ToolKit

version V1.0.0 date 2019.01.31

1、介绍

ToolKit是一套应用于嵌入式系统的通用工具包，可灵活应用到有无RTOS的程序中，采用C语言面向对象的思路实现各个功能，尽可能最大化的复用代码，目前为止工具包包含：**循环队列、软件定时器、事件集。**

- **Queue** 循环队列
    1. 支持动态、静态方式进行队列的创建与删除。
    2. 可独立配置缓冲区大小。
    3. 支持数据**最新保持**功能，当配置此模式并且缓冲区已满，若有新的数据存入，将会移除最早数据，并保持缓冲区已满。
- **Timer** 软件定时器
    1. 支持动态、静态方式进行定时器的创建与删除。
    2. 支持**循环**、**单次**模式。
    3. 可配置有无超时回调函数。
    4. 可配置定时器工作在**周期**或**间隔**模式。
    5. 使用双向链表，超时统一管理，不会因为增加定时器而增加超时判断代码。
- **Event** 事件集
    1. 支持动态、静态方式进行事件集的创建与删除。
    2. 每个事件最大支持**32**个标志位。
    3. 事件的触发可配置为**“标志与”**和**“标志或”**。

2 、文件目录

```
toolkit
├── include                         // 包含文件目录
|   ├── toolkit.h                   // toolkit头文件
|   └── toolkit_cfg.h               // toolkit配置文件
├── src                             // toolkit源码目录
|   ├── tk_queue.c                  // 循环队列源码
|   ├── tk_timer.c                  // 软件定时器源码
|   └── tk_event.c                  // 事件集源码
├── samples                         // 例子
|   ├── tk_queue_samples.c          // 循环队列使用例程源码
|   ├── tk_timer_samples.c          // 软件定时器使用例程源码
|   └── tk_event_samples.c          // 事件集使用例程源码
└── README.md                       // 说明文档
```

3 、函数定义

3.1 配置文件

- **ToolKit配置项**

宏定义	描述
TOOLKIT_USING_ASSERT	ToolKit使用断言功能
TOOLKIT_USING_QUEUE	ToolKit使用循环队列功能
TOOLKIT_USING_TIMER	ToolKit使用软件定时器功能
TOOLKIT_USING_EVENT	ToolKit使用事件集功能

- **Queue 循环队列配置项**

宏定义	描述
TK_QUEUE_USING_CREATE	Queue 循环队列使用动态创建和删除

- **Timer 软件定时器配置项**

宏定义	描述
TK_TIMER_USING_CREATE	Timer 软件定时器使用动态创建和删除
TK_TIMER_USING_INTERVAL	Timer 软件定时器使用为间隔模式
TK_TIMER_USING_TIMEOUT_CALLBACK	Timer 软件定时器使用超时回调函数

- **Event 事件集配置项**

宏定义	描述
TK_EVENT_USING_CREATE	Event 事件集使用动态创建和删除

> **说明**：当配置**TOOLKIT_USING_ASSERT**后，所有功能都将会启动参数检查。

3.2 Queue 循环队列API函数

> 以下为详细API说明及简要示例程序，综合demo可查看tk_queue_samples.c示例。

3.2.1 动态创建队列

> **注意**：当配置**TOOLKIT_USING_QUEUE**后，才能使用此函数。此函数需要用到**malloc**。

```
tk_queue_t tk_queue_create(uint16_t poolsize, bool keep_fresh);
```

参数	描述
poolsize	缓存区大小(单位字节)
keep_fresh	是否为保持最新模式，**true**：保持最新；**false**：默认(存满不能再存)
返回值	创建的队列对象（**NULL**为创建失败）

队列创建示例：

```c
int main(int argc, char *argv[])
{
    /* 动态方式创建一个循环队"queue",缓冲区大小50字节，不保持最新 */
    tk_queue_t queue = tk_queue_create(50, false);
    if( queue == NULL){
        printf("队列创建失败!\n");
    }
    /* ... */
    /* You can add your code under here. */
    return 0;
}
```

