# 软件与微电子学院
# 集成电路与智能系统系

## 《嵌入式与物联网操作系统》

# 代码分析报告

姓　名：冯轩

学　号：2001210240

日　期：2021/5/24

# 一、 消息队列数据结构

消息队列结构体如下图 1 所示：

```
struct  os_q {

#if (OS_OBJ_TYPE_REQ == DEF_ENABLED)
    OS_OBJ_TYPE           Type;
#endif
#if (OS_CFG_DBG_EN == DEF_ENABLED)
    CPU_CHAR             *NamePtr;
#endif
    OS_PEND_LIST          PendList;
#if (OS_CFG_DBG_EN == DEF_ENABLED)
    OS_Q                 *DbgPrevPtr;
    OS_Q                 *DbgNextPtr;
    CPU_CHAR             *DbgNamePtr;
#endif

    OS_MSG_Q              MsgQ;
};
```

图 1 结构体

- Type，表明消息队列的类型，一般设置为 OS_OBJ_TYPE_Q
- 消息队列的名字
- PendList 用来记录正在这个消息队列上等待的任务
- 与调试有关
- MsgQ 又是一个结构体类型，如下图 2 所示：

```
struct  os_msg_q {                                          /* OS_MS
    OS_MSG              *InPtr;                             /* Point
    OS_MSG              *OutPtr;                            /* Point
    OS_MSG_QTY           NbrEntriesSize;                    /* Maxim
    OS_MSG_QTY           NbrEntries;                        /* Curre
#if (OS_CFG_DBG_EN == DEF_ENABLED)
    OS_MSG_QTY           NbrEntriesMax;                     /* Peak
#endif
#if (defined(OS_CFG_TRACE_EN) && (OS_CFG_TRACE_EN == DEF_ENABLED))
    CPU_INT16U           MsgQID;                            /* Uniqu
#endif
};
```

图 2 结构体

- *InPtr：指向下一个入队的消息的指针
- *OutPtr：指向即将出队的消息的指针
- 队列最大容量
- 当前队列内消息个数
- 消息数量峰值

## 二、 OSQCreate

1. 功能：创建一个消息队列

2. 所需参数如下图 3 所示：

```
void  OSQCreate (OS_Q         *p_q,
                 CPU_CHAR     *p_name,
                 OS_MSG_QTY    max_qty,
                 OS_ERR       *p_err)
```

图 3 OSQCreate

- *p_q：指向一个消息队列
- *p_name：消息队列的名字
- max_qty：消息队列的长度，必须大于 0
- *p_err：错误码

3. 源码分析
   - 设置类型和名字，如下图 4 所示：

```
#if (OS_OBJ_TYPE_REQ == DEF_ENABLED)
    p_q->Type    = OS_OBJ_TYPE_Q;
#endif
#if (OS_CFG_DBG_EN == DEF_ENABLED)
    p_q->NamePtr = p_name;
#else
    (void)p_name;
#endif
```

图 4 设置类型和名字

   - 初始化队列和等待队列，如下图 5 所示：

```
OS_MsgQInit(&p_q->MsgQ,
            max_qty);
OS_PendListInit(&p_q->PendList);
```

图 5 初始化

## 三、 OSQPend

1. 功能：等待消息队列

2. 所需参数如下图 6 所示：

```
void  *OSQPend (OS_Q         *p_q,
                OS_TICK       timeout,
                OS_OPT        opt,
                OS_MSG_SIZE  *p_msg_size,
                CPU_TS       *p_ts,
                OS_ERR       *p_err)
{
```

图 6 参数

- *p_q：指向一个消息队列
- timeout：等待消息的超时时间
- opt：选择是否使用阻塞模式
- *p_msg_size：接收到的消息的长度
- *p_ts：指向一个时间戳，表示什么时候接收到消息
- *p_err：错误码

3. 源码分析
   - 检查是否有消息在指定队列中，如果有则返回此消息，如下图 7 所示：

```
p_void = OS_MsgQGet(&p_q->MsgQ,
                    p_msg_size,
                    p_ts,
                    p_err);
if (*p_err == OS_ERR_NONE) {
    OS_TRACE_Q_PEND(p_q);
    CPU_CRITICAL_EXIT();
    OS_TRACE_Q_PEND_EXIT(OS_ERR_NONE);
    return (p_void);
}
```

图 7 返回消息

   - 阻塞在消息队列中 pending 的任务，阻塞之后，寻找下一个拥有最高优先级的任务运行，如下图 8 所示：

```
OS_Pend((OS_PEND_OBJ *)((void *)p_q),
        OS_TASK_PEND_ON_Q,
        timeout);
CPU_CRITICAL_EXIT();
OS_TRACE_Q_PEND_BLOCK(p_q);
OSSched();
```

图 8 阻塞

   - 通过一个 switch 语句，对 post 的信息进行分类处理，如下图 9 所示：

```
switch (OSTCBCurPtr->PendStatus) { ... }
```

图 9 分类

# 四、 OSQPost

1. 功能：向消息队列发送一条消息

2. 参数如下图 10 所示：

```
void  OSQPost (OS_Q         *p_q,
               void         *p_void,
               OS_MSG_SIZE   msg_size,
               OS_OPT        opt,
               OS_ERR       *p_err)
{
```

图 10 参数

- *p_q：指向一个消息队列
- *p_void：指向实际发送的内容，void 类型
- msg_size：设定消息的大小，单位为字节数
- opt：用来选择消息发送操作的类型，FIFO or LIFO
- *p_err：错误码

