### 4.4 Секвенсор

Секвенсор последовательно выполняет зарегистрированные функции. Он имеет следующие особенности:

- поддерживает до 32 функций
- запрашивает выполнение функций
- включает / отключает выполнение функции
- предоставляет интерфейс блокировки на основе приема события.

Секвенсор обеспечивает простую функцию фонового планирования. Он обеспечивает ловушку для реализации безопасного режима энергосберегающего режима (без потери событий), когда секвенсор не работает.

есть незавершенные задачи, которые нужно выполнить. Он также предоставляет приложению эффективный механизм ожидания определенного события перед тем, как двигаться дальше. Когда секвенсор

ожидая определенного события, он предоставляет ловушку, при которой приложение может либо войти в режим низкого энергопотребления, либо выполнить другой код.

#### 4.4.1 Реализация

Для использования секвенсора приложение должно:

- установить максимальное количество поддерживаемых функций (это делается путем определения значения для UTIL_SEQ_CONF_TASK_NBR)
- зарегистрировать функцию, которая будет поддерживаться секвенсором, с помощью UTIL_SEQ_RegTask ()
- запустить секвенсор, вызвав UTIL_SEQ_Run (), чтобы запустить цикл while в фоновом режиме
- вызвать UTIL_SEQ_SetTask (), когда функция должна быть выполнена.

#### 4.4.2 Интерфейс

*Таблица 3. Функции интерфейса*

| Функция | Описание |
|---|---|
| void UTIL_SEQ_Idle (void); | Вызывается (в критическом разделе - PRIMASK), когда нечего выполнять. |
| void UTIL_SEQ_Run (UTIL_SEQ_bm_t mask_bm) | Запрашивает, чтобы секвенсор выполнял функции, ожидающие обработки и включенные в маске mask_bm. |
| void UTIL_SEQ_RegTask (UTIL_SEQ_bm_t task_id_bm, uint32_t flags, void (*task)( void )) | Регистрирует функцию (задачу), связанную с сигналом (task_id_bm) в секвенсоре. В task_id_bm должен быть установлен один бит. |
| void UTIL_SEQ_SetTask (UTIL_SEQ_bm_t task_id_bm, UTIL_SEQ_bm_t task_prio) | Запрашивает выполнение функции, связанной с task_id_bm. Task_prio оценивается секвенсором только после завершения функции. Если несколько функций ожидают одновременного выполнения, выполняется функция с наивысшим приоритетом (0). |
| void UTIL_SEQ_PauseTask (UTIL_SEQ_bm_t task_id_bm) | Отключает секвенсор для выполнения функции, связанной с task_id_bm. |
| void UTIL_SEQ_ResumeTask (UTIL_SEQ_bm_t task_id_bm) | Позволяет секвенсору выполнять функцию, связанную с task_id_bm. |

| | |
|---|---|
| void UTIL_SEQ_WaitEvt (UTIL_SEQ_bm_t evt_id_bm) | Запрашивает секвенсор дождаться определенного события evt_id_bm и не возвращается, пока событие не будет установлено с помощью UTIL_SEQ_SetEvt (). |
| void UTIL_SEQ_SetEvt (UTIL_SEQ_bm_t evt_id_bm) | Уведомляет секвенсор о том, что произошло событие evt_id_bm (событие должно быть запрошено первым). |
| void UTIL_SEQ_EvtIdle (UTIL_SEQ_bm_t task_id_bm, UTIL_SEQ_bm_t evt_waited_bm) | Вызывается, когда секвенсор ожидает определенного события. |
| void UTIL_SEQ_ClrEvt (UTIL_SEQ_bm_t evt_id_bm) | Удаляет отложенное событие. |
| UTIL_SEQ_bm_t UTIL_SEQ_IsEvtPend (void) | Возвращает evt_id_bm ожидающего события. |

