```
static MS_FORCE_INLINE ms_bool_t ms_list_is_empty(const ms_list_head_t *list);
```

- **参数**

输入/输出	参数	描述
[in]	list	链表的指针

- **返回值** MS_TRUE：链表为空，MS_FALSE：链表不为空
- **注意事项** 无
- **示例** 无

ms_list_is_head()

- **描述** 判断一个节点是否为链表的表头
- **函数原型**

```
static MS_FORCE_INLINE ms_bool_t ms_list_is_head(const ms_list_head_t *entry,
                                                 const ms_list_head_t *list);
```

- **参数**

输入/输出	参数	描述
[in]	entry	节点
[in]	list	链表的指针

- **返回值** MS_TRUE：节点是链表的表头，MS_FALSE：节点不是链表的表头
- **注意事项** 无
- **示例** 无

ms_list_is_tail()

- **描述** 判断一个节点是否为链表的末尾
- **函数原型**

```
static MS_FORCE_INLINE ms_bool_t ms_list_is_tail(const ms_list_head_t *entry,
                                                 const ms_list_head_t *list);
```

- **参数**

输入/输出	参数	描述
[in]	entry	节点
[in]	list	链表指针

- **返回值** MS_TRUE：节点是链表的末尾，MS_FALSE：节点不是链表的末尾
- **注意事项** 无
- **示例** 无

ms_list_is_only_one()

- **描述** 判断一个节点是否为链表的唯一节点
- **函数原型**

```
static MS_FORCE_INLINE ms_bool_t ms_list_is_only_one(const ms_list_head_t *entry,
                                                     const ms_list_head_t *list);
```

- **参数**

输入/输出	参数	描述
[in]	entry	节点
[in]	list	链表的指针

- **返回值** MS_TRUE：节点是链表的唯一节点，MS_FALSE：节点不是链表的唯一节点
- **注意事项** 无
- **示例** 无

ms_list_add()

- **描述** 在链表头插入一个节点
- **函数原型**

```
static MS_FORCE_INLINE void ms_list_add(ms_list_head_t *new_entry,
                                        ms_list_head_t *list);
```

- **参数**

输入/输出	参数	描述
[in]	new_entry	需要插入的节点
[in]	list	链表的指针

- **返回值** 无
- **注意事项** 无
- **示例** 无

ms_list_add_tail()

- **描述** 在链表末尾插入一个节点
- **函数原型**

```
static MS_FORCE_INLINE void ms_list_add_tail(ms_list_head_t *new_entry,
                                             ms_list_head_t *list);
```

- **参数**

输入/输出	参数	描述
[in]	new_entry	需要插入的节点
[in]	list	链表的指针

- **返回值** 无
- **注意事项** 无
- **示例** 无

ms_list_del()

- **描述** 从链表中移除一个节点
- **函数原型**

```
static MS_FORCE_INLINE void ms_list_del(ms_list_head_t *const entry);
```

- **参数**

输入/输出	参数	描述
[in]	entry	需要移除的节点

- **返回值** 无
- **注意事项** 被移除的节点的 prev、next 指针仍指向原节点
- **示例** 无

ms_list_del_init()

- **描述** 从链表中移除一个节点，并更新被移除节点中的 prev、next 指针
- **函数原型**

```
static MS_FORCE_INLINE void ms_list_del_init(ms_list_head_t *entry);
```

- **参数**

输入/输出	参数	描述
[in]	entry	需要移除的节点

- **返回值** 无
- **注意事项** 被移除的节点的 prev、next 指针指向 NULL
- **示例** 无

ms_list_for_each()

- **描述** 非安全方式遍历一个链表
- **宏实现**

```
#define ms_list_for_each(itervar, list) \
    for (itervar = (list)->next; itervar != (list); itervar = itervar->next)
```

- **参数**

输入/输出	参数	描述
[in]	itervar	临时迭代变量
[in]	list	链表的指针

- **返回值** 无
- **注意事项** 无
- **示例** 无

ms_list_for_each_safe()

- **描述** 安全方式遍历一个链表
- **宏实现**

```
#define ms_list_for_each_safe(itervar, save_var, list) \
    for (itervar = (list)->next, save_var = (list)->next->next; \
        itervar != (list); \
        itervar = save_var, save_var = save_var->next)
```

- **参数**

输入/输出	参数	描述
[in]	itervar	临时迭代变量
[in]	save_var	临时保存变量
[in]	list	链表的指针

- **返回值** 无
- **注意事项** 无
- **示例** 无

26 MS-RTOS APP 启动器

> 本章将介绍 MS-RTOS APP 启动器 launcher 的使用。

libmslauncher 使用

BSP 需要链接 `libmslauncher.a` 静态库，即 `bspxxx.mk` 的 `LOCAL_DEPEND_LIB` 需要添加 `-lmslauncher`。

launcher 相关 API

下表展示了 launcher 相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_launcher_init		●
ms_apps_update		●
ms_app_start	●	●
ms_apps_start	●	●

ms_launcher_init()

- **描述** 初始化 launcher
- **函数原型**

```
ms_err_t ms_launcher_init(const void *sign_key, ms_size_t sign_key_len);
```

- **参数**

输入/输出	参数	描述
[in]	sign_key	APP 数字签名公钥，空指针时不使用 APP 数字签名
[in]	sign_key_len	APP 数字签名公钥长度

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

在 BSP 中，一般由启动线程 `boot_thread` 来调用：

```
/* Autogenerated by imgtool.py, do not edit. */
const unsigned char rsa_pub_key[] = {
    ...
```



```
};
const unsigned int rsa_pub_key_len = 270;

static void boot_thread(ms_ptr_t arg)
{
    // do something

    ms_launcher_init(rsa_pub_key, rsa_pub_key_len);

    // do something
}
```

ms_apps_update()

- **描述** 完成 APP 镜像、启动参数文件等的升级
- **函数原型**

```
ms_err_t ms_apps_update(const char *update_req_path, const char *log_path);
```

- **参数**

输入/输出	参数	描述
[in]	update_req_path	升级请求文件的路径
[in]	log_path	升级日志文件的路径

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

  在 BSP 中，一般由启动线程 `boot_thread` 来调用：

```
static void boot_thread(ms_ptr_t arg)
{
    // do something

    ms_launcher_init(rsa_pub_key, rsa_pub_key_len);

    ms_apps_update("/nor/update/firmware/update_req", "/nor/update/firmware/update_log");

    // do something
}
```

ms_apps_start()

- **描述** 解析指定的启动参数文件，完成启动参数文件指定的共享内存、管道、全局 IPC 设备的创建，依次**启动具有自动启动属性**的 APP

- **函数原型**

```
ms_err_t ms_apps_start(const char *boot_param_file);
```

- **参数**

输入/输出	参数	描述
[in]	boot_param_file	启动参数文件的名字，不能为空指针

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

  在 BSP 中，一般由启动线程 `boot_thread` 来调用：

```
static void boot_thread(ms_ptr_t arg)
{
    // do some thing

    ms_launcher_init(rsa_pub_key, rsa_pub_key_len);

    ms_apps_update("/nor/update/firmware/update_req", "/nor/update/firmware/update_log");

    ms_apps_start("ms-boot-param.dtb"); /* boot_param_file 参数为已安装到 flash 目录下的启动参数文件 */

    // do some thing
}
```

  在 APP 中：

```
int main(int argc, char *argv[])
{
    ms_apps_start("ms-boot-param.dtb");

    return 0;
}
```

ms_app_start()

- **描述** 启动指定启动参数文件中的指定的 APP
- **函数原型**

```
ms_err_t ms_app_start(const char *boot_param_file, const char *app_name, ms_pid_t *pid);
```

- **参数**

输入/输出	参数	描述
[in]	boot_param_file	启动参数文件的名字，不能为空指针
[in]	app_name	APP 的名字，不能为空指针
[out]	pid	进程的 ID

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

  在 APP 中：

```
int main(int argc, char *argv[])
{
    ms_pid_t pid;

    ms_app_start("ms-boot-param.dtb", "app1", &pid); /* app_name 为启动参数中 APP 的名字，而不是镜像文件名 */

    // do some thing

    return 0;
}
```


27 MS-RTOS 升级

> 本章将介绍 MS-RTOS 升级功能相关接口的使用。

libmspatch 使用

如果要使用差分函数 `ms_patch`，应用程序需要链接 `libmspatch.a` 静态库，即 `xxx.mk` 的 `LOCAL_DEPEND_LIB` 需要添加 `-lmspatch`。

BSP 需要链接 `libmskpatch.a` 静态库，即 `bspxxx.mk` 的 `LOCAL_DEPEND_LIB` 需要添加 `-lmskpatch`。

