SL_RTE 用户手册

作者：单雷 日期：2019/03/19 RTE_VER：3.8.0319
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，主要包含八个部分：Core 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；Module 层，对以上几层的进一步封装，为用户使用 SL_RTE 提供一个较为规范的引导。

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（因为 BSP 层和 Board 层大都是 STM32 系列的实现，如果用户本身积累了相当的 MCU 相关的 BSP，那么同样可以用于 SL_RTE）。

实时操作系统下的 SL_RTE 和安卓的机制类似，设计好的 GUI 界面的刷新只在 GUI 线程中完成，其余线程不能对 GUI 界面产生直接影响。

在 3.7.0319 版本中，正式完成了 APP_Module，该模块作为 RTE_Module 部分的第一个模块，初步确立了基于 SL_RTE 的应用编写规范。

（1）一个完整的应用可以由以下几种方式实现：

裸机；

实时操作系统；

裸机+GUI；

实时操作系统+GUI；

（2）在使用实时操作系统但不使用 GUI 的环境下，可以采用线程+状态机的方式实现

整个应用；在裸机且不使用 GUI 的环境下，可以采用前台（时间片轮询+状态机）+后台（各种中断）的方式实现整个应用。

（3）在使用 GUI 时，一个完成的应用可以由一个或者多个图形 APP 构成。如果是在操作系统环境下，每个 APP 可以拥有一个线程。对于拥有线程的 APP 而言，其 app_run 和 app_close 需要实现线程的创建和删除。

（4）每个 APP 的生命周期由如下几个部分组成：

a.建立 APP 初期：　（该部分由 RTE 自己完成 不需要用户交互参与）

APP 建立->APP_GUI 绘制->app_sc_open

该部分完成于整个 SL_RTE 初始化期间，在 GUI 线程进入线程体之前。

b.APP 开始：（该部分需要用户交互参与）

用户点击 APP 图标->app_run->APP 窗口绘制->app_win_open

该部分在功能上实现了 APP 的主窗体绘制以及 APP 内部窗体绘制，同时在 app_run 函数中，应当完成 APP 线程的建立。

c.APP 运行时操作：（该部分需要用户交互参与）

用户点击关闭->关闭动画->app_win_close->APP 窗口销毁

此时 APP 真正关闭 线程退出 APP 内部相关资源需要释放

用户点击最小化>APP 窗口隐藏

此时 APP 进入后台状态 线程继续运行

d.APP 生命周期结束：（该部分通常不需要实现）

调用 APP_Remove->销毁可能存在的 APP 窗口->APP_GUI 销毁->释放 APP 资源

这一阶段是真正意义上的 APP 生命周期结束，如果发生，那么主界面上的 APP 相关 GUI 也会消失，所有 APP 有关的资源都会被释放。

（5）对于拥有线程的 APP 而言，线程内可以由状态机模块实现，不同的状态可以内部切换，也可以由用户与 GUI 交互实现切换（即 GUI 线程对 APP 线程的改变）。