3.2.2 动态删除队列

> **注意**：当配置**TOOLKIT_USING_QUEUE**后，才能使用此函数。此函数需要用到**free**。必须为**动态**方式创建的队列对象。

```c
bool tk_queue_delete(tk_queue_t queue);
```

参数	描述
queue	要删除的队列对象
返回值	**true**：删除成功；**false**：删除失败

3.2.3 静态初始化队列

```c
bool tk_queue_init(tk_queue_t queue, uint8_t *pool, uint16_t poolsize, bool
keep_fresh);
```

参数	描述
queue	要初始化的队列对象
*pool	缓冲区首地址
poolsize	缓冲区长度
keep_fresh	是否为保持最新模式，**true**：保持最新；**false**：默认(存满不能再存)
返回值	**true**：初始化成功；**false**：初始化失败

队列创建示例：

```c
int main(int argc, char *argv[])
{
    /* 定义一个循环队列 */
    struct tk_queue queue;
    /* 定义循环队列缓冲区 */
    uint8_t queue_pool[100];
    /* 静态方式创建一个循环队列"queue",缓存区为queue_pool，大小为queue_pool的大小,模式为
保持最新 */
    if( tk_queue_init(&queue, queue_pool, sizeof(queue_pool), true) == false){
        printf("队列创建失败!\n");
    }
```

```
    /* ... */
    /* You can add your code under here. */
}
```

3.2.4 静态脱离队列

> **注意**: 会使缓存区脱离与队列的关联。必须为**静态**方式创建的队列对象。

```
bool tk_queue_detach(tk_queue_t queue);
```

参数	描述
queue	要脱离的队列对象
返回值	**true**：脱离成功；**false**：脱离失败

3.2.5 判断队列是否为空

```
bool tk_queue_empty(tk_queue_t queue);
```

参数	描述
queue	要查询的队列对象
返回值	**true**：空；**false**：不为空

3.2.6 判断队列是否已满

```
bool tk_queue_full(tk_queue_t queue);
```

参数	描述
queue	要查询的队列对象
返回值	**true**：满；**false**：不为满

3.2.7 向队列压入(入队)1字节数据

```
bool tk_queue_push(tk_queue_t queue, uint8_t val);
```

参数	描述
queue	要压入的队列对象
val	压入值
返回值	**true**：成功；**false**：失败

3.2.8 从队列弹出(出队)1字节数据

```
bool tk_queue_pop(tk_queue_t queue, uint8_t *pval);
```

参数	描述
queue	要弹出的队列对象
*pval	弹出值
返回值	**true**：成功；**false**：失败

3.2.9 查询队列当前数据长度

```
uint16_t tk_queue_curr_len(tk_queue_t queue);
```

参数	描述
queue	要查询的队列对象
返回值	队列数据当前长度

3.2.10 向队列压入(入队)多个字节数据

```
uint16_t tk_queue_push_multi(tk_queue_t queue, uint8_t *pval, uint16_t len);
```

参数	描述
queue	要压入的队列对象
*pval	压入数据首地址
len	压入数据长度
返回值	实际压入长度

3.2.11 从队列弹出(出队)多字节数据

```
uint16_t tk_queue_pop_multi(tk_queue_t queue, uint8_t *pval, uint16_t len);
```

参数	描述
queue	要弹出的队列对象
*pval	存放弹出数据的首地址
len	希望弹出的数据长度
返回值	实际弹出长度

3.3 Timer 软件定时器API函数

> 以下为详细API说明及简要示例程序，综合demo可查看tk_timer_samples.c示例。

3.3.1 软件定时器功能初始化

> **注意**：此函数在使用定时器功能最初调用，目的是创建定时器列表头结点，和配置tick获取回调函数。

```
bool tk_timer_func_init(uint32_t (*get_tick_func)(void));
```

参数	描述
get_tick_func	获取系统tick回调函数
返回值	**true**：初始化成功；**false**：初始化失败

3.3.2 动态创建定时器

> **注意**：当配置**TOOLKIT_USING_TIMER**后，才能使用此函数。此函数需要用到**malloc**。

```
tk_timer_t tk_timer_create(tk_timer_timeout_callback *timeout_func);
```

