SL_RTE 用户手册

作者：单雷 日期：2019/01/10 RTE_VER：3.3.0109
项目地址：https://github.com/sudashannon/SL_RTE

0 总体介绍

0.0 版本号说明

程序版本变动历史参见 doc\版本历史记录.txt。

从本文档建立起，从前使用的版本号全部作废，但保留版本历史纪录。

版本号于 APP_CONFIG 中定义，包含三部分，示例：RTE_VER：0.0.0615。其中，第一部分为大版本版本号；第二部分为小版本版本号；第三部分为版本发布日期。

0.1 SL_RTE 介绍

SL_RTE 是多个项目开发实践中积累下来的一些代码的整合，主要包含七个部分：APP 层，即脱离于硬件平台的部分；BSP 层，即适配不同硬件平台的部分；Board 层，即针对具体的板级的支持包；Utils 层，即第三方库；Port 层，移植 RTE 需要变动的文件；GUI 层，以 lvgl 为内核的 GUI；MV 层，以 openmv 为内核的机器视觉模块。

使用 SL_RTE，可以大大减少嵌入式开发尤其是 STM32 系列开发的时间。

0.2 第三方库与其他

目前使用或参考的第三方库包括：

http://software-dl.ti.com/tiva-c/SW-TM4C/latest/index_FDS.html
https://github.com/littlevgl/lvgl
https://github.com/openmv/openmv
https://github.com/micropython/micropython
https://www.fourmilab.ch/
https://github.com/MaJerle
https://github.com/rxi/vec
869119842@qq.com 唐童鞋 《C 语言最优化状态机规范》
https://github.com/armink/EasyFlash
https://github.com/ARM-software/CMSIS_5
http://blog.chinaunix.net/uid-22915173-id-2185545.html

需要注意的是，本 RTE 需要配合 KEIL5 以上版本使用。

1 内存管理

1.0 说明

本 RTE 自带的内存管理原始为 lvgl 中的内存管理模块，为了配合多块 RAM 的 MCU，RTE 对其进行了改写，使其适配于多块片内或片外 RAM。

有关动态内存分配的原理本手册不加赘述。

1.1 使用方法

当需要使用一块 RAM 区域以动态管理时，首先需要在 RTE_MEM.h 中加以声明：

```
18      //-------------------------------------------------------------------
19      // MEM枚举
20      //-------------------------------------------------------------------
21      typedef enum
22      {
23          MEM_RTE = 0,
24          MEM_DMA = 1,
25          MEM_N,
26      }RTE_MEM_Name_e;
```

如上图所示，枚举了两块使用的 RAM 空间。

然后，需要在 RTE_MEM.c 文件中完成结构体变量的初始化：

```
19  static RTE_MEM_t MemoryControlHandle[MEM_N] =
20  {
21      {
22          .MemName = MEM_RTE,
23          .work_mem = (void *)0,
24          .totalsize = 0,
25      },
26      {
27          .MemName = MEM_DMA,
28          .work_mem = (void *)0,
29          .totalsize = 0,
30      },
31  };
```

完成以上两步骤之后，便可以在其他需要的地方对动态内存管理进行初始化了，这里我们在 RTE_Port.c 中完成此项工作。

```
23  /*************************************************
24  *** RTE所管理的内存，静态分配，32位对齐
25  *************************************************/
26  #if RTE_USE_MEMMANAGE == 1
27  RTE_ALIGN_32BYTES (__attribute__((section (".RAM_RTE"))) uint8_t RTE_RAM[RTE_MEM_SIZE * 1024]) = {0};
28  RTE_ALIGN_32BYTES (uint8_t DMA_RAM[4 * 1024]) = {0};
29  #endif
```

如上所示，第一块内存我们通过 attribute 分配到了 F407 的 TCM 中，（.RAM_RTE 是在分散加载文件中定义的 TCM 区域别名）；第二块内存我们直接声明，由 KEIL 自动完成分配。

```
78  void RTE_Init(void)
79  {
80  #if RTE_USE_MEMMANAGE == 1
81      RTE_MEM_Pool(MEM_RTE,RTE_RAM,RTE_MEM_SIZE*1024);
82      RTE_MEM_Pool(MEM_DMA,DMA_RAM,4*1024);
83  #endif
```