（6）在裸机环境下，GUI 模块作为 RR 的 system 组的一个定时器存在。

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

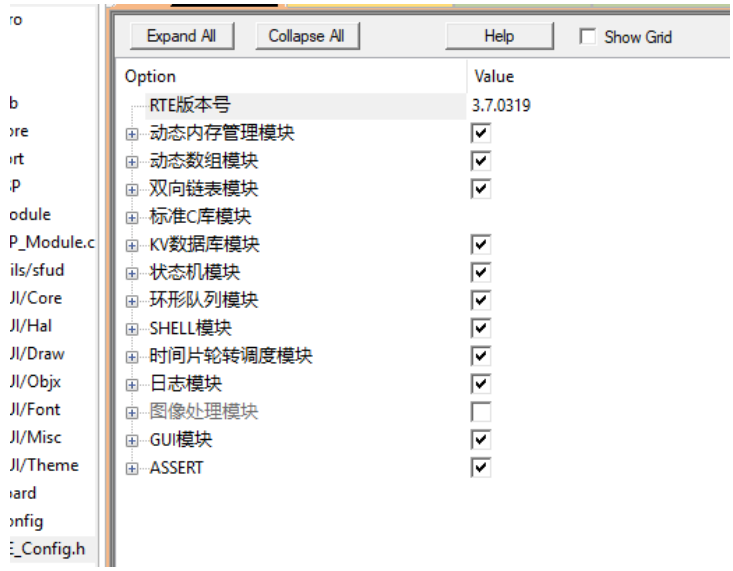
https://github.com/ARM-software/CMSIS_5

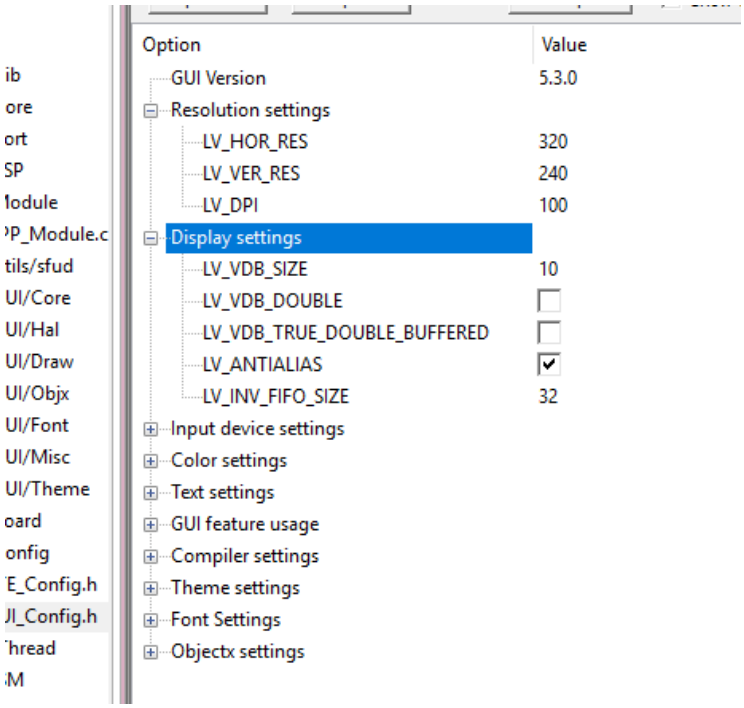
http://blog.chinaunix.net/uid-22915173-id-2185545.html

http://www.gii.upv.es/tlsf/mm/intro

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

需要注意的是，本 RTE 需要配合 KEIL5 以上版本使用，在 KEIL 中可以进行如下所示的界面配置：





1 内存管理

1.0 说明

本 RTE 自带的内存管理原始为 tlsf 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。

Tlsf solves the problem of the worst case bound maintaining the efficiency of the allocation and deallocation operations allows the reasonable use of dynamic memory management in real-time applications. The proposed algorithm, with a constant cost Θ(1), opens new possibilities with respect to the use of dynamic memory in real-time applications. There exists an increasing number of emerging applications using high amounts of memory, such as multimedia systems, video streaming, video surveillance, virtual reality, scientific data gathering, or data acquisition in control systems.

tlsf 与常见的内存管理实现对比如下（这里的四个 test 为 tlsf 实现的四个测试，可以从其官网获得）：

malloc()	*First-Fit*		*Best-Fit*		*DL's malloc*		*Binary Buddy*		*TLSF*	
	worst	mean	worst	mean	worst	mean	worst	mean	worst	mean
Test 1	**25636208**	**11256641**	**17007384**	**10322776**	168	81	4140	1239	155	148
Test 2	2124	1971	2124	1971	**16216**	**15974**	244	220	172	148
Test 3	568	201	592	197	128	76	**5660**	**5448**	120	115
Test 4	792	235	536	193	124	75	5460	5309	**189**	**168**

free()	*First-Fit*		*Best-Fit*		*DL's malloc*		*Binary Buddy*		*TLSF*	
	worst	mean	worst	mean	worst	mean	worst	mean	worst	mean
Test 1	528	103	2012	899	460	67	2728	345	140	97
Test 2	428	150	428	150	**96**	**71**	1976	1448	152	96
Test 3	340	107	324	113	560	131	**6512**	**3504**	124	93
Test 5	348	157	372	204	136	68	3728	2881	**188**	**164**

Table 1. Test summary for the malloc and free operations. Results are given in number of processor cycles.

