RKB Global

BDSLE Scheduler

Event-Based Operating System gOS

For Embrace+

v1.3.1r1

RKB Global Co. Ltd.

2014.01.23

Version Control

Revision	Date	Descriptions
v1.0	2013-10-12	Initial release
v1.3	2013-11-19	Updated for BDSLE v1.3
v1.3.1	2013-12-23	Updated to align with v1.3.1
v1.3.1r1	2014-01-23	Update to align with v1.3.1r1

Table of Contents

1. Introduction 1
2. How gOS works 1
3. How to use 2
3.1 Startup 2
3.2 Task 4
3.3 Event 7
3.4 Timer 10
3.5 Message 12
3.6 Memory 17
3.7 Power management 20
4. gOS demo project performing effect 21
4.1 App_01_task_add_related 22
4.2 App_02_event_related 22
4.3 App_03_msg_related 22
4.4 App_04_timer_related 22
4.5 App_05_lower_power_related 22

1. Introduction

gOS is a simple event-based operating system designed by BDE Technology to drive the BDSLE™ stack and the applications based on it. gOS has two main features. The first one is its scheduler, which schedules the tasks that are registered on gOS and based on the event triggered by user or timer. That is to say, if there is an event triggered, gOS will schedule the corresponding task. Because gOS is single-threaded and non-preemptive, the next task can't be scheduled until the current task has been scheduled. The second feature is memory management. The user can allocate the memory and free it by the APIs of gOS.

2. How gOS works

To understand how gOS works, it is very important and helpful to learn the task and event. A task can support zero or no more than 31 events. When gOS starts running, it checks the first task to know whether the task has events

triggered. And if there are events triggered, gOS will schedule the task. After scheduling the first task with events triggered, gOS continues to check the next task and do the same work. After all the tasks have been checked, gOS restarts from the first task.