### 4.4.3 Подробности интерфейса и поведение

Секвенсор представляет собой упаковку циклов while для вызова функций по запросу пользователя:

```
while(1)
{
	if(task_id1)
	{
		task_id1 = 0;
		Fct1();
	}

	if (task_id2)
	{
		task_id2= 0;
		Fct2();
	}

	__disable_irq();
	If (! (task_id1|| task_id2))
	{
		UTIL_SEQ_Idle();
	}
	__enable_irq();
}
```

**void UTIL_SEQ_Run (UTIL_SEQ_bm_t mask_bm)**

Реализует тело цикла while (1). Параметр mask_bm - это список функций, которые может выполнять секвенсор. Каждая функция связана с одним битом в этой mask_bm. В конце запуска этот API должен быть вызван в цикле while (1) с mask_bm = (~ 0), чтобы позволить секвенсору выполнить любую ожидающую функцию.

**void UTIL_SEQ_Idle (void)**

Вызывается в критической секции (устанавливается битом CortexM PRIMASK - все прерывания маскируются), когда у секвенсора нет функции для выполнения. Здесь приложение должно перейти в режим пониженного энергопотребления.

**void UTIL_SEQ_RegTask (UTIL_SEQ_bm_t task_id_bm, uint32_t flags, void (*task)( void ))**

Информирует секвенсор о необходимости добавить функциональную задачу, связанную с флагом task_id_bm, в его цикл while.

**void UTIL_SEQ_SetTask (UTIL_SEQ_bm_t task_id_bm , UTIL_SEQ_bm_t task_prio)**

Устанавливает флаг task_id_bm для планировщика для вызова связанной функции.

Task_prio оценивается секвенсором, когда ему нужно решить, какую функцию вызывать следующей. Это можно сделать только после завершения выполнения текущей функции. Когда у нескольких функций установлен свой флаг, выполняется функция с более высоким приоритетом (0 - наивысший). Этот API может быть вызван несколько раз, прежде чем функция будет фактически выполнена с другим приоритетом. В этом случае секвенсор записывает наивысший приоритет. Каким бы ни было количество вызовов API перед выполнением функции, секвенсор запускает связанную функцию только один раз.

**void UTIL_SEQ_PauseTask (UTIL_SEQ_bm_t task_id_bm)**

Сообщает секвенсору не выполнять функцию, связанную с флагом task_id_bm, даже если он установлен. Если API UTIL_SEQ_SetTask () вызывается после UTIL_SEQ_PauseTask (), запрос записывается, но функция не выполняется. Маска, связанная с UTIL_SEQ_PauseTask (), не зависит от маски, связанной с void UTIL_SEQ_Run (UTIL_SEQ_bm_t mask_bm).

Функция может быть выполнена, только если ее флаг установлен и включен в обеих масках (случай по умолчанию).

**void UTIL_SEQ_ResumeTask (UTIL_SEQ_bm_t task_id_bm)**

Отменяет запрос, сделанный UTIL_SEQ_PauseTask (). Если этот API вызывается при отсутствии запроса UTIL_SEQ_PauseTask (), он не имеет никакого эффекта.

**void UTIL_SEQ_WaitEvt (UTIL_SEQ_bm_t evt_id_bm)**

Когда этот API вызывается, он не возвращается, пока не будет установлен соответствующий сигнал evt_id_bm. Необходимо установить только один бит в 32-битном значении evt_id_bm. Пока секвенсор ожидает этого события, он вызывает UTIL_SEQ_EvtIdle () в цикле while для события evt_id_bm. Это необходимо использовать для замены всего кода, в котором выполняется опрос флага перед тем, как двигаться дальше.

**void UTIL_SEQ_SetEvt (UTIL_SEQ_bm_t evt_id_bm)**

Должен вызываться только тогда, когда UTIL_SEQ_WaitEvt () уже был вызван. Он устанавливает сигнал evt_id_bm, которого ожидает функция UTIL_SEQ_WaitEvt (). Вызов этого API перед функцией UTIL_SEQ_WaitEvt () вызывает немедленный возврат вызова UTIL_SEQ_WaitEvt (), поскольку флаг уже установлен.