参数	描述
timeout_func	定时器超时回调函数，不使用可配置为**NULL**
返回值	创建的定时器对象(**NULL**为创建失败)

定时器创建示例：：

```
/* 定义获取系统tick回调函数 */
uint32_t get_sys_tick(void)
{
    return tick;
}

/* 定时器超时回调函数 */
void timer_timeout_callback(tk_timer_t timer)
{
    printf("timeout_callback: timer timeout:%ld\n", get_sys_tick());
}

int main(int argc, char *argv[])
{
    /* 初始化软件定时器功能，并配置tick获取回调函数*/
    tk_timer_func_init(get_sys_tick);

    /* 定义定时器指针 */
    tk_timer_t timer = NULL;
    /* 动态方式创建timer,并配置定时器超时回调函数 */
    timer = tk_timer_create((tk_timer_timeout_callback
*)timer_timeout_callback);
    if (timer == NULL)
    {
        printf("定时器创建失败!\n");
        return 0;
    }
    /* ... */
    /* You can add your code under here. */
    return 0;
```

```
}
```

3.3.3 动态删除定时器

> 当配置**TOOLKIT_USING_TIMER**后，才能使用此函数。此函数需要用到**free**。必须为**动态**方式创建的定时器对象。

```
bool tk_timer_delete(tk_timer_t timer);
```

参数	描述
timer	要删除的定时器对象
返回值	**true**：删除成功；**false**：删除失败

3.3.4 静态初始化定时器

```
bool tk_timer_init(tk_timer_t timer, tk_timer_timeout_callback *timeout_func);
```

参数	描述
timer	要初始化的定时器对象
timeout_func	定时器超时回调函数，不使用可配置为**NULL**
返回值	**true**：创建成功；**false**：创建失败

队列创建示例：

```
/* 定义获取系统tick回调函数 */
uint32_t get_sys_tick(void)
{
    return tick;
}

/* 定时器超时回调函数 */
void timer_timeout_callback(tk_timer_t timer)
{
    printf("timeout_callback: timer timeout:%ld\n", get_sys_tick());
}

int main(int argc, char *argv[])
{
    /* 定义定时器timer */
    struct tk_timer timer;
    bool result = tk_timer_init( &timer,(tk_timer_timeout_callback
*)timer_timeout_callback);
    if (result == NULL)
    {
        printf("定时器创建失败!\n");
        return 0;
    }
    /* ... */
    /* You can add your code under here. */
```

```
    return 0;
}
```

3.3.5 静态脱离定时器

> **注意**: 会将timer从定时器链表中移除。必须为**静态**方式创建的定时器对象。

```
bool tk_timer_detach(tk_timer_t timer);
```

参数	描述
timer	要脱离的定时器对象
返回值	**true**：脱离成功；**false**：脱离失败

3.3.6 定时器启动

```
bool tk_timer_start(tk_timer_t timer, tk_timer_mode mode, uint16_t delay_tick);
```

参数	描述
timer	要启动的定时器对象
mode	工作模式，**单次：** *TIMER_MODE_SINGLE*；**循环：** *TIMER_MODE_LOOP*
delay_tick	定时器时长(单位tick)
返回值	**true**：启动成功；**false**：启动失败