升级相关 API

下表展示了升级功能相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_rtos_update	●	●
ms_patch	●	●

ms_rtos_update()

- **描述** MS-RTOS 将文件系统 CACHE 回写到磁盘后进行重启并完成 OS、APP 镜像、启动参数文件等的升级
- **函数原型**

```
ms_err_t ms_rtos_update(void);
```

- **参数** 无
- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_patch()

- **描述** 使用旧的文件和差分包生成新的文件
- **函数原型**

```
int ms_patch(const char *old_file, const char *new_file, const char *patch_file);
```

- **参数**

输入/输出	参数	描述
[in]	old_file	旧文件的路径
[in]	new_file	需要创建的新文件的路径
[in]	patch_file	差分包的文件路径

- **返回值** 成功返回 0，失败返回非 0
- **注意事项** 无
- **示例** 无

28 MS-RTOS 电源管理

> 本章将介绍 MS-RTOS 电源管理 PM 的使用。

PM 相关数据类型

类型	描述
ms_pm_sleep_mode_t	CPU 休眠模式
ms_pm_run_mode_t	CPU 运行模式
ms_pm_boot_mode_t	CPU 启动模式
ms_pm_event_t	电源管理事件

ms_pm_sleep_mode_t

CPU 休眠模式类型，可取的值为以下的宏：

宏	含义
MS_PM_SLEEP_MODE_NONE	CPU 不休眠，系统空闲时执行 NOP 指令，不使用 Tick-less
MS_PM_SLEEP_MODE_IDLE	系统空闲时执行 CPU idle 指令，使用 Tick-less，任一中断唤醒
MS_PM_SLEEP_MODE_LIGHT	CPU 浅休眠，目前与 MS_PM_SLEEP_MODE_IDLE 相同
MS_PM_SLEEP_MODE_DEEP	CPU 深度休眠，挂起设备，使用低功耗定时器，任一中断唤醒
MS_PM_SLEEP_MODE_STANDBY	CPU 待机模式，挂起设备，只能被某些中断唤醒，唤醒后重新执行程序，可通过启动模式与关机模式进行区分
MS_PM_SLEEP_MODE_SHUTDOWN	CPU 关机模式，挂起设备，彻底关机

ms_pm_run_mode_t

CPU 运行模式类型，可取的值为以下的宏：

宏	含义
MS_PM_RUN_MODE_HIGH_SPEED	CPU 高速运行
MS_PM_RUN_MODE_NORMAL_SPEED	CPU 正常速度运行

宏	含义
MS_PM_RUN_MODE_MEDIUM_SPEED	CPU 中等速度运行
MS_PM_RUN_MODE_LOW_SPEED	CPU 低速运行

ms_pm_boot_mode_t

CPU 启动模式类型，可取的值为以下的宏：

宏	含义
MS_PM_BOOT_MODE_COLD	冷机启动（关机模式后启动）
MS_PM_BOOT_MODE_REBOOT	重启（热启动）
MS_PM_BOOT_MODE_STANDBY	CPU 从待机模式唤醒（热启动）

ms_pm_event_t

电源管理事件类型，可取的值为以下的宏：

宏	含义
MS_PM_EVENT_SLEEP_ENTER	CPU 进入休眠
MS_PM_EVENT_SLEEP_EXIT	CPU 退出休眠

PM 相关 API

下表展示了电源管理相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_pm_request_sleep_mode		●
ms_pm_release_sleep_mode		●
ms_pm_set_default_sleep_mode		●
ms_pm_set_run_mode		●
ms_pm_get_run_mode		●
ms_pm_set_boot_mode		●
ms_pm_get_boot_mode		●

用户空间不能使用电源管理相关的 API，只能通过对 `/dev/pm` 设备操作达到电源管理的目的。如果 APP 没有对 `/dev/pm` 设备的访问权限，则 APP 不能进行电源管理相关的操作。

ms_pm_request_sleep_mode()

- **描述** 请求进入指定的 CPU 休眠模式
- **函数原型**

```
ms_err_t ms_pm_request_sleep_mode(ms_pm_sleep_mode_t sleep_mode);
```

- **参数**

输入/输出	参数	描述
[in]	sleep_mode	CPU 休眠模式

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_pm_release_sleep_mode()

- **描述** 取消请求指定的 CPU 休眠模式
- **函数原型**

```
ms_err_t ms_pm_release_sleep_mode(ms_pm_sleep_mode_t sleep_mode);
```

- **参数**

输入/输出	参数	描述
[in]	sleep_mode	CPU 休眠模式

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_pm_set_default_sleep_mode()

- **描述** 设置默认的 CPU 休眠模式
- **函数原型**

```
ms_err_t ms_pm_set_default_sleep_mode(ms_pm_sleep_mode_t sleep_mode);
```

- **参数**

输入/输出	参数	描述
[in]	sleep_mode	CPU 休眠模式

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_pm_set_run_mode()

- **描述** 设置 CPU 运行模式
- **函数原型**

```
ms_err_t ms_pm_set_run_mode(ms_pm_run_mode_t run_mode);
```

- **参数**

输入/输出	参数	描述
[in]	run_mode	CPU 运行模式

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_pm_get_run_mode()

- **描述** 获得当前的 CPU 运行模式
- **函数原型**

```
ms_pm_run_mode_t ms_pm_get_run_mode(void);
```

- **参数** 无
- **返回值** 当前的 CPU 运行模式
- **注意事项** 无
- **示例** 无

ms_pm_set_boot_mode()

- **描述** 设置 CPU 启动模式
- **函数原型**

```
ms_err_t ms_pm_set_boot_mode(ms_pm_boot_mode_t boot_mode);
```

- **参数**

输入/输出	参数	描述
[in]	boot_mode	CPU 启动模式

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_pm_get_boot_mode()

- **描述** 获得 CPU 启动模式
- **函数原型**

```
ms_pm_boot_mode_t ms_pm_get_boot_mode(void);
```

- **参数** 无
- **返回值** CPU 的启动模式
- **注意事项** 无
- **示例** 无

/dev/pm 设备 ioctl 命令

ioctl 命令及相关说明如下表所示:

命令	描述	参数
MS_PM_CMD_REQUEST_SLEEP_MODE	请求进入指定的 CPU 休眠模式	ms_pm_sleep_mode_t 指针
MS_PM_CMD_RELEASE_SLEEP_MODE	取消请求进入指定的 CPU 休眠模式	ms_pm_sleep_mode_t 指针
MS_PM_CMD_SET_DEFAULT_SLEEP_MODE	设置默认的 CPU 休眠模式	ms_pm_sleep_mode_t 指针
MS_PM_CMD_SET_RUN_MODE	设置 CPU 运行模式	ms_pm_run_mode_t 指针
MS_PM_CMD_GET_RUN_MODE	获得当前的 CPU 运行模式	ms_pm_run_mode_t 指针
MS_PM_CMD_GET_BOOT_MODE	获得 CPU 的启动模式	ms_pm_boot_mode_t 指针

29 MS-RTOS CACHE 操作

> 本章将介绍 MS-RTOS CPU CACHE 操作相关接口的使用。

CACHE 介绍

CPU CACHE 是 CPU 性能的加速器，CPU CACHE 缓存了 CPU 频繁访问的数据，CPU 访问 CACHE 的速度要比访问 RAM 的速度快得多，这样 CPU 就能以较快的速度访问到所需的数据。

CACHE 按缓存的数据类型来分，可分为数据 CACHE 和指令 CACHE，按层次来分，可分为 L1-CACHE、L2-CACHE.....