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_Memory.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。

然后就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 Memory_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 Memory_Pool(mem_bank_e bank,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 * Memory_Alloc(mem_bank_e bank,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 * Memory_Alloc0(mem_bank_e bank,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 Memory_Free(mem_bank_e bank,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 * Memory_Realloc(mem_bank_e bank,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 已申请空间大小获取

```
uint32_t Memory_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。

1.2.7 内存块最大空余

```
void *Memory_AllocMaxFree(mem_bank_e bank,uitn32_t *size);
```

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 （2）保存最大申请量的变量地址

返回值：申请成功时返回地址，失败时返回 NULL。

1.2.8 获取内存块使用量

```
uint32_t Memory_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 内存块监控

```
uint32_t Memory_Demon(mem_bank_e bank);
```

功能：对指定的内存块的使用情况进行统计;

形参：（1）内存块名称

返回值：无，会打印如下信息：

```
453 ------------------------------------------------
454 ------------------------------------------------
455 [MEM]         BANK COUNTS:2
456 ------------------------------------------------
457 [MEM]         BANK1 start at 20001ed8
458 ------------------------------------------------
459 [MEM]         >> [20002b48] (8 bytes, used, prev used)
460 [MEM]         >> [20002b58] (128 bytes, used, prev used)
461 [MEM]         >> [20002be0] (64 bytes, used, prev used)
462 [MEM]         >> [20002c28] (2048 bytes, used, prev used)
463 [MEM]         >> [20003430] (27288 bytes, free [0, 0], prev used)
464 [MEM]         >> [20009ed0] (sentinel, used, prev. free [20003430])
465 ------------------------------------------------
```

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。

在 3.30124 版本中，SL_RTE 对原有的 shell 进行了彻底的重构，实现了 shell 内的多模块。更远的计划是利用一种合适的脚本语言替代 shell 模块。

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 添加一个 Shell 模块组

```
RTE_Shell_Err_e RTE_Shell_CreateModule(const char *module);
```

3.2.4 为一个 Shell 模块组添加指令

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

3.2.5 删除一个 Shell 模块组中的指定指令

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

3.2.6 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);
```

3.3 使用示例

完成 SL_RTE 的移植后，在串口终端中输入 system.help，即可看到框架内全部 SHELL 模块：

```
RTE Version:3.7.0319
-------------------------------------------------
[SHELL]       Available Module
---------------------------
[SHELL]         Module Name:system
[SHELL]           Fuction Name:help           ---:Available help when using Shell Example:system.help
[SHELL]           Fuction Name:print          ---:Simple printf Example:system.print(str)
---------------------------
[SHELL]         Module Name:timer
[SHELL]           Fuction Name:pause          ---:Pause a running timer Example:timer.pause(groupname,timername)
[SHELL]           Fuction Name:resume         ---:Resume a pasuing timer Example:timer.resume(groupname,timername)
[SHELL]           Fuction Name:demon          ---:Demon all running timers Example:timer.demon
---------------------------
[SHELL]         Module Name:mem
[SHELL]           Fuction Name:demon          ---:Demon a specific membank Example:mem.demon(0)
---------------------------
[SHELL]         Module Name:kvdb
[SHELL]           Fuction Name:printenv       ---:Print all env variables Example:kvdb.printenv
[SHELL]           Fuction Name:newenv         ---:Create a env variable Example:kvdb.newenv(key,value)
[SHELL]           Fuction Name:setenv         ---:Set a env variable Example:kvdb.setenv(key,value)
[SHELL]           Fuction Name:delenv         ---:Delete a env variable Example:kvdb.delenv(key)
[SHELL]           Fuction Name:resetenv       ---:Reset all env variables Example:kvdb.resetenv
---------------------------
[SHELL]         Module Name:ESP8266
[SHELL]           Fuction Name:AT             ---:ESP8266 Debug
```

按照给出的 demo 使用即可。

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。
在 3.4.0313 版本中，该部分更新到了 EasyFlash 的最新版本 4.0.0。

5.1 使用方法

定义默认的 KV 数组：

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

初始化：

```
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

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

5.2.4 为一个 KV 数据设置新值

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

5.2.5 删除一个 KV 数据

```c
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、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 RTE 所需的数学计算实现。

LinkList 模块：双向链表实现。

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 https://docs.littlevgl.com/。

7.1 GUI 应用框架

利用 GUI 进行程序设计的基本思路请参见 0.1 小节。

7.2 函数列表

7.2.1 初始化程序 APP 框架