3.3.7 定时器停止

```
bool tk_timer_stop(tk_timer_t timer);
```

参数	描述
timer	要停止的定时器对象
返回值	**true**：停止成功；**false**：停止失败

3.3.8 定时器继续

```
bool tk_timer_continue(tk_timer_t timer);
```

参数	描述
timer	要继续的定时器对象
返回值	**true**：继续成功；**false**：继续失败

3.3.9 定时器重启

> **注意**：重启时长为最后一次启动定时器时配置的时长。

```
bool tk_timer_restart(tk_timer_t timer);
```

参数	描述
timer	要重启的定时器对象
返回值	**true**：重启成功；**false**：重启失败

3.3.10 获取定时器模式

```
tk_timer_mode tk_timer_get_mode(tk_timer_t timer);
```

参数	描述
timer	要获取的定时器对象
返回值	定时器模式

定时器模式	描述
TIMER_MODE_SINGLE	单次模式
TIMER_MODE_LOOP	循环模式

3.3.11 获取定时器状态

```
tk_timer_state tk_timer_get_state(tk_timer_t timer);
```

参数	描述
timer	要获取的定时器对象
返回值	定时器状态

定时器模式	描述
TIMER_STATE_RUNNING	运行状态
TIMER_STATE_STOP	停止状态
TIMER_STATE_TIMEOUT	超时状态

3.3.12 定时器处理

```
bool tk_timer_loop_handler(void);
```

参数	描述
返回值	**true**：正常；**false**：异常，在调用此函数前，未初始化定时器功能“*tk_timer_func_init*”

> **注意**：tk_timer_loop_handler函数要不断的循环调用。

3.3.13 超时回调函数

函数原型：

```
typedef void (*tk_timer_timeout_callback)(struct tk_timer *timer);
```

> **说明**：超时回调函数可定义多个，即一个定时器对应一个回调函数，也可多个定时器对应一个回调函数。

- **一对一**

```
/* 定义两个回调函数，对应定时器timer1和timer2 */
void timer1_timeout_callback(tk_timer_t timer){
    printf("定时器1超时!\n");
}
void timer2_timeout_callback(tk_timer_t timer){
    printf("定时器2超时!\n");
}
/* 创建两个定时器,配置单独超时回调函数 */
timer1 = tk_timer_create((tk_timer_timeout_callback
*)timer1_timeout_callback);
timer2 = tk_timer_create((tk_timer_timeout_callback
*)timer2_timeout_callback);
```

- **多对一**

```
/* 定时器timer1和timer2共用一个回调函数，在回调函数做区分 */
void timer_timeout_callback(tk_timer_t timer){
    if (timer == timer1)
        printf("定时器1超时!\n");
    else if (timer == timer2)
        printf("定时器2超时!\n");
}
/* 创建两个定时器,使用相同的超时回调函数 */
timer1 = tk_timer_create((tk_timer_timeout_callback
*)timer_timeout_callback);
timer2 = tk_timer_create((tk_timer_timeout_callback
*)timer_timeout_callback);
```

3.4 Event 事件集API函数

> 以下为详细API说明及简要示例程序，综合demo可查看tk_event_samples.c示例。

3.4.1 动态创建一个事件

> **注意**：当配置**TOOLKIT_USING_EVENT**后，才能使用此函数。此函数需要用到**malloc**。

```
tk_event_t tk_event_create(void);
```

参数	描述
返回值	创建的事件对象(**NULL**为创建失败)

3.4.2 动态删除一个事件

> 当配置**TOOLKIT_USING_TIMER**后，才能使用此函数。此函数需要用到**free**。必须为**动态**方式创建的事件对象。

```
bool tk_event_delete(tk_event_t event);
```

参数	描述
event	要删除的事件对象
返回值	**true**：删除成功；**false**：删除失败

3.4.3 静态初始化一个事件

```
bool tk_event_init(tk_event_t event);
```

参数	描述
event	要初始化的事件对象
返回值	**true**：创建成功；**false**：创建失败

3.4.4 发送事件标志

```
bool tk_event_send(tk_event_t event, uint32_t event_set);
```

参数	描述
event	发送目标事件对象
event_set	事件标志，每个标志占1Bit，发送多个标志可“|”
返回值	**true**：发送成功；**false**：发送失败

3.4.5 接收事件

```
bool tk_event_recv(tk_event_t event, uint32_t event_set, tk_event_option option,
uint32_t *recved);
```

参数	描述
event	接收目标事件对象
event_set	感兴趣的标志，每个标志占1Bit，多个标志可“|”
option	操作，**标志与**：TK_EVENT_OPTION_AND; **标志或**：TK_EVENT_OPTION_OR; **清除标志**:TK_EVENT_OPTION_CLEAR
返回值	**true**：发送成功；**false**：发送失败