3. 源码分析
   - 选择 FIFO 模式还是 LIFO 模式，如下图 11 所示：

```
if ((opt & OS_OPT_POST_LIFO) == 0u) {
    post_type = OS_OPT_POST_FIFO;
} else {
    post_type = OS_OPT_POST_LIFO;
}
```

图 11 FIFO or LIFO

   - 放置消息进消息队列吗，如下图 12 所示：

```
OS_MsgQPut(&p_q->MsgQ,
           p_void,
           msg_size,
           post_type,
           ts,
           p_err);
```

图 12 放置消息

   - 发送消息，并判断是否给所有等待的任务全部发送，如果不是，直接退出 while 循环，即只执行一次，如下图 13 所示：

```
while (p_tcb != (OS_TCB *)0) {
    p_tcb_next = p_tcb->PendNextPtr;
    OS_Post((OS_PEND_OBJ *)((void *)p_q),
            p_tcb,
            p_void,
            msg_size,
            ts);
    if ((opt & OS_OPT_POST_ALL) == 0u)  {
        break;
    }
    p_tcb = p_tcb_next;
}
```

图 13 发送消息

## 五、 OSMemCreate

1. 功能：创建一个固定大小的内存分区，由 ucos 管理

2. 参数如下图 14 所示：

```
void  OSMemCreate (OS_MEM       *p_mem,
                   CPU_CHAR     *p_name,
                   void         *p_addr,
                   OS_MEM_QTY    n_blks,
                   OS_MEM_SIZE   blk_size,
                   OS_ERR       *p_err)
{
```

图 14 参数

- *p_mem：指向存储区控制块地址
- *p_name：存储区名字
- *p_addr：存储区所有存储空间基地址
- n_blks：存储区中存储块个数
- blk_size：存储块大小
- *p_err：错误码

3. 源码分析
   - 创建空闲内存块之间的连接，如下图 15 所示：

```
p_link = (void **)p_addr;
p_blk  = (CPU_INT08U *)p_addr;
loops  = n_blks - 1u;
for (i = 0u; i < loops; i++) {
    p_blk += blk_size;
   *p_link = (void  *)p_blk;
    p_link = (void **)(void *)p_blk;
}
*p_link             = (void *)0;
```

图 15 创建连接

- 设置 p_mem 数据结构中的一些字段，如下图 16 所示：

```
#if (OS_OBJ_TYPE_REQ == DEF_ENABLED)
    p_mem->Type        = OS_OBJ_TYPE_MEM;
#endif
#if (OS_CFG_DBG_EN == DEF_ENABLED)
    p_mem->NamePtr     = p_name;
#else
    (void)p_name;
#endif
    p_mem->AddrPtr     = p_addr;
    p_mem->FreeListPtr = p_addr;
    p_mem->NbrFree     = n_blks;
    p_mem->NbrMax      = n_blks;
    p_mem->BlkSize     = blk_size;
```

图 16 设置数据结构

## 六、 OSMemGet

1. 功能：获取存储块

2. 参数如下图 17 所示：

```
void  *OSMemGet (OS_MEM  *p_mem,
                 OS_ERR  *p_err)
{
```

图 17 参数

- *p_mem：要使用的存储区
- *p_err：错误码

3. 源码分析
   - 检查是否存在空闲内存区，如果没有，则返回，如下图 18 所示：

```
if (p_mem->NbrFree == 0u) {
    CPU_CRITICAL_EXIT();
    OS_TRACE_MEM_GET_FAILED(p_mem);
    OS_TRACE_MEM_GET_EXIT(OS_ERR_MEM_NO_FREE_BLKS);
   *p_err = OS_ERR_MEM_NO_FREE_BLKS;
    return ((void *)0);
}
```

图 18 检查是否存在空闲内存区

- 如果存在空闲内存区，则指向下一个空闲的内存区，并把空闲内存区数量减一，如下图 19 所示：

```
 p_blk              = p_mem->FreeListPtr;
 p_mem->FreeListPtr = *(void **)p_blk;
 p_mem->NbrFree--;
 CPU_CRITICAL_EXIT();
 OS_TRACE_MEM_GET(p_mem);
 OS_TRACE_MEM_GET_EXIT(OS_ERR_NONE);
*p_err = OS_ERR_NONE;
 return (p_blk);
```

图 19 处理有空闲内存区的情况

## 七、 OSMemPut

1. 功能：释放存储块

2. 参数如下图 20 所示：

```
void  OSMemPut (OS_MEM  *p_mem,
                void    *p_blk,
                OS_ERR  *p_err)
{
```

图 20 参数

- *p_mem：要释放的存储区
- *p_blk：要释放的地址空间指针
- *p_err：错误码

3. 源码分析
   - 确保存储块没有全部被回收，如下图 21 所示：

```
if (p_mem->NbrFree >= p_mem->NbrMax) {
    CPU_CRITICAL_EXIT();
    OS_TRACE_MEM_PUT_FAILED(p_mem);
    OS_TRACE_MEM_PUT_EXIT(OS_ERR_MEM_FULL);
   *p_err = OS_ERR_MEM_FULL;
    return;
}
```

图 21 检查是否存在空闲内存区

- 把释放的存储块插入空闲存储块列表，然后将空闲存储块个数加一，如下图 22 所示：

```
 *(void **)p_blk     = p_mem->FreeListPtr;
 p_mem->FreeListPtr = p_blk;
 p_mem->NbrFree++;
 CPU_CRITICAL_EXIT();
 OS_TRACE_MEM_PUT(p_mem);
 OS_TRACE_MEM_PUT_EXIT(OS_ERR_NONE);
*p_err              = OS_ERR_NONE;
```

图 22 处理有空闲内存区的情况