接下来只需要调用 RTE_MEM_Pool 进行初始化即可。

至此，便可以使用动态内存管理模块了。

1.2 函数列表

1.2.1 内存池初始化

```
void RTE_MEM_Pool(RTE_MEM_Name_e mem_name,void *buf,uint32_t len);
```

功能：该函数完成对将要管理的 RAM 的初始化。

形参：（1）内存块名称（2）待管理 RAM 首地址（3）待管理 RAM 长度

返回值：无。

1.2.2 内存分配

```
void *RTE_MEM_Alloc(RTE_MEM_Name_e mem_name,uint32_t size);
```

功能：该函数完成从已初始化内存池的动态内存申请。

形参：（1）内存块名称（2）待申请空间大小

返回值：申请成功时返回地址，失败时返回 NULL。

1.2.3 内存值 0 分配

```
void *RTE_MEM_Alloc0(RTE_MEM_Name_e mem_name,uint32_t size);
```

功能：该函数完成从已初始化内存池的动态内存申请，并将该区域初始化为 0.

形参：（1）内存块名称（2）待申请空间大小

返回值：申请成功时返回地址，失败时返回 NULL。

1.2.4 内存释放

```
void RTE_MEM_Free(RTE_MEM_Name_e mem_name,const void * data);
```

功能：该函数完成对申请后的动态内存的释放;

形参：（1）内存块名称（2）待释放空间首地址

返回值：无。

1.2.5 内存再分配

```
void  *RTE_MEM_Realloc(RTE_MEM_Name_e  mem_name,void  *data_p,uint32_t
new_size);
```

功能：该函数完成对申请后的动态内存的重新申请;

形参：（1）内存块名称（2）待重新申请空间首地址（3）新申请空间大小

返回值：申请成功时返回地址，失败时返回 NULL。

1.2.6 内存整理

```
void RTE_MEM_Defrag(RTE_MEM_Name_e mem_name);
```

功能：该函数完成对选定内存块的碎片清理;

形参：（1）内存块名称

返回值：无

1.2.7 内存监控

```
void   RTE_MEM_Monitor(RTE_MEM_Name_e   mem_name,RTE_MEM_Monitor_t   *
mon_p);
```

功能：该函数完成对选定内存块的状态监控;

形参：（1）内存块名称（2）内存监控结构体变量指针

返回值：无。

1.2.8 内存大小获取

```
uint32_t RTE_MEM_GetDataSize(const void * data);
```

功能：该函数返回对申请的内存块的大小;

形参：（1）已申请空间首地址

返回值：申请空间的大小。

1.2.9 内存块最大空余

```
uint32_t RTE_MEM_MaxFree(RTE_MEM_Name_e mem_name);
```

功能：该函数返回选定内存块的最大可申请量;

形参：（1）内存块名称

返回值：最大可申请量。

2 时间片轮询

2.0 说明

RTE 自带的时间片轮询模块母版为：

http://stm32f4-discovery.net/2014/05/all-stm32f429-libraries-at-one-place/

RTE 在其基础上进行了大量修改，主要包括引入 VEC 模块和 MEM 模块进行动态管理；适配 RTOS，同时利用多线程进行分组管理。

2.1 使用方法

为了使用时间片轮询模块，需要为其提供时钟基准，即在 1MS 的定时器中断中调用：

```
7  #if RTE_USE_OS == 0
8  void SysTick_Handler(void)
9  {
10     RTE_RoundRobin_TickHandler();
11 }
12 #endif
```