**void UTIL_SEQ_EvtIdle (UTIL_SEQ_bm_t task_id_bm, UTIL_SEQ_bm_t evt_waited_bm)**

Вызывается, когда API void UTIL_SEQ_WaitEvt () ожидает установки сигнала с помощью UTIL_SEQ_SetEvt ().

Этот API слабо реализован в секвенсоре для вызова UTIL_SEQ_Run (0), что означает, что в ожидании этого события функция UTIL_SEQ_Idle () позволяет системе перейти в режим низкого энергопотребления, ожидая флага.

Приложение может реализовать этот API для передачи параметров, отличных от 0, в UTIL_SEQ_Run (mask_bm). Каждый бит, установленный в 1 в mask_bm, запрашивает у секвенсора выполнение функции, связанной с этим флагом, когда он

установлен с помощью UTIL_SEQ_SetTask (). Это означает, что когда вызывается функция UTIL_SEQ_WaitEvt (), пока она ожидает возврата запрошенного события, она может либо выполнить немаскированные функции, когда установлен их флаг, либо вызвать UTIL_SEQ_Idle (), если ни одна из задач не ожидает выполнения секвенсором.

**void UTIL_SEQ_ClrEvt (UTIL_SEQ_bm_t evt_id_bm)**

Этот API может быть вызван, когда в некоторых приложениях необходимо вызвать API UTIL_SEQ_WaitEvt (), когда Evt уже установлен. В этом случае необходимо очистить Evt.

**UTIL_SEQ_bm_t UTIL_SEQ_IsEvtPend (void)**

Этот API возвращает ожидаемое в настоящее время Evt. Когда несколько UTIL_SEQ_WaitEvt () вложены, он возвращает последний, что означает тот, который заставляет более глубокий UTIL_SEQ_WaitEvt () возвращаться к вызывающему.

### 7.6.2 Как добавить задачу в секвенсор

• Объявите task ID - app_conf.h -

```
/**< Добавьте в этот список все задачи, которые могут отправлять команду ACI / HCI.  */

typedef enum
{
        CFG_TASK_ADV_CANCEL_ID,
        CFG_TASK_SW1_BUTTON_PUSHED_ID,
        CFG_TASK_HCI_ASYNCH_EVT_ID,
        CFG_LAST_TASK_ID_WITH_HCICMD,  /**< Shall be LAST in the list */
} CFG_Task_Id_With_HCI_Cmd_t;

/**<Добавьте в этот список все задачи, которые никогда не отправляют команду ACI / HCI.  */

typedef enum
{
        CFG_FIRST_TASK_ID_WITH_NO_HCICMD = CFG_LAST_TASK_ID_WITH_HCICMD - 1,
        /**< Будет ПЕРВЫМ в списке  */
        CFG_TASK_SYSTEM_HCI_ASYNCH_EVT_ID,

        CFG_LAST_TASK_ID_WITHO_NO_HCICMD
        /**< Должен быть ПОСЛЕДНИМ в списке  */
} CFG_Task_Id_With_NO_HCI_Cmd_t;

#define UTIL_SEQ_CONF_TASK_NBR    CFG_LAST_TASK_ID_WITHO_NO_HCICMD
```

•Зарегистрируйте задачу с функцией обратного вызова - “Cancel Advertising” - app_ble.c

```
SCH_RegTask(CFG_TASK_ADV_CANCEL_ID, Adv_Cancel);
```

• Запустите задачу с приоритетом - app_ble.c

```
SCH_SetTask(1 << CFG_TASK_ADV_CANCEL_ID, CFG_SCH_PRIO_0);
```

### 4.5 Сервер таймера

Сервер таймера имеет следующие особенности:

- До 255 виртуальных таймеров в зависимости от доступного объема ОЗУ.
- Одиночный снимок и повторяющийся режим
- Останавливает виртуальный таймер и перезапускает его с другим значением тайм-аута.
- Удаляет таймер
- Таймаут от 1 до 232 - 1 тик