```
extern void APP_Init(lv_obj_t *father,const char *name);
```

7.2.1 为 APP 框架添加一个 APP 描述句柄

```
extern void APP_Add(app_dsc_t *app,void * conf);
```

7.2.2 为一个已经添加的 APP 描述句柄新建其实例

```
extern app_inst_t * APP_New(app_dsc_t * app_dsc, void * conf);
```

7.2.3 移除 APP 框架里的一个 APP 实例

```
extern void APP_Remove(app_inst_t * app);
```

7.2.4 初始化一个 APP 实例的 GUI 系统

```
extern lv_obj_t * APP_GUI_Init(app_inst_t * app);
```

7.2.5 销毁一个 APP 实例的 GUI 系统

```
extern void APP_GUI_DeInit(app_inst_t * app);
```

7.2.6 打开一个 APP 实例的窗口

```
extern lv_obj_t * APP_Win_Open(app_inst_t * app);
```

7.2.7 关闭一个 APP 实例的窗口

```
extern void APP_Win_Close(app_inst_t * app);
```

7.2.8 重命名一个 APP 实例

```
extern void APP_Rename(app_inst_t * app, const char * name);
```

7.2.9 根据命名获取一个 APP 实例

```
extern app_dsc_t * APP_Dsc_Get(const char * name);
```

7.2.10 检查两个 APP 实例之间的通讯

```
extern bool APP_Con_Check(app_inst_t * sender, app_inst_t *
receiver);
```

7.2.11 为两个 APP 实例建立通讯

```
extern void APP_Con_Set(app_inst_t * sender, app_inst_t * receiver);
```

7.2.12 删除两个 APP 实例的通讯

```
extern void APP_Con_Del(app_inst_t * sender, app_inst_t * receiver);
```

7.2.13 两个 APP 实例之间进行数据传送