在 RTOS 环境下，可以单独建立一个优先级最高的线程：

```
39 __NO_RETURN void ThreadTaskSYS (void *argument) {
40   osDelay(2);
41   StaticsIdleCnt = 0;
42   osDelay(100);
43   StaticsIdleCntMax = StaticsIdleCnt;
44   RTE_RoundRobin_CreateTimer(0,"LEDTimer",500,1,1,LEDTimer_Callback,(void *)0);
45   RTE_RoundRobin_CreateTimer(0,"StaticsTimer",100,1,1,SystemStaticsTimer_Callback,(void *)0);
46   RTE_Printf("[SYSTEM]    TuringBoart Run At %d Hz!\r\n",SystemCoreClock);
47   ThreadIDWIFI = osThreadNew(ThreadTaskWIFI, NULL, &WIFIThreadControl);
48   ThreadIDGUI = osThreadNew(ThreadTaskGUI, NULL, &GUIThreadControl);
49   ThreadIDSensor = osThreadNew(ThreadTaskSensor, NULL, &SensorThreadControl);
50   for (;;)
51   {
52     RTE_RoundRobin_TickHandler();
53     osDelay(1);
54   }
55 }
```

时间片轮询关于 RTOS 的适配基于 CMSIS_RTOS2 接口。

提供了时钟基准后，对时间片轮询默认的定时器组进行初始化，这里依然在 RTE_Port 中完成：

```
84 #if RTE_USE_ROUNDROBIN == 1
85     RTE_RoundRobin_Init();
86     RTE_RoundRobin_CreateGroup("RTEGroup");
```

裸机环境中，还需要在主循环中调用定时器组处理函数：

```
39   RTE_RoundRobin_CreateTimer(0,"LEDTimer",500,1,1,LEDTimer_Callback,(void *)0);
40   for(;;)
41   {
42     RTE_RoundRobin_Run(0);
43   }
```

RTOS 环境下不需要进行这一步。

如此即可正常使用。

2.2 函数列表

2.2.1 时间片轮询初始化

```
void RTE_RoundRobin_Init(void);
```

2.2.2 创建定时器组

```
 RTE_RoundRobin_Err_e RTE_RoundRobin_CreateGroup(const char
*GroupName);
```

2.2.3 获取定时器的 ID

```
 int8_t RTE_RoundRobin_GetGroupID(const char *GroupName);
```

2.2.4 创建定时器

```
 RTE_RoundRobin_Err_e RTE_RoundRobin_CreateTimer(
    uint8_t GroupID,
    const char *TimerName,
    uint32_t ReloadValue,
    uint8_t ReloadEnable,
    uint8_t RunEnable,
    void (*TimerCallback)(void *),
    void* UserParameters);
```

2.2.5 获取定时器的 ID

```
 int8_t RTE_RoundRobin_GetTimerID(uint8_t GroupID,const char
*TimerName);
```

2.2.6 删除定时器

```
 RTE_RoundRobin_Err_e RTE_RoundRobin_RemoveTimer(uint8_t
GroupID,uint8_t TimerID);
```

2.2.7 时间片轮询时基函数

```
 void RTE_RoundRobin_TickHandler(void);
```

2.2.8 定时器组运行

```
 void RTE_RoundRobin_Run(uint8_t GroupID);
```

2.2.9 就绪一个定时器

```
 RTE_RoundRobin_Err_e RTE_RoundRobin_ReadyTimer(uint8_t
GroupID,uint8_t TimerID);
```

2.2.10 复位一个定时器

```
 RTE_RoundRobin_Err_e RTE_RoundRobin_ResetTimer(uint8_t
GroupID,uint8_t TimerID);
```

2.2.11 暂停一个定时器

```
RTE_RoundRobin_Err_e RTE_RoundRobin_PauseTimer(uint8_t GroupID,uint8_t TimerID);
```

2.2.12 恢复一个定时器

```
RTE_RoundRobin_Err_e RTE_RoundRobin_ResumeTimer(uint8_t GroupID,uint8_t TimerID);
```

2.2.13 判断定时器是否运行

```
bool RTE_RoundRobin_IfRunTimer(uint8_t GroupID,uint8_t TimerID);
```