由于 DMA（直接内存访问）搬运数据并不通过 CPU 的运算单元，所以 DMA 搬运数据后，会出现 CACHE 内容与内存里的值不一致的情况，某些高端处理器具备 DMA CACHE 一致性的特性，但很遗憾，目前应用的很多 MCU 都没有该特性，所以需要软件（主要是操作系统和驱动程序）手动维护 CACHE 与内存的一致性。

CACHE 相关 API

下表展示了 CACHE 操作相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_dcache_clean		●
ms_dcache_flush		●
ms_dcache_invalid		●
ms_dma_dcache_clean		●
ms_dma_dcache_flush		●
ms_dma_dcache_invalid		●
ms_icache_invalid		●
ms_cache_text_update		●

ms_dcache_clean()

- **描述** 回写指定范围内存的 D-CACHE
- **函数原型**

```
void ms_dcache_clean(ms_ptr_t ptr, ms_size_t len);
```

- **参数**

输入/输出	参数	描述
[in]	ptr	起始地址
[in]	len	长度

- **返回值** 无
- **注意事项** 无
- **示例** 无

ms_dcache_flush()

- **描述** 回写并无效指定范围内存的 D-CACHE
- **函数原型**

```
void ms_dcache_flush(ms_ptr_t ptr, ms_size_t len);
```

- **参数**

输入/输出	参数	描述
[in]	ptr	起始地址
[in]	len	长度

- **返回值** 无
- **注意事项** 无
- **示例** 无

ms_dcache_invalid()

- **描述** 无效指定范围内存的 D-CACHE（非 D-CACHE 行大小对齐部分将会回写并无效）
- **函数原型**

```
void ms_dcache_invalid(ms_ptr_t ptr, ms_size_t len);
```

- **参数**

输入/输出	参数	描述
[in]	ptr	起始地址
[in]	len	长度

- **返回值** 无
- **注意事项** 无

- **示例** 无

ms_dma_dcache_clean()

- **描述** 回写指定范围 DMA 内存的 D-CACHE，在具备 DMA 内存一致性特性的 CPU 上，将变成空操作
- **函数原型**

```
void ms_dma_dcache_clean(ms_ptr_t ptr, ms_size_t len);
```

- **参数**

输入/输出	参数	描述
[in]	ptr	起始地址
[in]	len	长度

- **返回值** 无
- **注意事项** 无
- **示例** 无

ms_dma_dcache_flush()

- **描述** 回写并无效指定范围 DMA 内存的 D-CACHE，在具备 DMA 内存一致性特性的 CPU 上，将变成空操作
- **函数原型**

```
void ms_dma_dcache_flush(ms_ptr_t ptr, ms_size_t len);
```

- **参数**

输入/输出	参数	描述
[in]	ptr	起始地址
[in]	len	长度

- **返回值** 无
- **注意事项** 无
- **示例** 无

ms_dma_dcache_invalid()

- **描述** 无效指定范围 DMA 内存的 D-CACHE（非 D-CACHE 行大小对齐部分将会回写并无效），在具备 DMA 内存一致性特性的 CPU 上，将变成空操作
- **函数原型**


```
void ms_dma_dcache_invalid(ms_ptr_t ptr, ms_size_t len);
```

- **参数**

输入/输出	参数	描述
[in]	ptr	起始地址
[in]	len	长度

- **返回值** 无
- **注意事项** 无
- **示例** 无

ms_icache_invalid()

- **描述** 无效指定范围内存的 I-CACHE
- **函数原型**

```
void ms_icache_invalid(ms_ptr_t ptr, ms_size_t len);
```

- **参数**

输入/输出	参数	描述
[in]	ptr	起始地址
[in]	len	长度

- **返回值** 无
- **注意事项** 无
- **示例** 无

ms_cache_text_update()

- **描述** 对指定范围内存进行代码更新（执行 D-CACHE 回写和 I-CACHE 无效），一般在代码搬运和代码修改后需要做代码更新
- **函数原型**

```
void ms_cache_text_update(ms_ptr_t ptr, ms_size_t len);
```

- **参数**

输入/输出	参数	描述
[in]	ptr	起始地址
[in]	len	长度

- **返回值** 无
- **注意事项** 无
- **示例** 无

30 MS-RTOS 虚拟内存管理

> 本章将介绍 MS-RTOS 虚拟内存管理 VMM 相关接口的使用。

VMM 相关 API

下表展示了虚拟内存管理 VMM 相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_vmm_ioremap_ex		●
ms_vmm_ioremap		●
ms_vmm_iounmap		●
ms_vmm_kmap_in		●
ms_vmm_kmap_v2p		●
ms_vmm_kmap_p2v		●

ms_vmm_ioremap_ex()

- **描述** 以指定的映射属性对指定范围的物理地址进行重映射
- **函数原型**

```
ms_addr_t ms_vmm_ioremap_ex(ms_addr_t phy_addr, ms_size_t size, ms_uint32_t attr);
```

- **参数**

输入/输出	参数	描述
[in]	phy_addr	起始物理地址（内部会向下对齐到页面大小）
[in]	size	长度（内部会向上圆整到页面大小）
[in]	attr	映射属性

其中映射属性为以下的宏组合：

宏	含义
MS_MMU_ATTR_R	可读
MS_MMU_ATTR_W	可写
MS_MMU_ATTR_X	可执行

宏	含义
MS_MMU_ATTR_RX	可读可执行
MS_MMU_ATTR_RW	可读写
MS_MMU_ATTR_RWX	可读写可执行
MS_MMU_ATTR_CACHE	可以 CACHE
MS_MMU_ATTR_WB	可以写缓冲
MS_MMU_ATTR_GLOGAL	全局映射（映射关系对所有进程有效）
MS_MMU_ATTR_USER	用户空间页面（用户态可访问）

- **返回值** 成功返回虚拟地址，失败时返回 -1
- **注意事项** 无
- **示例** 无

ms_vmm_ioremap()

- **描述** 以可读写不可 CACHE 和不可写缓冲的属性进行重映射
- **函数原型**

```
ms_addr_t ms_vmm_ioremap(ms_addr_t phy_addr, ms_size_t size);
```

- **参数**

输入/输出	参数	描述
[in]	phy_addr	起始物理地址（内部会向下对齐到页面大小）
[in]	size	长度（内部会向上圆整到页面大小）

- **返回值** 成功返回虚拟地址，失败时返回 -1
- **注意事项** 无
- **示例** 无

ms_vmm_iounmap()

- **描述** 截取虚拟地址到物理地址的映射
- **函数原型**

```
ms_err_t  ms_vmm_iounmap(ms_addr_t virt_addr);
```

- **参数**

输入/输出	参数	描述
[in]	virt_addr	虚拟地址

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_vmm_kmap_in()

- **描述** 判断指定的虚拟地址是否为内核固定映射的虚拟地址，内核固定映射是指不经过 MMU 转换的虚拟地址空间，内核固定映射的虚拟地址与物理地址一般是线性映射关系。
- **函数原型**

```
ms_bool_t ms_vmm_kmap_in(ms_addr_t virt_addr);
```

- **参数**

输入/输出	参数	描述
[in]	virt_addr	虚拟地址

- **返回值** MS-TRUE：虚拟地址是内核固定映射的虚拟地址，MS-FLASE：虚拟地址不是内核固定映射的虚拟地址
- **注意事项** 无
- **示例** 无

ms_vmm_kmap_v2p()

- **描述** 将内核固定映射的虚拟地址转换成物理地址
- **函数原型**

```
ms_addr_t ms_vmm_kmap_v2p(ms_addr_t virt_addr);
```

- **参数**

输入/输出	参数	描述
[in]	virt_addr	虚拟地址

- **返回值** 对应的物理地址
- **注意事项** 无
- **示例** 无

ms_vmm_kmap_p2v()

- **描述** 将内核固定映射的物理地址转换成虚拟地址
- **函数原型**

```
ms_addr_t ms_vmm_kmap_p2v(ms_addr_t phy_addr);
```

- **参数**

输入/输出	参数	描述
[in]	phy_addr	物理地址

- **返回值** 对应的虚拟地址
- **注意事项** 无
- **示例** 无

31 MS-RTOS 日志

> 本章将介绍 MS-RTOS 日志 LOG 的使用。

LOG 相关 API

下表展示了 LOG 相关的 API 在两个权限空间下是否可用：

API	用户空间	内核空间
ms_log	●	●
ms_log_set_level	●	●
ms_log_set_fd	●	●
ms_log_crash		●
MS_LOGM	●	●
MS_LOGA	●	●
MS_LOGC	●	●
MS_LOGE	●	●
MS_LOGW	●	●
MS_LOGN	●	●
MS_LOGI	●	●
MS_LOGD	●	●
_MS_LOGM	●	●
_MS_LOGA	●	●
_MS_LOGC	●	●
_MS_LOGE	●	●
_MS_LOGW	●	●
_MS_LOGN	●	●
_MS_LOGI	●	●
_MS_LOGD	●	●

ms_log()

- **描述** 记录一则日志

- **函数原型**

```
void ms_log(ms_log_level_t level, const char *fmt, ...) MS_LOG_ATTR;
```

- **参数**

输入/输出	参数	描述
[in]	level	日志级别
[in]	fmt	格式化字符串

日志级别为以下的宏：

宏	值	描述
MS_LOG_EMERG	0U	紧急事件消息，系统崩溃之前提示，表示系统不可用
MS_LOG_ALERT	1U	报告消息，表示必须立即采取措施
MS_LOG_CRIT	2U	临界条件，通常涉及严重的硬件或软件操作失败
MS_LOG_ERR	3U	错误条件
MS_LOG_WARNING	4U	警告条件，对可能出现问题的情况进行警告
MS_LOG_NOTICE	5U	正常但又重要的条件，用于提醒
MS_LOG_INFO	6U	提示信息
MS_LOG_DEBUG	7U	调试级别的消息