Сервер таймера предоставляет несколько виртуальных таймеров, совместно использующих таймер пробуждения RTC. Каждый виртуальный таймер может быть определен как однократный или повторяющийся. Когда повторяющийся таймер подходит к концу цикла, пользователь получает уведомление, и виртуальный таймер автоматически перезапускается с тем же тайм-аутом. Когда таймер одиночного выстрела заканчивается, пользователь получает уведомление, и виртуальный таймер устанавливается в состояние ожидания (что означает, что он остается зарегистрированным и может быть перезапущен в любое время). Пользователь может остановить виртуальный таймер и перезапустить его с другим значением тайм-аута. Когда виртуальный таймер больше не нужен, пользователь должен удалить его, чтобы освободить слот на сервере таймера.

Сервер таймера можно использовать одновременно с календарем.

#### 4.5.1 Реализация

Чтобы использовать сервер таймера, приложение должно:

- Настройте RTC IP. Если в приложении требуется календарь, конфигурация RTC должна быть совместима с требованиями настроек календаря. Когда календарь не используется, RTC можно оптимизировать только для использования сервера таймера.
- Инициализировать сервер таймера с помощью HW_TS_Init ().
- Реализовать HW_TS_RTC_Int_AppNot () (необязательно). Если не реализован, обратный вызов таймера вызывается в контексте обработчика прерывания RTC.
- Создайте виртуальный таймер с HW_TS_Create ().
- Используйте виртуальный таймер с HW_TS_Stop (), HW_TS_Start ().
- Удалите виртуальный таймер, когда он не нужн, с помощью HW_TS_Delete ().

4.5.2 Интерфейс

*Таблица 4. Функции интерфейса*

| Функция | Описание |
|---|---|
| void HW_TS_Init (HW_TS_InitMode_t TimerInitMode, RTC_HandleTypeDef *hrtc); | Инициализирует сервер таймера. |
| HW_TS_ReturnStatus_t HW_TS_Create (uint32_t TimerProcessID, uint8_t *pTimerId, HW_TS_Mode_t TimerMode, HW_TS_pTimerCb_t pTimerCallBack) | Создает виртуальный таймер. |
| void HW_TS_Stop (uint8_t TimerID) | Останавливает виртуальный таймер. |

| | |
|---|---|
| void HW_TS_Start (uint8_t TimerID, uint32_t timeout_ticks) | Запускает виртуальный таймер. |
| void HW_TS_Delete (uint8_t TimerID) | Удаляет виртуальный таймер. |
| void HW_TS_RTC_Wakeup_Handler(void) | Обработчик сервера таймера, который будет вызываться из обработчика прерывания RTC. |
| uint16_t HW_TS_RTC_ReadLeftTicksToCount (void) | Возвращает количество тактов для подсчета до следующего прерывания. |
| void HW_TS_RTC_Int_AppNot (uint32_t TimerProcessID, uint8_t TimerID, HW_TS_pTimerCb_t pTimerCallBack) | Сообщает приложению, что срок действия виртуального таймера истек. |
| void HW_TS_RTC_CountUpdated_AppNot(void) | Сообщает приложению, что количество тактов до следующего прерывания было обновлено сервером таймера. |

### 4.5.3 Подробности интерфейса и поведение

Сервер таймера предоставляет виртуальные таймеры, которые работают, когда система находится в режиме пониженного энергопотребления вплоть до режима ожидания.

**void HW_TS_Init (HW_TS_InitMode_t TimerInitMode, RTC_HandleTypeDef * hrtc)**

Эта команда инициализирует сервер таймера на основе конфигурации IP RTC, которая должна быть сделана заранее.

TimerInitMode выбирает режим загрузки сервера таймера. Если режим ожидания поддерживается и устройство выходит из режима ожидания, установите для TimerInitMode значение hw_ts_InitMode_Limited, чтобы контекст сервера таймера не сбрасывался. В противном случае для TimerInitMode необходимо установить значение hw_ts_InitMode_Full для выполнения полной инициализации.

hrtc - это дескриптор Cube HAL RTC.