2.2.14 时间片轮询监控

```
void RTE_RoundRobin_Demon(void);
```

2.2.15 获取当前心跳值

```
uint32_t RTE_RoundRobin_GetTick(void);
```

2.2.16 计算当前与前次之时基差

```
uint32_t RTE_RoundRobin_TickElaps(uint32_t prev_tick);
```

2.2.17 延迟 MS 级别（CPU 占用率 100% RTOS 下可被抢占）

```
void RTE_RoundRobin_DelayMS(uint32_t Delay);
```

2.2.18 延迟 US 级别 （CPU 占用率 100% RTOS 下可被抢占 需要用到 DWT）

```
__inline void RTE_RoundRobin_DelayUS(volatile uint32_t micros);
```

3 Embeded Shell 与标准输出

3.0 说明

RTE 自带的嵌入式 Shell 模块和标准输出母版为：
http://software-dl.ti.com/tiva-c/SW-TM4C/latest/index_FDS.html
在其基础上，RTE 引入了 VEC 和 MEM 进行动态管理，同时适配 RTOS 加入互斥量。

3.1 使用方法

首先在 RTE_Port.c 中完成初始化，RTOS 环境下完成互斥量初始化，同时为 Shell 的处理建立一个定时器任务：

```
87      #if RTE_USE_SHELL == 1
88          RTE_Shell_Init();
89          RTE_RoundRobin_CreateTimer(0,"ShellTimer",100,1,1,RTE_Shell_Poll,(void *)0);
90      #endif
91  #endif
92  #if RTE_USE_STDOUT != 1
93      #if RTE_USE_OS == 1
94      static const osMutexAttr_t MutexIDStdioAttr = {
95          "StdioMutex",      // human readable mutex name
96          0U,                    // attr_bits
97          NULL,                // memory for control block
98          0U                   // size for control block
99      };
100     MutexIDStdio = osMutexNew(&MutexIDStdioAttr);
101     #endif
102 #endif
```

如果要使用标准输出，需要为其注册物理输出函数：

```
23  void RTE_Puts (const char *pcString,uint16_t length)
24  {
25      HAL_UART_Transmit(&UartHandle[USART_DEBUG].UartHalHandle, (uint8_t *)pcString,length,HAL_MAX_DELAY);
26  }
```

```
36      RTE_Reg_Puts(RTE_Puts);
37      RTE_Printf("HelloWorld!\r\n");
```

当有数据需要送入 Shell 进行处理时，调用如下：

```
32  RTE_Shell_Input(UartHandle[*usart_name].UsartData.pu8Databuf,UartHandle[*usart_name].UsartData.u16Datalength);
```

如此，即完成模块的使用。

3.2 函数列表

3.2.1 输出设备注册

```
void RTE_Reg_Puts(void (*PutsFunc)(const char *,uint16_t));
```

3.2.2 标准输出（不支持%f)

```
void RTE_Printf(const char *pcString, ...);
```

3.2.3 为 Shell 模块添加指令

```
RTE_Shell_Err_e RTE_Shell_AddCommand(const char *cmd,RTE_Shell_Err_e
(*func)(int argc, char *argv[]),const char *help);
```

3.2.4 删除一个 Shell 模块中的指令

```
RTE_Shell_Err_e RTE_Shell_DeleteCommand(const char *cmd);
```

3.2.5 初始化 Shell 模块

```
void RTE_Shell_Init(void);
```

3.2.6 Shell 模块定时器任务

```
void RTE_Shell_Poll(void *Params);
```

3.2.7 Shell 模块输入

```
void RTE_Shell_Input(uint8_t *Data,uint16_t Length);
```

4 MessageQuene 与 RingBuffer

4.0 说明

RTE 自带的 RingBuffer 模块母版为：

http://software-dl.ti.com/tiva-c/SW-TM4C/latest/index_FDS.html

在其基础上，RTE 引入了 MEM 进行动态管理，同时进行封装，在此基础上完成了 MessageQuene。

4.1 使用方法

RingBuffer 的使用方法：

```
MessageQuene->QueneBuffer = (uint8_t *)RTE_MEM_Alloc0(MEM_RTE,Size);
RTE_AssertParam(MessageQuene->QueneBuffer);
RTE_AssertParam(RTE_RingQuene_Init(&MessageQuene->RingBuff,MessageQuene->QueneBuffer,sizeof(uint8_t),Size));
```