- **返回值** 无
- **示例** 无

日志宏

- **描述** MS-RTOS 定义了一些日志相关的宏，使用前，需要在源文件里定义 `MS_LOG_TAG` 字符串。
- **日志宏原型**

```
#define MS_LOGM(fmt, ...)   ms_log(MS_LOG_EMERG,   "<0>" MS_LOG_TAG fmt, ##__VA_ARGS__)
#define MS_LOGA(fmt, ...)   ms_log(MS_LOG_ALERT,   "<1>" MS_LOG_TAG fmt, ##__VA_ARGS__)
#define MS_LOGC(fmt, ...)   ms_log(MS_LOG_CRIT,    "<2>" MS_LOG_TAG fmt, ##__VA_ARGS__)
#define MS_LOGE(fmt, ...)   ms_log(MS_LOG_ERR,     "<3>" MS_LOG_TAG fmt, ##__VA_ARGS__)
#define MS_LOGW(fmt, ...)   ms_log(MS_LOG_WARNING, "<4>" MS_LOG_TAG fmt, ##__VA_ARGS__)
#define MS_LOGN(fmt, ...)   ms_log(MS_LOG_NOTICE,  "<5>" MS_LOG_TAG fmt, ##__VA_ARGS__)
#define MS_LOGI(fmt, ...)   ms_log(MS_LOG_INFO,    "<6>" MS_LOG_TAG fmt, ##__VA_ARGS__)
#define MS_LOGD(fmt, ...)   ms_log(MS_LOG_DEBUG,   "<7>" MS_LOG_TAG fmt, ##__VA_ARGS__)

#define _MS_LOGM(fmt, ...)  ms_log(MS_LOG_EMERG,   fmt, ##__VA_ARGS__)
```

```
#define _MS_LOGA(fmt, ...)  ms_log(MS_LOG_ALERT,   fmt, ##__VA_ARGS__)
#define _MS_LOGC(fmt, ...)  ms_log(MS_LOG_CRIT,    fmt, ##__VA_ARGS__)
#define _MS_LOGE(fmt, ...)  ms_log(MS_LOG_ERR,     fmt, ##__VA_ARGS__)
#define _MS_LOGW(fmt, ...)  ms_log(MS_LOG_WARNING, fmt, ##__VA_ARGS__)
#define _MS_LOGN(fmt, ...)  ms_log(MS_LOG_NOTICE,  fmt, ##__VA_ARGS__)
#define _MS_LOGI(fmt, ...)  ms_log(MS_LOG_INFO,    fmt, ##__VA_ARGS__)
#define _MS_LOGD(fmt, ...)  ms_log(MS_LOG_DEBUG,   fmt, ##__VA_ARGS__)
```

- **参数**

输入/输出	参数	描述
[in]	fmt	格式化字符串

- **返回值** 无
- **示例** 无

ms_log_set_level()

- **描述** 设置 `ms_log` 函数可记录的日志级别（默认为 `MS_LOG_DEBUG`）
- **函数原型**

```
ms_err_t ms_log_set_level(ms_log_level_t level);
```

- **参数**

输入/输出	参数	描述
[in]	level	日志级别

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例** 无

ms_log_set_fd()

- **描述** 设置日志的文件描述符（默认为标准输出文件）
- **函数原型**

```
ms_err_t ms_log_set_fd(int fd);
```

- **参数**

输入/输出	参数	描述
[in]	fd	日志的文件描述符

- **返回值** MS-RTOS 内核错误码
- **注意事项** 无
- **示例**

```
// define log tag
#define MS_LOG_TAG "[test]"

int main(int argc, char *argv[])
{
    // create log file
    int fd = ms_io_creat("/nor/app_log.txt", 0666);

    // set log file
    ms_log_set_fd(fd);

    // set log level
    ms_log_set_level(MS_LOG_DEBUG);

    // log info
    MS_LOGI("Please wait...");

    _MS_LOGI("Done!\n");

    // do some thing

    return 0;
}
```

ms_log_crash()

- **描述** 记录一则崩溃日志信息，会通过 `ms_bsp_log_write` 接口记录崩溃日志信息
- **函数原型**

```
void ms_log_crash(const char *fmt, ...);
```

- **参数**

输入/输出	参数	描述
[in]	fmt	格式化字符串

- **返回值** 无

- **注意事项** 该函数一般在 CPU 异常处理函数中调用，所以该函数实现时需要确保能在中断环境而非任务环境中断执行
- **示例** 无

32 MS-RTOS 板级支持包

> 本章将介绍 MS-RTOS 板级支持包 BSP 相关接口的使用。

MS-RTOS 移植介绍

当移植 MS-RTOS 到某一款芯片上，需要实现一个板级支持包 BSP，BSP 里面主要包括了启动代码、链接脚本、驱动程序和 MS-RTOS 依赖于硬件平台需要实现的一些函数，MS-RTOS 定义了这些函数的接口、功能规范，开发者需要实现这些接口方能使得 MS-RTOS 能够在该芯片上运行起来，但这些函数并非强制需要全部都实现（参考函数列表的依赖）。

BSP 相关函数

下表展示了 BSP 相关的函数的依赖：

函数	依赖
ms_bsp_reboot	无
ms_bsp_shutdown	无
ms_bsp_printk	无
ms_bsp_log_write	日志系统
ms_bsp_int_enable	中断管理
ms_bsp_int_disable	中断管理
ms_bsp_int_is_enable	中断管理
ms_bsp_device_name	Trace
ms_bsp_cpu_freq	Trace
ms_bsp_timestamp_freq	Trace
ms_bsp_int_trace_desc	Trace
ms_bsp_tick_less_param	Tick-less
ms_bsp_tick_less_sleep	Tick-less
ms_bsp_pm_set_run_mode	电源管理
ms_bsp_pm_notify	电源管理
ms_bsp_pm_sleep	电源管理
ms_bsp_pm_timer_start	电源管理
ms_bsp_pm_timer_stop	电源管理

函数	依赖
ms_bsp_pm_timer_elapsed	电源管理
ms_bsp_hm_notify	健康监控设备驱动
ms_bsp_mmu_pre_enable	带有 MMU 的 CPU
ms_bsp_mpu_pre_enable	带有 MPU 的 CPU

ms_bsp_reboot()

- **描述** 机器重启
- **函数原型**

```
void ms_bsp_reboot(void);
```

- **参数** 无
- **返回值** 无
- **注意事项** 无
- **示例** 无

ms_bsp_shutdown()

- **描述** 关机
- **函数原型**

```
void ms_bsp_shutdown(void);
```

- **参数** 无
- **返回值** 无
- **注意事项** 无
- **示例** 无

ms_bsp_printk()

- **描述** 打印一则内核消息
- **函数原型**

```
void ms_bsp_printk(const char *buf, ms_size_t len);
```

- **参数**

输入/输出	参数	描述
[in]	buf	需要打印的消息
[in]	len	需要打印的消息的长度

- **返回值** 无
- **注意事项** 无
- **示例** 无

ms_bsp_log_write()

- **描述** 记录一则 LOG 日志
- **函数原型**

```
void ms_bsp_log_write(const char *buf, ms_size_t len);
```

- **参数**

输入/输出	参数	描述
[in]	buf	需要记录的日志
[in]	len	需要记录的日志的长度

- **返回值** 无
- **注意事项** 无
- **示例** 无

ms_bsp_int_enable()

- **描述** 使能指定的中断
- **函数原型**

```
ms_err_t  ms_bsp_int_enable(ms_irq_t irq);
```

- **参数**

输入/输出	参数	描述
[in]	irq	中断号

- **返回值** MS-RTOS 内核错误码
- **注意事项** 中断管理使能时需要实现，BSP、驱动开发请使用中断安全版本函数 `ms_int_enable`。
- **示例** 无

ms_bsp_int_disable()

- **描述** 屏蔽指定的中断
- **函数原型**

```
ms_err_t  ms_bsp_int_disable(ms_irq_t irq);
```

- **参数**

输入/输出	参数	描述
[in]	irq	中断号

- **返回值** MS-RTOS 内核错误码
- **注意事项** 中断管理使能时需要实现，BSP、驱动开发请使用中断安全版本函数 `ms_int_disable`。
- **示例** 无

ms_bsp_int_is_enable()

- **描述** 判断指定的中断是否使能
- **函数原型**

```
ms_bool_t ms_bsp_int_is_enable(ms_irq_t irq);
```

- **参数**

输入/输出	参数	描述
[in]	irq	中断号

- **返回值** MS_TRUE：使能，MS_FALSE：被屏蔽
- **注意事项** 中断管理使能时需要实现，BSP、驱动开发请使用中断安全版本函数 `ms_int_is_enable`
- **示例** 无

ms_bsp_device_name()