```
extern  uint16_t  APP_Con_Send(app_inst_t  *  app_send,  app_com_type_t
type , const void * data, uint32_t size);
```

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。

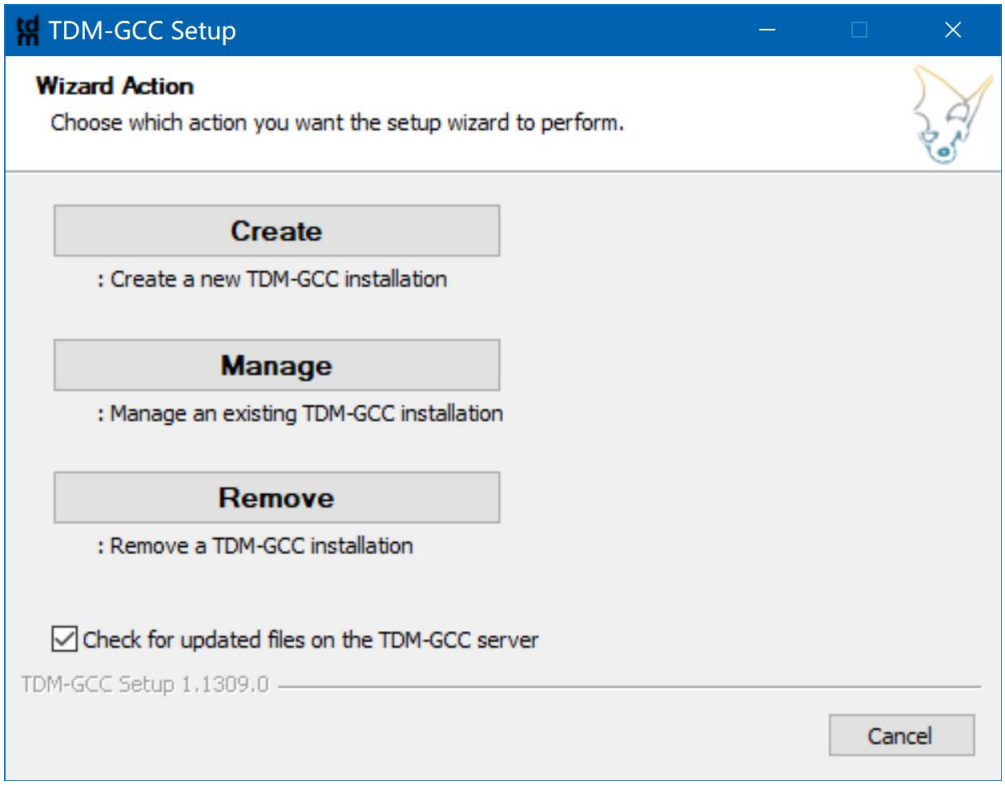
9 模拟器

LVGL 目前尚没有 GUI_Builder，同时为了方便用户在正式开发前熟悉 SL_RTE 的环境使用。改小节使用 codeblock+gcc+sdl2+windos api 这样的框架搭建了一个 SL_RTE 的上位机使用环境，并且用镜像成功地模拟了存储设备。

9.0 模拟器环境搭建

首先安装 CodeBlocks，一路 Next 即可，安装好了会提示没有合适的编译器。

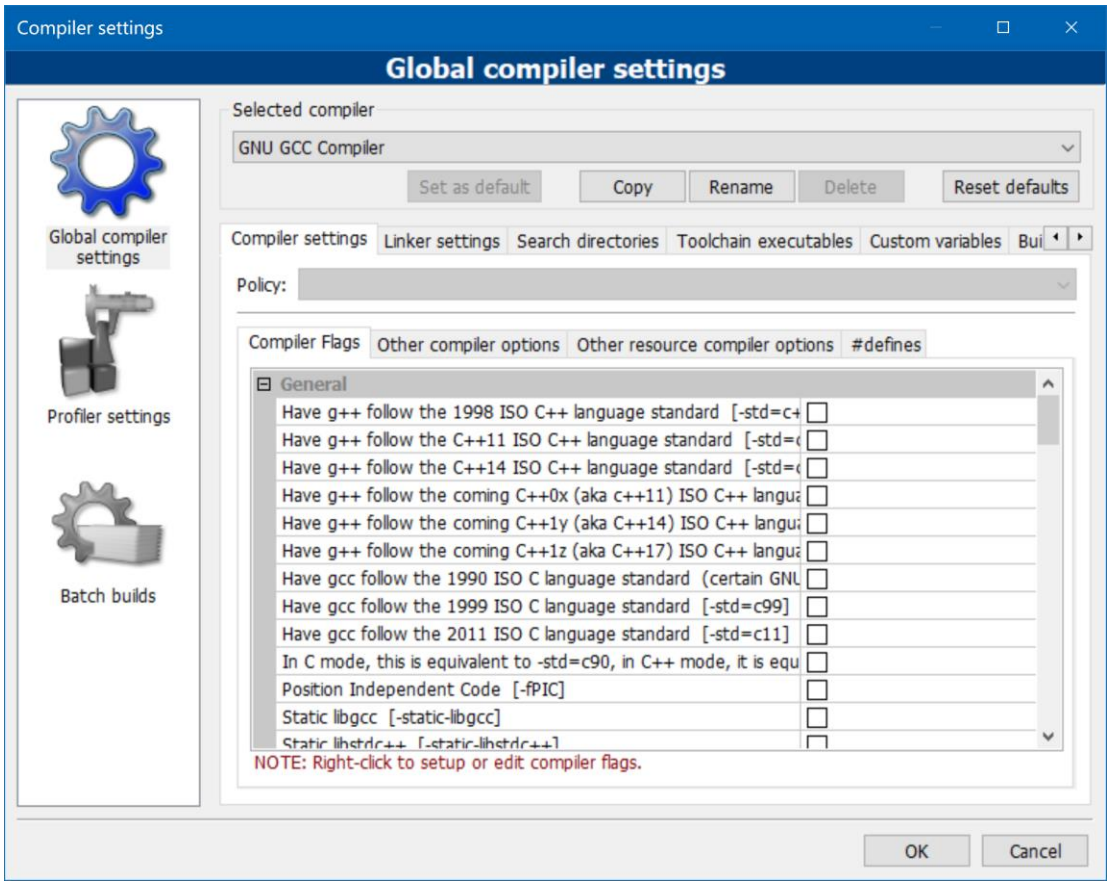
接着安装 TDM-GCC:

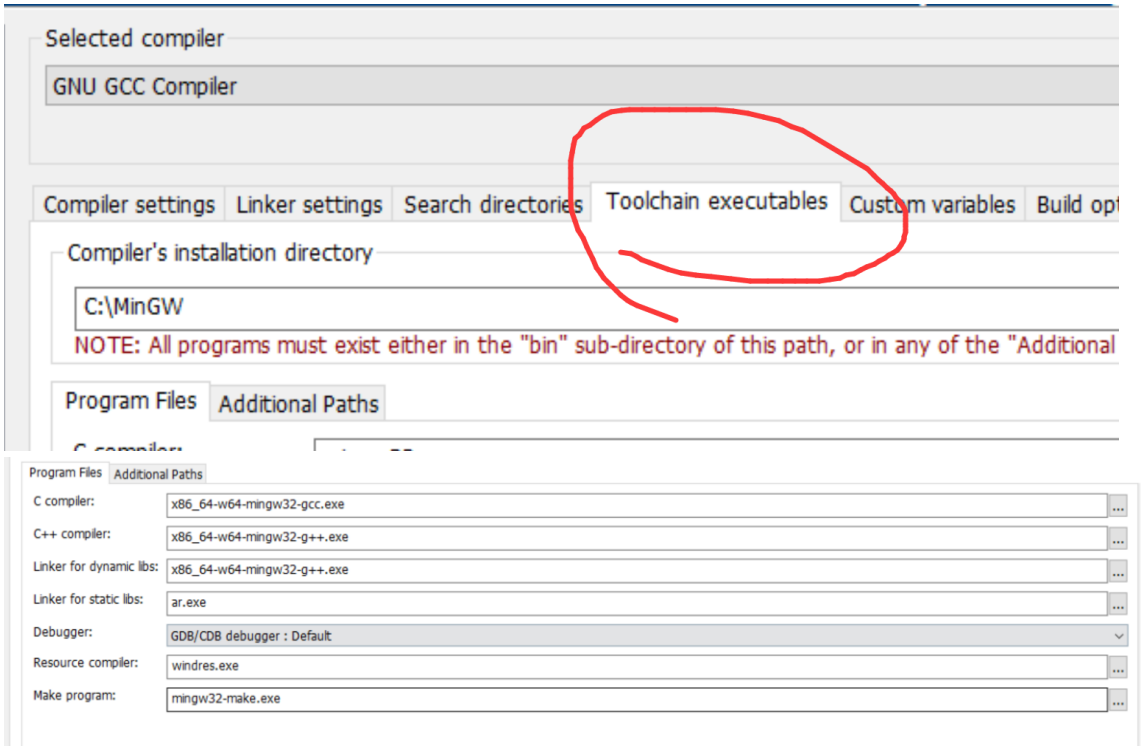


选择 Create，然后选择安装类型，一路 next。

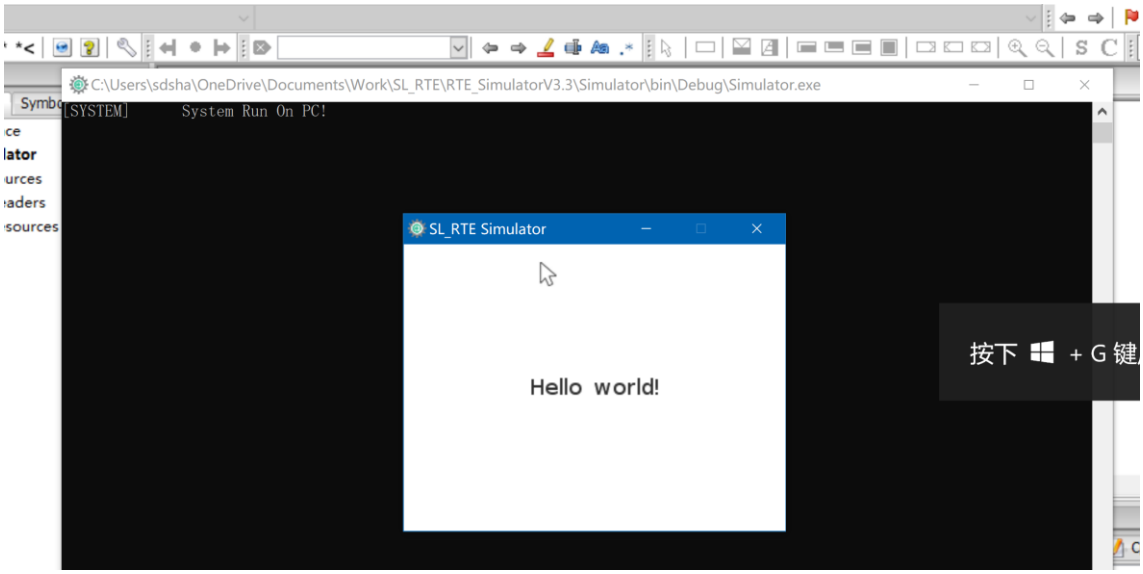
安装好了之后需要在 CodeBlocks 中添加编译器:

选择 settings-compiler。





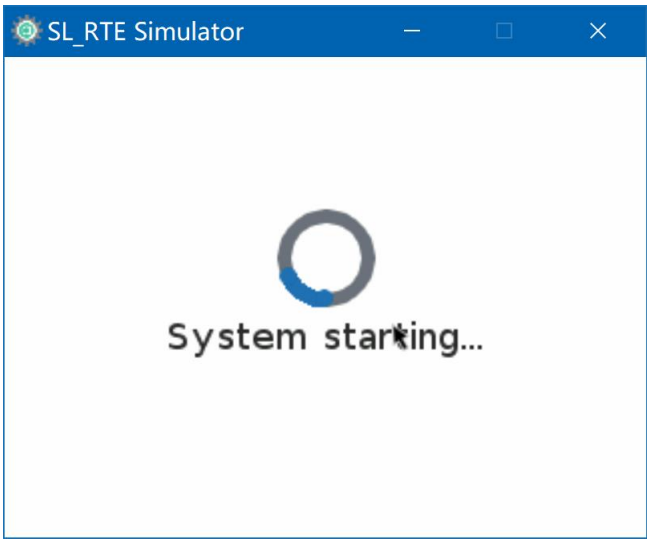
OK 即可，然后打开已经移植好的模拟器。



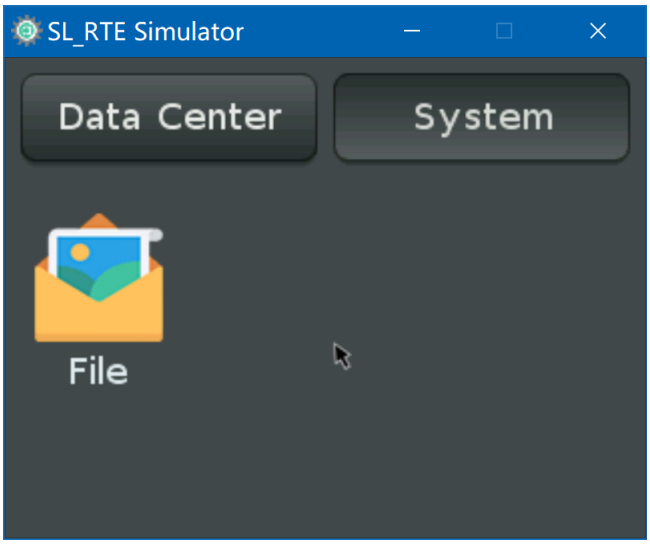
命令行界面类似于串口，可以输入 shell 指令。

我们得主界面依然是 tabview：