首先声明结构体变量，然后为其申请缓存，然后调用 RTE_RingQuene_Init 进行初始化。

MessageQuene 的使用方法：

```
199     RTE_MessageQuene_Init(&ShellQuene,SHELL_BUFSIZE);
```

首先声明结构体变量，然后调用 RTE_MessageQuene_Init 进行初始化。

4.2 函数列表

4.2.1 初始化 RingBuffer

```
int RTE_RingQuene_Init(RTE_RingQuene_t *RingBuff, void *buffer, int itemSize, int count);
```

4.2.2 清空一个 RingBuffer

```
static inline void RTE_RingQuene_Flush(RTE_RingQuene_t *RingBuff);
```

4.2.3 获取一个 RingBuffer 大小

```
static inline int RTE_RingQuene_GetSize(RTE_RingQuene_t *RingBuff);
```

4.2.4 获取一个 RingBuffer 已有数据大小

```
static inline int RTE_RingQuene_GetCount(RTE_RingQuene_t *RingBuff);
```

4.2.5 获取一个 RingBuffer 空余

```
static inline int RTE_RingQuene_GetFree(RTE_RingQuene_t *RingBuff);
```

4.2.6 判断一个 RingBuffer 是否已满

```
static inline int RTE_RingQuene_IsFull(RTE_RingQuene_t *RingBuff);
```

4.2.7 判断一个 RingBuffer 是否为空

```
static inline int RTE_RingQuene_IsEmpty(RTE_RingQuene_t *RingBuff);
```

4.2.8 插入单个数据

```
int RTE_RingQuene_Insert(RTE_RingQuene_t *RingBuff, const void *data);
```

4.2.9 插入多个数据

```
int RTE_RingQuene_InsertMult(RTE_RingQuene_t *RingBuff, const void *data, int num);
```

4.2.10 取出单个数据

```
int RTE_RingQuene_Pop(RTE_RingQuene_t *RingBuff, void *data);
```

4.2.11 取出多个数据

```
int RTE_RingQuene_PopMult(RTE_RingQuene_t *RingBuff, void *data, int num);
```

4.2.12 MessageQuene 初始化

```
void RTE_MessageQuene_Init(RTE_MessageQuene_t *MessageQuene, uint16_t Size);
```

4.2.13 消息入列

```
RTE_MessageQuene_Err_e RTE_MessageQuene_In(RTE_MessageQuene_t *MessageQuene, uint8_t *Data,uint16_t DataSize);
```

4.2.14 消息出列

```
RTE_MessageQuene_Err_e RTE_MessageQuene_Out(RTE_MessageQuene_t *MessageQuene, uint8_t *Data,uint16_t *DataSize);
```

5 KVDB 模块

5.0 说明

本模块母版为：https://github.com/armink/EasyFlash
RTE 删减了其余部分，只保留了 KVDB 的部分。

5.1 使用方法

定义默认的 KV 数组：

```
34  /* default ENV set for user */
35  static const ef_env rte_kvdb_lib[] = {
36      {"boot_times","0"},
37  };
```

初始化：

```c
void RTE_KVDB_Init(void)
{
    #if RTE_USE_OS == 1
    static const osMutexAttr_t MutexIDKVDBAttr = {
        "KVDBMutex",      // human readable mutex name
        0U,                     // attr_bits
        NULL,                // memory for control block
        0U                    // size for control block
    };
    MutexIDKVDB = osMutexNew(&MutexIDKVDBAttr);
    #endif
    size_t default_env_set_size = sizeof(rte_kvdb_lib) / sizeof(rte_kvdb_lib[0]);
    EfErrCode result = EF_NO_ERR;
    if (result == EF_NO_ERR)
    {
        result = ef_env_init(rte_kvdb_lib, default_env_set_size);
```