- **描述** 获得设备名
- **函数原型**

```
const char *ms_bsp_device_name(void);
```

- **参数** 无
- **返回值** 设备名

- **注意事项** Trace 使能时需要实现
- **示例** 无

ms_bsp_cpu_freq()

- **描述** 获得 CPU 主频
- **函数原型**

```
ms_uint32_t ms_bsp_cpu_freq(void);
```

- **参数** 无
- **返回值** CPU 主频
- **注意事项** Trace 使能时需要实现
- **示例** 无

ms_bsp_timestamp_freq()

- **描述** 获得时间戳的频率
- **函数原型**

```
ms_uint32_t ms_bsp_timestamp_freq(void);
```

- **参数** 无
- **返回值** 时间戳的频率
- **注意事项** Trace 使能时需要实现
- **示例** 无

ms_bsp_int_trace_desc()

- **描述** 获得中断的 trace 描述信息
- **函数原型**

```
const char *ms_bsp_int_trace_desc(void);
```

- **参数** 无
- **返回值** 中断的 trace 描述信息
- **注意事项** Trace 使能时需要实现
- **示例** 无

ms_bsp_tick_less_param()

- **描述** 获得 tick-less 模式相关工作参数

- **函数原型**

```
void ms_bsp_tick_less_param(ms_tick_t *max_tick, ms_tick_t *min_tick,
                            ms_uint32_t *timer_cnt_per_tick,
                            ms_uint32_t *timer_stop_compensation);
```

- **参数**

输入/输出	参数	描述
[out]	max_tick	硬件定时器支持的最大 tick-less 时间
[out]	min_tick	至少休眠多少个 tick 时才进入 tick-less CPU 休眠模式
[out]	timer_cnt_per_tick	每个 tick 多少个硬件定时器的计数值
[out]	timer_stop_compensation	硬件定时器停止时的计数值补偿量

- **返回值** 无
- **注意事项** tick-less 模式使能时需要实现
- **示例** 无

ms_bsp_tick_less_sleep()

- **描述** 进入 tick-less CPU 休眠模式
- **函数原型**

```
void ms_bsp_tick_less_sleep(ms_tick_t expected_tick, ms_arch_sr_t sr);
```

- **参数**

输入/输出	参数	描述
[in]	expected_tick	期望 CPU 休眠多少个 tick
[in]	sr	CPU 进入休眠前需要调用 ms_arch_int_resume 函数恢复 CPU 中断状态的参数

- **返回值** 无
- **注意事项** tick-less 模式使能时需要实现
- **示例** 无

ms_bsp_pm_set_run_mode()

- **描述** 设置 CPU 运行模式

- **函数原型**

```
void ms_bsp_pm_set_run_mode(ms_pm_run_mode_t run_mode);
```

- **参数**

输入/输出	参数	描述
[in]	run_mode	CPU 运行模式

- **返回值** 无
- **注意事项** 电源管理使能时需要
- **示例** 无

ms_bsp_pm_notify()

- **描述** 响应电源管理的通知事件
- **函数原型**

```
void ms_bsp_pm_notify(ms_pm_sleep_mode_t sleep_mode, ms_pm_event_t event);
```

- **参数**

输入/输出	参数	描述
[in]	sleep_mode	CPU 休眠模式
[in]	event	电源管理的通知事件

- **返回值** 无
- **注意事项** 电源管理使能时需要
- **示例** 无

ms_bsp_pm_sleep()

- **描述** 让 CPU 进入指定的休眠模式
- **函数原型**

```
void ms_bsp_pm_sleep(ms_pm_sleep_mode_t sleep_mode, ms_arch_sr_t sr);
```

- **参数**

输入/输出	参数	描述
[in]	sleep_mode	需要进入的 CPU 休眠模式
[in]	sr	CPU 进入休眠前需要调用 ms_arch_int_resume 函数恢复 CPU 中断状态的参数

- **返回值** 无
- **注意事项** 电源管理使能时需要
- **示例** 无

ms_bsp_pm_timer_start()

- **描述** 启动低功耗定时器
- **函数原型**

```
void ms_bsp_pm_timer_start(ms_tick_t tick);
```

- **参数**

输入/输出	参数	描述
[in]	tick	低功耗定时器到期时间（tick 单位）

- **返回值** 无
- **注意事项** 电源管理使能时需要
- **示例** 无

ms_bsp_pm_timer_stop()

- **描述** 停止低功耗定时器
- **函数原型**

```
void ms_bsp_pm_timer_stop(void);
```

- **参数** 无
- **返回值** 无
- **注意事项** 电源管理使能时需要
- **示例** 无

ms_bsp_pm_timer_elapsed()

- **描述** 获得 CPU 休眠期间低功耗定时器流逝的嘀嗒数

- **函数原型**

```
ms_tick_t ms_bsp_pm_timer_elapsed(void);
```

- **参数** 无
- **返回值** CPU 休眠期间低功耗定时器流逝的嘀嗒数
- **注意事项** 电源管理使能时需要
- **示例** 无

ms_bsp_hm_notify()

- **描述** 响应健康监控的通知事件
- **函数原型**

```
void ms_bsp_hm_notify(ms_pid_t pid, ms_hm_handle_mode_t mode);
```

- **参数**

输入/输出	参数	描述
[in]	pid	健康异常的进程 ID
[in]	mode	健康异常处理模式

- **返回值** 无
- **注意事项** 健康监控设备驱动使能时需要
- **示例** 无

ms_bsp_mmu_pre_enable()

- **描述** MMU 使能前的处理（一般用于 IO 寄存器重映射）
- **函数原型**

```
void ms_bsp_mmu_pre_enable(void);
```

- **参数** 无
- **返回值** 无
- **注意事项** 无
- **示例** 无

ms_bsp_mpu_pre_enable()

- **描述** MPU 使能前的处理

- **函数原型**

```
void ms_bsp_mpu_pre_enable(void);
```

- **参数** 无
- **返回值** 无
- **注意事项** 无
- **示例** 无

33 libsddc

> 本章将介绍 libsddc 库的使用。

libsddc 使用

如果要使用 libsddc，应用程序需要链接 `libsddc.a` 静态库，即应用程序的 `Makefile` 文件 `xxx.mk` 的 `LOCAL_DEPEND_LIB` 需要添加 `-lsddc`，同时 `LOCAL_DEPEND_LIB_PATH` 需要添加 `-L"$(MSRTOS_BASE_PATH)/libsddc/$(OUTDIR)"`。为了能让应用程序的源文件找到 libsddc 的头文件，`LOCAL_INC_PATH` 需要添加 `-I"$(MSRTOS_BASE_PATH)/libsddc/src"`。

如果使能了数据加密通信，则 `LOCAL_DEPEND_LIB` 需要添加 `-lmbedtls`、`-lmbedx509`、`-lmbedcrypto`，`LOCAL_DEPEND_LIB_PATH` 需要添加 `-L"$(MSRTOS_BASE_PATH)/mbedtls/$(OUTDIR)"`。

我们推荐使用 cJSON 作为 JSON 的解析库，`LOCAL_DEPEND_LIB` 需要添加 `-lcjson`，`LOCAL_DEPEND_LIB_PATH` 需要添加 `-L"$(MSRTOS_BASE_PATH)/cJSON/$(OUTDIR)"`，`LOCAL_INC_PATH` 需要添加 `-I"$(MSRTOS_BASE_PATH)/cjson/src/cJSON"`。

详细可以参考 libsddc 示例程序的 `Makefile libsddc/sddc_examples.mk` 文件。

libsddc API

下表展示了 libsddc 相关的 API：

API	功能
sddc_create	创建 SDDC
sddc_set_uid	设置唯一 ID
sddc_set_token	设置加解密 token（设备密码）
sddc_set_invite_data	设置 INVITE 数据
sddc_set_report_data	设置 REPORT 数据
sddc_set_on_invite	设置收到 INVITE 处理函数
sddc_set_on_invite_end	设置 INVITE 完毕后处理函数
sddc_set_on_update	设置收到 UPDATE 处理函数
sddc_set_on_edgeros_lost	设置 EdgerOS 断开链接处理函数
sddc_set_on_message_lost	设置消息 lost 处理函数
sddc_set_on_message	设置收到 MESSAGE 处理函数
sddc_set_on_message_ack	设置收到 MESSAGE ACK 处理函数
sddc_set_on_timestamp	设置收到 TIMESTAMP ACK 处理函数