**HW_TS_ReturnStatus_t HW_TS_Create (uint32_t TimerProcessID, uint8_t * pTimerId, HW_TS_Mode_t TimerMode, HW_TS_pTimerCb_t pTimerCallBack)**

pTimerId. Это идентификатор, возвращаемый сервером таймера вызывающей стороне, который необходимо использовать для остановки / запуска / удаления созданного таймера.

TimerMode. Режим таймера может быть как одиночным, так и повторным. В режиме одиночного выстрела таймер останавливается по истечении времени ожидания. В повторяющемся режиме он перезапускается с тем же ранее запрограммированным значением при каждом тайм-ауте. Этот режим фиксируется при создании таймера. Чтобы изменить режим, необходимо удалить таймер и создать новый. Обратите внимание, что в этом случае новый выделенный pTimerId может быть другим.

pTimerCallBack. Обратный вызов пользователя по таймауту.

TimerProcessID. Это определяется пользователем и, как ожидается, будет использоваться в HW_TS_RTC_Int_AppNot (). Когда таймер создан, только вызывающая сторона знает присвоенный идентификатор. TimerProcessID возвращается в HW_TS_RTC_Int_AppNot () с pTimerCallBack, так что соответствующее решение может быть принято при реализации HW_TS_RTC_Int_AppNot ().

**void HW_TS_Stop (uint8_t TimerID)** Останавливает таймер TimerID. Это не действует, если таймер не работает. Таймер TimerID должен быть создан. Когда таймер оста-

новлен, TimerID остается выделенным на сервере таймера, так что тот же таймер (с тем же TimerMode и тем же pTimerCallBack) может быть перезапущен с другим значением.

**void HW_TS_Start (uint8_t TimerID, uint32_t timeout_ticks)**

Запускает таймер TimerID со значением timeout_ticks. Значение timeout_ticks зависит от конфигурации RTC IP. Если TimerID уже запущен, он сначала останавливается на сервере таймера и перезапускается с новым значением timeout_ticks.

**void HW_TS_Delete (uint8_t TimerID)**

Удаляет TimerID с сервера таймера. TimerID может быть назначен новому виртуальному таймеру. Этот API может быть вызван для работающего TimerID. В этом случае он сначала останавливается, а затем удаляется.

**void HW_TS_RTC_Wakeup_Handler (void )**

Этот обработчик прерывания должен вызываться приложением в обработчике прерывания RTC. Этот обработчик сбрасывает все необходимые флажки состояния в периферийных устройствах RTC и EXTI.

**uint16_t HW_TS_RTC_ReadLeftTicksToCount (void)**

Этот API возвращает количество тактов, которое осталось подсчитать до того, как сервер таймера сгенерирует прерывание. Его можно использовать, когда системе необходимо войти в режим низкого энергопотребления и решить, какой режим низкого энергопотребления применить, в зависимости от того, когда ожидается следующее пробуждение.

Когда таймер отключен (в списке нет таймера), он возвращает 0xFFFF.

**void HW_TS_RTC_Int_AppNot (uint32_t TimerProcessID,**
**uint8_t TimerID, HW_TS_pTimerCb_t pTimerCallBack)**

Этот API должен быть реализован пользовательским приложением.

Он уведомляет приложение об истечении таймера. Этот API работает в контексте прерывания пробуждения RTC, и приложение может предпочесть вызов pTimerCallBack в качестве фоновой задачи в зависимости от того, сколько кода выполняется в pTimerCallBack. Пока TimerID известен только вызывающей стороне, TimerProcessID может использоваться для определения того, к какому модулю принадлежит этот pTimerCallBack, и приложение может оценить, может ли он быть вызван в контексте прерывания пробуждения RTC или нет.

**void HW_TS_RTC_CountUpdated_AppNot (void)**

Этот API должен быть реализован пользовательским приложением.