完成 Flash 或者其他存储设备的读写接口：

```c
EfErrCode ef_port_read(uint32_t addr, uint32_t *buf, size_t size) {
    EfErrCode result = EF_NO_ERR;
    RTE_AssertParam(size % 4 == 0);
    /*copy from flash to ram */
    for (; size > 0; size -= 4, addr += 4, buf++) {
        *buf = *(uint32_t *) addr;
    }
    return result;
}
```

```c
EfErrCode ef_port_erase(uint32_t addr, size_t size) {
    EfErrCode result = EF_NO_ERR;
    FLASH_Status flash_status;
    size_t erased_size = 0;
    uint32_t cur_erase_sector;

    /* make sure the start address is a multiple of EF_ERASE_MIN_SIZE */
    RTE_AssertParam(addr % KVDB_ERASE_MIN_SIZE == 0);
```

```c
EfErrCode ef_port_write(uint32_t addr, const uint32_t *buf, size_t size) {
    EfErrCode result = EF_NO_ERR;
    size_t i;
    uint32_t read_data;

    RTE_AssertParam(size % 4 == 0);

```

即可使用：

```c
            c_old_boot_times = ef_get_env("boot_times");
            RTE_AssertParam(c_old_boot_times);
            i_boot_times = atol(c_old_boot_times);
            /* boot count +1 */
            i_boot_times ++;
            RTE_Printf("%10s    The system boot times:%d\r\n",KVDB_TXT,i_boot_times);
            /* interger to string */
            usprintf(c_new_boot_times,"%d", i_boot_times);
            ef_set_env("boot_times", c_new_boot_times);
            ef_save_env();
```

5.2 函数列表

5.2.1 加载 KV 数据库

```c
EfErrCode ef_load_env(void);
```

5.2.2 输出所有的 KV 数据

```c
void ef_print_env(void);
```

5.2.3 获取一个 KV 数据

```
char *ef_get_env(const char *key);
```

5.2.4 为一个 KV 数据设置新值

```
EfErrCode ef_set_env(const char *key, const char *value);
```

5.2.5 删除一个 KV 数据

```
EfErrCode ef_del_env(const char *key);
```

5.2.6 保存当前 KV 数据库

```
EfErrCode ef_save_env(void);
```

5.2.7 重置 KV 数据库为默认

```
EfErrCode ef_env_set_default(void);
```

5.2.8 获取写字节

```
size_t ef_get_env_write_bytes(void);
```

5.2.9 设置并保存 KV 数据库

```
EfErrCode ef_set_and_save_env(const char *key, const char *value);
```

5.2.10 删除并保存 KV 数据库

```
EfErrCode ef_del_and_save_env(const char *key);
```

5.2.11 初始化 KV 数据库

```
EfErrCode ef_env_init(ef_env const *default_env, size_t
default_env_size);
```

5.2.12 读接口

```
EfErrCode ef_port_read(uint32_t addr, uint32_t *buf, size_t size);
```

5.2.13 擦除接口

```
EfErrCode ef_port_erase(uint32_t addr, size_t size);
```

5.2.14 写接口

```
EfErrCode ef_port_write(uint32_t addr, const uint32_t *buf, size_t
size);
```

5.2.15 互斥锁接口

```
void ef_port_env_lock(void);
void ef_port_env_unlock(void);
```

6 其他

——（VEC 模块、UString 模块、MATH 模块、LOG 模块、LinkList 模块）

6.0 说明

该部分的模块多为支持其他大模块的内容。
VEC 模块：动态数组实现。
UString 模块：代替 c 库字符串处理的模块。
MATH 模块：一些数学计算。
LOG 模块：标准输出的封装。
LinkList 模块：双向链表。

7 GUI 模块

7.0 说明

该部分内核为 LVGL，RTE 对其进行了接口对接（见 GUI_Port），有关 GUI 的相关说明和手册请参考 GUI 文档。

8 机器视觉模块

8.0 说明

该部分内核为 openmv，RTE 对其进行了接口对接，有关 MV 的相关说明和手册请参考 TuringBoard 用户手册。