API	功能
sddc_run	运行 SDDC
sddc_send_message	向指定的 EdgerOS 发送 MESSAGE
sddc_broadcast_message	向所有链接的 EdgerOS 发送 MESSAGE
sddc_send_timestamp_request	向指定的 EdgerOS 发送 TIMESTAMP 请求
sddc_send_update	向指定的 EdgerOS 发送 UPDATE，用于节点 IP 改变时，通知 EdgerOS
sddc_broadcast_update	向所有链接的 EdgerOS 发送 UPDATE，用于节点 IP 改变时，通知 EdgerOS
sddc_connector_create	创建一个连接到 EdgerOS 的数据连接器
sddc_connector_destroy	销毁一个数据连接器
sddc_connector_fd	获得数据连接器的文件描述符
sddc_connector_put	发送数据到数据连接器
sddc_connector_get	从数据连接器接收数据

sddc_create()

- **描述** 创建 SDDC
- **函数原型**

```
sddc_t *sddc_create(uint16_t port);
```

- **参数**

输入/输出	参数	描述
[in]	port	UDP 端口

- **返回值** SDDC 对象指针
- **注意事项** 无
- **示例** 无

sddc_set_uid()

- **描述** 设置 SDDC 的唯一 ID
- **函数原型**


```
int sddc_set_uid(sddc_t *sddc, const uint8_t *mac_addr);
```

- **参数**

输入/输出	参数	描述
[in]	sddc	SDDC 对象指针
[in]	mac_addr	MAC 地址（6 个 BYTE 的数组）

- **返回值** 成功返回 0，失败返回 -1
- **注意事项** 无
- **示例** 无

sddc_set_token()

- **描述** 设置 SDDC 的加解密 token（设备密码）
- **函数原型**

```
int sddc_set_token(sddc_t *sddc, const char *token);
```

- **参数**

输入/输出	参数	描述
[in]	sddc	SDDC 对象指针
[in]	token	加解密 token

- **返回值** 成功返回 0，失败返回 -1
- **注意事项** 无
- **示例** 无

sddc_set_invite_data()

- **描述** 设置 SDDC 的 INVITE 数据
- **函数原型**

```
int sddc_set_invite_data(sddc_t *sddc, const char *invite_data, size_t len);
```

- **参数**

输入/输出	参数	描述
[in]	sddc	SDDC 对象指针

输入/输出	参数	描述
[in]	invite_data	INVITE 数据（JSON 格式）
[in]	len	INVITE 数据长度

- **返回值** 成功返回 0，失败返回 -1
- **注意事项** 无
- **示例**

sddc_set_report_data()

- **描述** 设置 SDDC 的 REPORT 数据
- **函数原型**

```
int sddc_set_report_data(sddc_t *sddc, const char *report_data, size_t len);
```

- **参数**

输入/输出	参数	描述
[in]	sddc	SDDC 对象指针
[in]	report_data	REPORT 数据（JSON 格式）
[in]	len	REPORT 数据长度

- **返回值** 成功返回 0，失败返回 -1
- **注意事项** 无
- **示例** 无

sddc_set_on_invite()

- **描述** 设置 SDDC 收到 INVITE 处理函数
- **函数原型**

```
int sddc_set_on_invite(sddc_t *sddc, sddc_on_invite_t on_invite);
```

- **参数**

输入/输出	参数	描述
[in]	sddc	SDDC 对象指针
[in]	on_invite	INVITE 处理函数

其中 `on_invite` 的类型如下：


```
typedef sddc_bool_t (*sddc_on_invite_t)(sddc_t *sddc, const uint8_t *uid, const char *invite_data, size_t len);
```

SDDC 回调 on_invite 函数时会传入 EdgerOS 的 UID、INVITE 数据和长度。 on_invite 函数返回一个 sddc_bool_t 布尔类型，返回 SDDC_TRUE 意味端设备接受 EdgerOS 的邀请，SDDC_FALSE 意味端设备拒绝 EdgerOS 的邀请。

- **返回值** 成功返回 0，失败返回 -1
- **注意事项** 无
- **示例** 无

sddc_set_on_invite_end()

- **描述** 设置 SDDC 的 INVITE 完毕后处理函数
- **函数原型**

```
int sddc_set_on_invite_end(sddc_t *sddc, sddc_on_invite_end_t on_invite_end);
```

- **参数**

输入/输出	参数	描述
[in]	sddc	SDDC 对象指针
[in]	on_invite_end	INVITE 完毕后处理函数

其中 on_invite_end 的类型如下：

```
typedef sddc_bool_t (*sddc_on_invite_end_t)(sddc_t *sddc, const uint8_t *uid);
```

SDDC 回调 on_invite_end 函数时会传入 EdgerOS 的 UID。 on_invite_end 函数一般用于向 EdgerOS 报告端设备的当前状态信息。

- **返回值** 成功返回 0，失败返回 -1
- **注意事项** 无
- **示例** 无

sddc_set_on_update()

- **描述** 设置 SDDC 收到 UPDATE 处理函数
- **函数原型**

```
int sddc_set_on_update(sddc_t *sddc, sddc_on_update_t on_update);
```

- **参数**

输入/输出	参数	描述
[in]	sddc	SDDC 对象指针
[in]	on_update	收到 UPDATE 处理函数

其中 on_update 的类型如下：

```
typedef sddc_bool_t (*sddc_on_update_t)(sddc_t *sddc, const uint8_t *uid, const char *update_data, size_t len);
```

SDDC 回调 on_update 函数时会传入 EdgerOS 的 UID、UPDATE 数据和长度。

on_update 函数返回一个 sddc_bool_t 布尔类型，返回 SDDC_TRUE 将会给 EdgerOS 发送回应，否则不发送回应。

- **返回值** 成功返回 0，失败返回 -1
- **注意事项** 无
- **示例** 无

sddc_set_on_edgeros_lost()

- **描述** 设置 EdgerOS 断开链接处理函数
- **函数原型**

```
int sddc_set_on_edgeros_lost(sddc_t *sddc, sddc_on_edgeros_lost_t on_edgeros_lost);
```

- **参数**

输入/输出	参数	描述
[in]	sddc	SDDC 对象指针
[in]	on_edgeros_lost	EdgerOS 断开链接处理函数

其中 on_edgeros_lost 的类型如下：

```
typedef void (*sddc_on_edgeros_lost_t)(sddc_t *sddc, const uint8_t *uid);
```

SDDC 回调 on_edgeros_lost 函数时会传入 EdgerOS 的 UID。

- **返回值** 成功返回 0，失败返回 -1
- **注意事项** 无
- **示例** 无

sddc_set_on_message_lost()

- **描述** 设置 SDDC 的消息 lost 处理函数
- **函数原型**

```
int sddc_set_on_message_lost(sddc_t *sddc, sddc_on_message_lost_t on_message_lost);
```

- **参数**

输入/输出	参数	描述
[in]	sddc	SDDC 对象指针
[in]	on_message_lost	消息 lost 处理函数

其中 on_message_lost 的类型如下：

```
typedef void (*sddc_on_message_lost_t)(sddc_t *sddc, const uint8_t *uid, uint16_t seqno);
```

SDDC 回调 on_message_lost 函数时会传入 EdgerOS 的 UID 和 lost 掉的消息序列号。

- **返回值** 成功返回 0，失败返回 -1
- **注意事项** 无
- **示例** 无

sddc_set_on_message()

- **描述** 设置 SDDC 收到 MESSAGE 处理函数
- **函数原型**

```
int sddc_set_on_message(sddc_t *sddc, sddc_on_message_t on_message);
```

- **参数**

输入/输出	参数	描述
[in]	sddc	SDDC 对象指针
[in]	on_message	收到 MESSAGE 处理函数

其中 on_message 的类型如下：

```
typedef sddc_bool_t (*sddc_on_message_t)(sddc_t *sddc, const uint8_t *uid, const char *messag
e, size_t len);
```


SDDC 回调 on_message 函数时会传入 EdgerOS 的 UID 和消息数据及长度。如果收到的是一个需要 ACK 的消息，并且 on_message 函数返回 SDDC_TRUE，SDDC 将会给 EdgerOS 发送 ACK，否则不发送 ACK。

消息数据是 JSON 格式字符串，如果它里面带有 EdgerOS App Id（即带有类型为 Number 的 appid 属性），则说明此消息为 App 会话相关消息，设备端在回复这类消息时，应该在回复里加上 EdgerOS App Id（即加入类型为 Number 的 appid 属性）。 同时如果设备端需要给指定的 EdgerOS App 发送消息，也应该在消息里加上 EdgerOS App Id。

- **返回值** 成功返回 0，失败返回 -1
- **注意事项** 无
- **示例** 无

sddc_set_on_message_ack()

- **描述** 设置 SDDC 收到 MESSAGE ACK 处理函数
- **函数原型**

```
int sddc_set_on_message_ack(sddc_t *sddc, sddc_on_message_ack_t on_message_ack);
```