开机时可以通过 preloader 控件和一个定时器实现加载动画：



Tabview 控件效果主界面如下：



9.1 模拟器 GUI 界面开发

9.2 其他

10 版本历史

note：使用固件库建立工程时需要分别修改 stm32fxxx.h 和 system_stm32fxxx.c 中有关时钟配置的部分

使用 HAL 库建立工程需要修改 stm32fxxx_hal_conf.h 中时钟配置的部分

2017/09/22

版本 1.0release：

2017/09/23

完全剥离 bsp 和 app 确保 app 使用时独立于硬件底层（SoftTimer 除外 但针对 coterxm 内核 mcu 一致）
加入 bget 内存管理 将部分 app 变为动态内存分配机制（用于 debug 的串口缓存）
修改 ringbuffer 使出列和入列变为动态内存分配

2017/11/06

结合 tmlib 开始 1.1 版本编写 基于 F7 以及 HAL 库

2017/11/12

2.0 版本的除了 spi 其他都已经测试通过
为了配合 mppt 项目开始完成针对 f1 系列以及固件库的适配
修改 app 中 led 和 key 到 bsp 中去

2017/11/13

完成 com key led 对于 f1 的适配

修改了 key 和 led 使其适配于 m3 和 m4 系列内核
单独为 m3 内核建立 comf1
确定新 bsp 撰写规范
攥写了 bsp_timerbase

2018/04/17

新版本开始编写 1.2
app_mem 更换 rtx 自带的内存管理系统做为底层 方便管理多块内存空间。
增加 app_stdio 替换 printf
更换 APP_Config 和 BSP_Config 为适用于 MDK 开发环境，可勾选配置。

2018/05/30