Этот API уведомляет приложение об обновлении счетчика. Ожидается, что это будет использоваться вместе с API HW_TS_RTC_ReadLeftTicksToCount (). Счетчик мог быть обновлен с момента последнего вызова HW_TS_RTC_ReadLeftTicksToCount () и до перехода в режим низкого энергопотребления. Это уведомление позволяет приложению разрешить состояние гонки и повторно оценить значение счетчика перед переходом в режим низкого энергопотребления.

### 7.6.3 Как использовать сервер таймера

- Создать таймер с функцией обратного вызова

```
/**
    * Создайте таймер для выключения светодиодного переключателя
    */
```

```
HW_TS_Create (CFG_TIM_PROC_ID_ISR,
&(BleApplicationContext.SwitchOffGPIO_timer_Id), hw_ts_SingleShot, Switch_OFF_GPIO);
```

• Запустить таймер с таймаутом

```
HW_TS_Start(BleApplicationContext.SwitchOffGPIO_timer_Id, (uint32_t)LED_ON_TIMEOUT);
```

•Остановить таймер

```
HW_TS_Stop (BleApplicationContext.SwitchOffGPIO_timer_Id);
```

• Пример функции обратного вызова

```
static void Switch_OFF_GPIO()
{
 BSP_LED_Off(LED_GREEN);
}
```

### 4.6 Менеджер низкого энергопотребления

Менеджер с низким энергопотреблением предоставляет простой интерфейс для приема входных данных от 32 различных пользователей и вычисляет минимально возможный режим энергопотребления, который может использовать система. Он также обеспечивает привязку к приложению перед входом или выходом из режима пониженного энергопотребления.

Менеджер низкого энергопотребления предоставляет следующие функции:

- До 32 пользователей
- Режим остановки и режим выключения (режим ожидания и выключение).
- Выбор режима пониженного энергопотребления
- Выполнение в режиме пониженного энергопотребления
- Обратный вызов при входе или выходе из режима пониженного энергопотребления
- Режим выполнения не поддерживается, когда приложение должно оставаться в этом режиме, оно не должно вызывать UTIL_LPM_EnterModeSelected ().

#### 4.6.1 Реализация

Менеджер низкого энергопотребления может обрабатывать до 32 пользователей с различными запросами режима низкого энергопотребления.

Чтобы использовать диспетчер низкого энергопотребления, приложение должно:

- создать идентификатор пользователя
- вызвать либо UTIL_LPM_SetOffMode (), либо UTIL_LPM_SetStopMode () в любое время с определенным идентификатором пользователя, чтобы установить запрошенный режим низкого энергопотребления
- вызвать void UTIL_LPM_EnterLowPower () в фоновом режиме.

### 4.6.2 Интерфейс

Таблица 5. Интерфейсные функции

| Функция | Описание |
|---|---|
| UTIL_LPM_ModeSelected_t UTIL_LPM_ReadModeSel(void) | Возвращает выбранный режим пониженного энергопотребления, который будет применен. |
| UTIL_LPM_SetOffMode(UTIL_LPM_bm_t lpm_id_bm, UTIL_LPM_State_t state) | Включает или отключает режим Off для любого пользователя в любое время. |
| void UTIL_LPM_SetStopMode(UTIL_LPM_bm_t lpm_id_bm, UTIL_LPM_State_t state) | Включает или отключает режим остановки для любого пользователя в любое время. |
| void UTIL_LPM_EnterLowPower(void) | Переход в выбранный режим пониженного энергопотребления. |
| void UTIL_LPM_EnterSleepMode(void) | API вызывается перед переходом в спящий режим. |
| void UTIL_LPM_ExitSleepMode(void) | API вызывается при выходе из спящего режима. |
| void UTIL_LPM_EnterStopMode(void) | API вызывается перед переходом в режим остановки. |
| void UTIL_LPM_ExitStopMode(void) | API вызывается при выходе из режима остановки. |
| void UTIL_LPM_EnterOffMode(void) | API вызывается перед переходом в выключенный режим. |
| void UTIL_LPM_ExitOffMode(void) | API вызывается при выходе из выключенного режима. Это вызывается только в том случае, если MCU не перешел в режим Off, как ожидалось. |