- **参数**

输入/输出	参数	描述
[in]	sddc	SDDC 对象指针
[in]	on_message_ack	收到 MESSAGE ACK 处理函数

其中 on_message_ack 的类型如下：

```
typedef void (*sddc_on_message_ack_t)(sddc_t *sddc, const uint8_t *uid, uint16_t seqno);
```

SDDC 回调 on_message_ack 函数时会传入 EdgerOS 的 UID 和消息的序列号。

- **返回值** 成功返回 0，失败返回 -1
- **注意事项** 无
- **示例** 无

sddc_set_on_timestamp()

- **描述** 设置 SDDC 收到 TIMESTAMP ACK 处理函数
- **函数原型**

```
int sddc_set_on_timestamp(sddc_t *sddc, sddc_on_timestamp_t on_timestamp);
```

- **参数**

输入/输出	参数	描述
[in]	sddc	SDDC 对象指针
[in]	timestamp	收到 TIMESTAMP ACK 处理函数

其中 on_timestamp 的类型如下：

```
typedef void (*sddc_on_timestamp_t)(sddc_t *sddc, const uint8_t *uid, const char *message, size_t len);
```

SDDC 回调 on_timestamp 函数时会传入 EdgerOS 的 UID 和消息数据及长度。消息数据内容为如下格式的 JSON 字符串：

```
{
  "timestamp":    {
    "utc":  1652673207934,
    "tz":   -28800000
  }
}
```

- **返回值** 成功返回 0，失败返回 -1
- **注意事项** 无
- **示例** 无

sddc_run()

- **描述** 运行 SDDC
- **函数原型**

```
int sddc_run(sddc_t *sddc);
```

- **参数**

输入/输出	参数	描述
[in]	sddc	SDDC 对象指针

- **返回值** 成功返回 0，失败返回 -1
- **注意事项** 无
- **示例** 无

sddc_send_message()

- **描述** 发送 MESSAGE
- **函数原型**

```
int sddc_send_message(sddc_t *sddc, const uint8_t *uid,
                      const void *payload, size_t payload_len,
                      uint8_t retries, sddc_bool_t urgent,
                      uint16_t *seqno);
```

- **参数**

输入/输出	参数	描述
[in]	sddc	SDDC 对象指针
[in]	uid	EdgerOS 的唯一 ID
[in]	payload	消息数据
[in]	payload_len	消息数据长度
[in]	retries	消息重发次数
[in]	urgent	是否发送紧急消息
[out]	seqno	消息序列号

- **返回值** 成功返回 0，失败返回 -1
- **注意事项** 无
- **示例** 无

sddc_broadcast_message()

- **描述** 向所有链接的 EdgerOS 发送 MESSAGE
- **函数原型**

```
int sddc_broadcast_message(sddc_t *sddc,
                           const void *payload, size_t payload_len,
                           uint8_t retries, sddc_bool_t urgent,
                           uint16_t *seqno);
```

- **参数**

输入/输出	参数	描述
[in]	sddc	SDDC 对象指针
[in]	uid	EdgerOS 的唯一 ID

输入/输出	参数	描述
[in]	payload	消息数据
[in]	payload_len	消息数据长度
[in]	retries	消息重发次数
[in]	urgent	是否发送紧急消息
[out]	seqno	消息序列号数组

- **返回值** 成功返回 0，失败返回 -1
- **注意事项** 无
- **示例** 无

sddc_send_timestamp_request()

- **描述** 向链接的 EdgerOS 发送 TIMESTAMP 请求
- **函数原型**

```
int sddc_send_timestamp_request(sddc_t *sddc, const uint8_t *uid);
```

- **参数**

输入/输出	参数	描述
[in]	sddc	SDDC 对象指针
[in]	uid	EdgerOS 的唯一 ID，如果为 NULL，则使用已经链接的 EdgerOS 的唯一 ID

- **返回值** 成功返回 0，失败返回 -1
- **注意事项** 无
- **示例** 无

sddc_send_update()

- **描述** 向链接的 EdgerOS 发送 UPDATE 请求，用于节点 IP 改变时，通知 EdgerOS
- **函数原型**

```
int sddc_send_update(sddc_t *sddc, const uint8_t *uid);
```

- **参数**

输入/输出	参数	描述
[in]	sddc	SDDC 对象指针

输入/输出	参数	描述
		EdgerOS 的唯一 ID

- **返回值** 成功返回 0，失败返回 -1
- **注意事项** 无
- **示例** 无

sddc_broadcast_update()

- **描述** 向所有链接的 EdgerOS 发送 UPDATE 请求，用于节点 IP 改变时，通知 EdgerOS
- **函数原型**

```
int sddc_broadcast_update(sddc_t *sddc);
```

- **参数**

输入/输出	参数	描述
[in]	sddc	SDDC 对象指针

- **返回值** 成功返回 0，失败返回 -1
- **注意事项** 无
- **示例** 无

sddc_connector_create()

- **描述** 创建一个连接到指定的 EdgerOS 的数据连接器
- **函数原型**

```
sddc_connector_t *sddc_connector_create(sddc_t *sddc, const uint8_t *uid, uint16_t port, cons
t char *token, sddc_bool_t get_mode);
```

- **参数**

输入/输出	参数	描述
[in]	sddc	SDDC 对象指针
[in]	uid	EdgerOS 的唯一 ID
[in]	port	端口
[in]	token	数据加解密 token
[in]	get_mode	是否为获得数据模式

- **返回值** 成功返回 SDDC 数据连接器对象指针，失败返回 NULL

- **注意事项** 无
- **示例** 无

sddc_connector_destroy()

- **描述** 销毁一个数据连接器
- **函数原型**

```
int sddc_connector_destroy(sddc_connector_t *connector);
```

- **参数**

输入/输出	参数	描述
[in]	connector	SDDC 数据连接器对象指针

- **返回值** 成功返回 0，失败返回 -1
- **注意事项** 无
- **示例** 无

sddc_connector_fd()

- **描述** 获得指定数据连接器的通信文件描述符
- **函数原型**

```
int sddc_connector_fd(sddc_connector_t *connector);
```

- **参数**

输入/输出	参数	描述
[in]	connector	SDDC 数据连接器对象指针

- **返回值** 成功返回 通信文件描述符，失败返回 -1
- **注意事项** 无
- **示例** 无

sddc_connector_put()

- **描述** 发送数据到数据连接器，连接的 EdgerOS 将收到数据
- **函数原型**

```
int sddc_connector_put(sddc_connector_t *connector, const void *data, size_t len, sddc_bool_t
finish);
```


- **参数**

输入/输出	参数	描述
[in]	connector	SDDC 数据连接器对象指针
[in]	data	需要发送的数据缓冲区指针（finish 时可以为 NULL）
[in]	len	需要发送的数据长度（finish 时可以为 0）
[in]	finish	是否结束数据传输

- **返回值** 成功返回 0，失败返回 -1
- **注意事项** 无
- **示例** 无

sddc_connector_get()

- **描述** 从数据连接器中接收数据
- **函数原型**

```
ssize_t sddc_connector_get(sddc_connector_t *connector, void **data, sddc_bool_t *finish);
```

- **参数**

输入/输出	参数	描述
[in]	connector	SDDC 数据连接器对象指针
[out]	data	接收到的数据缓冲区指针
[out]	finish	EdgerOS 是否结束了数据传输