基本完成 F1 板级支持库的开发；
开始做 F4 的板级支持库和 bsp 包；

2018/06/15

完成对 bget 的重写 使其支持多块内存 删除 app_mem
重命名 SL_LIB 为 SL-RTE
修改 F1 的串口接收为 DMA 工作模式
优化对 MDK 开发的适配
最新的 RTE 在 STM32F103C8T 平台上通过 1ms 循环 debug 指令暴力测试

2018/08/07 版本号 2.0.0807

更换命名；
重写时间片；
重写 Debug 并更名为 shell；
重写 Config 文件；
管理思想改为动态管理；

2018/08/26 版本号 2.1.0826

整理了 RTE 文件框架；
移植了 ESAYFLASH 作为 KEY-VALUE 数据库；
目前用于 TM1294 项目（SHELL 与 SM 静态版本）；

2018/08/31 版本号 2.2.0831

换用新的内存管理；
重写了 roundrobin 和 shell；
目前用于 407 遥控开关项目；（SHELL 与 SM 动静态结合版本，纯动态版本会产生较多的内存碎片）；

2018/09/02 版本号 2.2.0902

407 的串口 bsp 文件重写；
完成 RTE_Stdlib 和 RTE_Stdio 的移植；
完善 RTE_RetargetIO，使其适配于非 MicroLib 环境；

修正了 Config 文件;

2018/09/30 版本号 2.3.0930

增加 RTE_Vec 和 RTE_List;
利用 RTE_Vec 重新对 Shell、Timer、SM 等模块进行重构;
开始移植 GUI;

2018/10/2 版本号 3.0.1002

完成了 GUI 的移植，GUI 版本号为 5.0.1;
完善了 RR 部分代码;

2018/11/2 版本号 3.1.1102

重构 GUI 代码，GUI 版本号为 5.2.0;
重构目录树;
重写 RTE_Config;

2018/12/27 版本号 3.2.1227

lvgl 内核升级到 5.2.1，修复部分 RTE_APP;
一些其他小改动;

2019/01/07 版本号 3.3.0107

lvgl 更新优化接口;
MEM、Message 增加互斥锁;
STDOUT 和 KVDB 优化互斥锁逻辑;
LOG 用于各个模块;
升级日志模块，可选输出到 RTE_Stdout、FILE、Flash;
优化 include 结构;

2019/01/13 版本号 3.3.0113

修复了 MEM、Message 的互斥锁逻辑;

2019/01/24 版本号 3.3.0124

删除无用的 GUI_File 模块;
完善 PC 使用 SL_RTE 的逻辑;
更新了 Shell 模块，并为 GUI 增加了 Shell 支持，当前包括 obj 建立和删除。

2019/01/30 版本号 3.3.0130

完善 PC 使用 SL_RTE 的逻辑;
为 GUI 增加了动态管理;

2019/03/13 版本号 3.4.0313

升级 GUI 到 5.3.0
升级 KVDB 到 4.0.0

2019/03/16 版本号 3.6.0316

换用 umm_malloc 作为内存管理模块；

2019/03/17 版本号 3.7.0317

换用 tlsf 作为内存管理模块；

2019/03/19 版本号 3.8.0319

完成了 APP_Module 的编写，正式确立 GUI 和 RTOS 编程规范；

下一版本计划

1、IAP 库的开发和移植；
2、lua 脚本解释器；
3、RTE_Simulator 的功能完善：网络、串口通讯、文件、线程机制转换到 cmsis 接口；