- **返回值** 成功返回 接收到的数据长度为 0，失败返回 -1
- **注意事项** 无
- **示例** 无

libsddc 示例

```
#include <ms_rtos.h>
#include "sddc.h"
#include "cJSON.h"

/*
 * handle MESSAGE
 */
static sddc_bool_t iot_pi_on_message(sddc_t *sddc, const uint8_t *uid, const char *message, si
ze_t len)
```

```
{
    cJSON *root = cJSON_Parse(message);
    sddc_return_value_if_fail(root, SDDC_TRUE);

    /*
     * Parse here
     */

    char *str = cJSON_Print(root);
    sddc_goto_error_if_fail(str);

    sddc_printf("iot_pi_on_message: %s\n", str);
    cJSON_free(str);

error:
    cJSON_Delete(root);

    return SDDC_TRUE;
}

/*
 * handle MESSAGE ACK
 */
static void iot_pi_on_message_ack(sddc_t *sddc, const uint8_t *uid, uint16_t seqno)
{
}

/*
 * handle MESSAGE lost
 */
static void iot_pi_on_message_lost(sddc_t *sddc, const uint8_t *uid, uint16_t seqno)
{
}

/*
 * handle EdgerOS lost
 */
static void iot_pi_on_edgeros_lost(sddc_t *sddc, const uint8_t *uid)
{
}

/*
 * handle UPDATE
 */
static sddc_bool_t iot_pi_on_update(sddc_t *sddc, const uint8_t *uid, const char *udpate_dat
a, size_t len)
{
    cJSON *root = cJSON_Parse(udpate_data);
    char *str;

    sddc_return_value_if_fail(root, SDDC_FALSE);

    /*
     * Parse here
```



```
     */

    str = cJSON_Print(root);
    sddc_goto_error_if_fail(str);

    sddc_printf("iot_pi_on_update: %s\n", str);
    cJSON_free(str);

    cJSON_Delete(root);

    return SDDC_TRUE;

error:
    cJSON_Delete(root);

    return SDDC_FALSE;
}

/*
 * handle INVITE
 */
static sddc_bool_t iot_pi_on_invite(sddc_t *sddc, const uint8_t *uid, const char *invite_dat
a, size_t len)
{
    cJSON *root = cJSON_Parse(invite_data);
    char *str;

    sddc_return_value_if_fail(root, SDDC_FALSE);

    /*
     * Parse here
     */

    str = cJSON_Print(root);
    sddc_goto_error_if_fail(str);

    sddc_printf("iot_pi_on_invite: %s\n", str);
    cJSON_free(str);

    cJSON_Delete(root);

    return SDDC_TRUE;

error:
    cJSON_Delete(root);

    return SDDC_FALSE;
}

/*
 * handle the end of INVITE
 */
static sddc_bool_t iot_pi_on_invite_end(sddc_t *sddc, const uint8_t *uid)
{
```



```
    return SDDC_TRUE;
}

/*
 * Handle TIMESTAMP ack
 */
static void iot_pi_on_timestamp(sddc_t *sddc, const uint8_t *uid, const char *message, ms_size
_t len)
{
    cJSON *root = cJSON_Parse(message);
    cJSON *timestamp, *utc, *tz;
    char *str;

    sddc_return_if_fail(root);

    str = cJSON_Print(root);
    sddc_goto_error_if_fail(str);

    sddc_printf("iot_pi_on_timestamp: %s\n", str);
    cJSON_free(str);

    timestamp = cJSON_GetObjectItem(root, "timestamp");
    if (cJSON_IsObject(timestamp)) {
        utc = cJSON_GetObjectItem(timestamp, "utc");
        if (cJSON_IsNumber(utc)) {

        }

        tz = cJSON_GetObjectItem(timestamp, "tz");
        if (cJSON_IsNumber(tz)) {

        }
    }

error:
    cJSON_Delete(root);
}

/*
 * Create REPORT data
 */
static char *iot_pi_report_data_create(void)
{
    cJSON *root;
    cJSON *report;
    char *str;

    root = cJSON_CreateObject();
    sddc_return_value_if_fail(root, NULL);

    report = cJSON_CreateObject();
    sddc_return_value_if_fail(report, NULL);

    cJSON_AddItemToObject(root, "report", report);
```



```
    cJSON_AddStringToObject(report, "name",   "IoT Pi");
    cJSON_AddStringToObject(report, "type",   "device");
    cJSON_AddBoolToObject(report,   "excl",   SDDC_FALSE);
    cJSON_AddStringToObject(report, "desc",   "翼辉 IoT Pi");
    cJSON_AddStringToObject(report, "model",  "1");
    cJSON_AddStringToObject(report, "vendor", "ACOINFO");

    /*
     * Add extension here
     */

    str = cJSON_Print(root);
    sddc_return_value_if_fail(str, NULL);

    sddc_printf("REPORT DATA: %s\n", str);

    cJSON_Delete(root);

    return str;
}

/*
 * Create INVITE data
 */
static char *iot_pi_invite_data_create(void)
{
    cJSON *root;
    cJSON *report;
    char *str;

    root = cJSON_CreateObject();
    sddc_return_value_if_fail(root, NULL);

    report = cJSON_CreateObject();
    sddc_return_value_if_fail(report, NULL);

    cJSON_AddItemToObject(root, "report", report);
    cJSON_AddStringToObject(report, "name",   "IoT Pi");
    cJSON_AddStringToObject(report, "type",   "device");
    cJSON_AddBoolToObject(report,   "excl",   SDDC_FALSE);
    cJSON_AddStringToObject(report, "desc",   "翼辉 IoT Pi");
    cJSON_AddStringToObject(report, "model",  "1");
    cJSON_AddStringToObject(report, "vendor", "ACOINFO");

    /*
     * Add extension here
     */

    str = cJSON_Print(root);
    sddc_return_value_if_fail(str, NULL);

    sddc_printf("INVITE DATA: %s\n", str);

    cJSON_Delete(root);
```



```
    return str;
}

/*
 * IoT Pi time thread
 */
static void iot_pi_time_thread(ms_ptr_t arg)
{
    sddc_t *sddc = arg;

    while (1) {
        ms_thread_sleep_s(5);

        sddc_send_timestamp_request(sddc, NULL);
    }
}

int main(int argc, char *argv[])
{
    struct ifreq ifreq;
    int sockfd;
    struct sockaddr_in *psockaddrin = (struct sockaddr_in *)&(ifreq.ifr_addr);
    sddc_t *sddc;
    char *data;
    int ret;

    /*
     * Set network implement
     */
#ifdef SDDC_CFG_NET_IMPL
    ret = ms_net_set_impl(SDDC_CFG_NET_IMPL);
    sddc_return_value_if_fail(ret == MS_ERR_NONE, -1);
#endif

    /*
     * Create SDDC
     */
    sddc = sddc_create(SDDC_CFG_PORT);
    sddc_return_value_if_fail(sddc, -1);

    /*
     * Set call backs
     */
    sddc_set_on_message(sddc, iot_pi_on_message);
    sddc_set_on_message_ack(sddc, iot_pi_on_message_ack);
    sddc_set_on_message_lost(sddc, iot_pi_on_message_lost);
    sddc_set_on_invite(sddc, iot_pi_on_invite);
    sddc_set_on_invite_end(sddc, iot_pi_on_invite_end);
    sddc_set_on_update(sddc, iot_pi_on_update);
    sddc_set_on_edgeros_lost(sddc, iot_pi_on_edgeros_lost);
    sddc_set_on_timestamp(sddc, iot_pi_on_timestamp);

    /*
```



```
     * Set token
     */
#if SDDC_CFG_SECURITY_EN > 0
    ret = sddc_set_token(sddc, "1234567890");
    sddc_return_value_if_fail(ret == 0, -1);
#endif

    /*
     * Set report data
     */
    data = iot_pi_report_data_create();
    sddc_return_value_if_fail(data, -1);
    sddc_set_report_data(sddc, data, strlen(data));

    /*
     * Set invite data
     */
    data = iot_pi_invite_data_create();
    sddc_return_value_if_fail(data, -1);
    sddc_set_invite_data(sddc, data, strlen(data));

    /*
     * Get mac address
     */
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    sddc_return_value_if_fail(sockfd >= 0, -1);

    ioctl(sockfd, SIOCGIFHWADDR, &ifreq);

    sddc_printf("MAC addr: %02x:%02x:%02x:%02x:%02x:%02x\n",
                (ms_uint8_t)ifreq.ifr_hwaddr.sa_data[0],
                (ms_uint8_t)ifreq.ifr_hwaddr.sa_data[1],
                (ms_uint8_t)ifreq.ifr_hwaddr.sa_data[2],
                (ms_uint8_t)ifreq.ifr_hwaddr.sa_data[3],
                (ms_uint8_t)ifreq.ifr_hwaddr.sa_data[4],
                (ms_uint8_t)ifreq.ifr_hwaddr.sa_data[5]);

    /*
     * Set uid
     */
    sddc_set_uid(sddc, (const ms_uint8_t *)ifreq.ifr_hwaddr.sa_data);

    /*
     * Get and print ip address
     */
    if (ioctl(sockfd, SIOCGIFADDR, &ifreq) == 0) {
        char ip[sizeof("255.255.255.255")];

        inet_ntoa_r(psockaddrin->sin_addr, ip, sizeof(ip));

        sddc_printf("IP addr: %s\n", ip);
    } else {
        sddc_printf("Failed to get IP address, Wi-Fi AP not online!\n");
    }
```



```
    close(sockfd);

    /*
     * Create time thread
     */
    ret = ms_thread_create("t_time",
                           iot_pi_time_thread,
                           sddc,
                           2048U,
                           30U,
                           70U,
                           MS_THREAD_OPT_USER | MS_THREAD_OPT_REENT_EN,
                           MS_NULL);
    sddc_return_value_if_fail(ret == MS_ERR_NONE, -1);

    /*
     * SDDC run
     */
    while (1) {
        sddc_printf("SDDC running...\n");

        sddc_run(sddc);

        sddc_printf("SDDC quit!\n");
    }

    /*
     * Destroy SDDC
     */
    sddc_destroy(sddc);

    return 0;
}
```