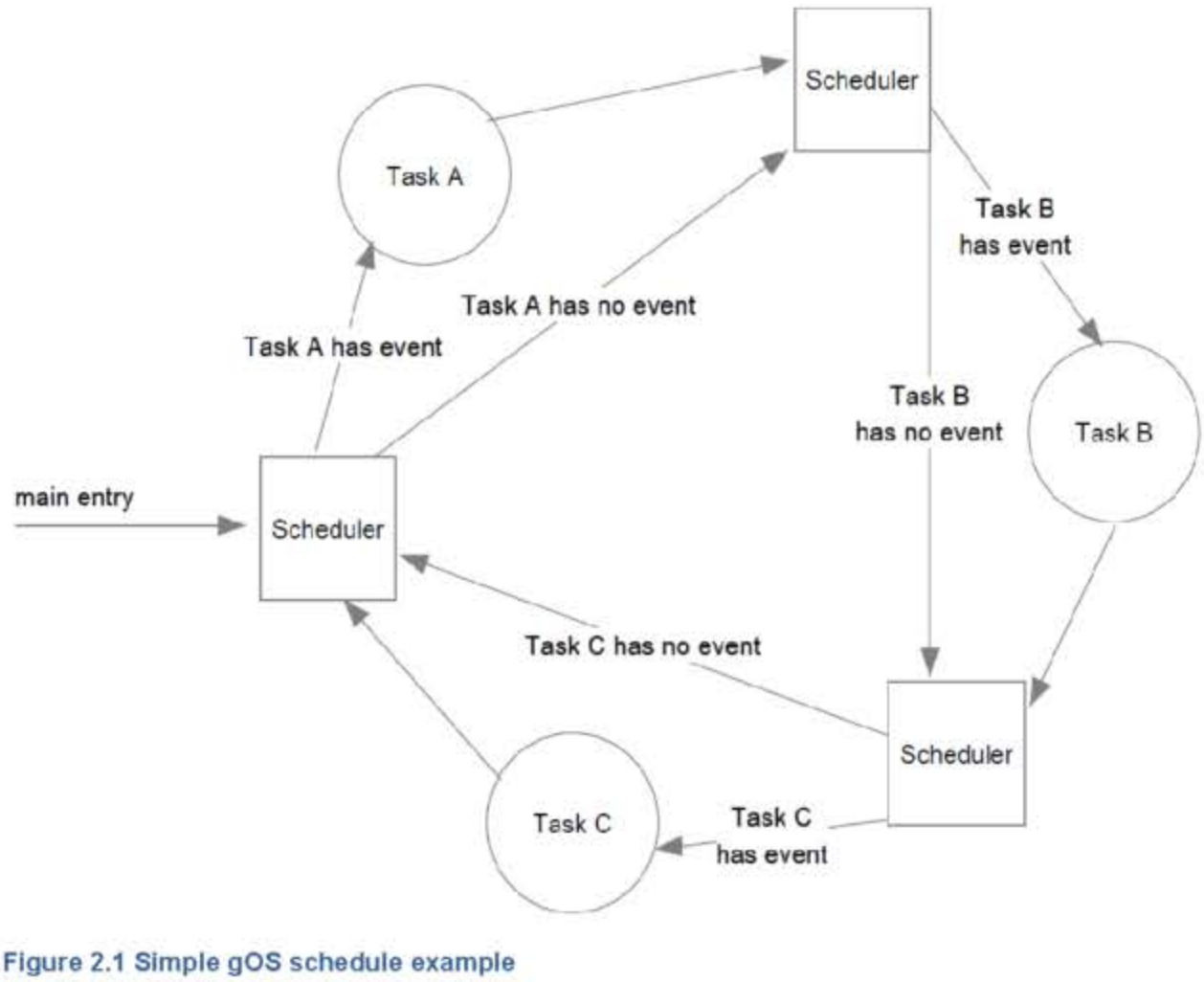

Figure 2.1 Simple gOS schedule example

From the figure above, we can understand how the tasks are scheduled by gOS and what the relationship between the task and events is. One point that needs to be mentioned is that the task with several events triggered will be scheduled only once. So the task event handler shall find out which events have been triggered and then handle the triggered events. After the task is scheduled, all the flags of the triggered events will be automatically cleared.

3. How to use

This chapter describes how to use gOS to develop applications. It is divided into seven parts, which includes Startup, Task, Event, Timer, Message, Memory, and Power Management.

3.1 Startup

This section describes the gOS startup procedure.

Before starting gOS, the following steps must be done:

Step 1: Register user tasks to gOS scheduler.

Step 2: Do some hardware related initializations.

Step 3: Set memory management method.

Step 4: Do gOS system level initialization.

Step 5: Start gOS system scheduler.

The following is a simple code for starting gOS:

```
int main()
{
    /**
     * gOS task creating function Task Initialization
     */
    gOS_DoCBsInit();
    /**
     * Initialize HAL Target
     */
    HAL_DoTargetInit();
    /**
     * Use default memory managing method, and set memory pool and its size.
     */
    gOS_SetMemMgr(0, 0, 0, gos_mem_poll, sizeof(gos_mem_poll));

    gOS_SystemInit();
    gOS_SystemStart(0);
    return 0;
}
```

NOTE: The order of step 1 to step 3 can be disrupted. But all the 3 steps should be done before the invocating of **gOS_SystemInit()**.

gOS_SystemInit() is the initializing routine of gOS internal tasks managing list, timers managing list, gOS message managing list. Because each task would be specified an task id after **gOS_SystemInit()** is invocated, any gOS API with a task_id parameter shall not be invocated before this API.

Calling **gOS_SystemStart()** function starts the gOS scheduler. If it is called with parameter 0，the polling loop inside this API will never be interrupted. Both the task list and timer list polling operations are performing in dead loop inside the **gOS_SystemStart()**. If it is called with parameter 1, the inside polling loop will be broken after it performs once.

NOTE: Invocating **gOS_SystemStart()** with parameter 0 means target

application takes gOS as its main scheduler. In this case, target application can use the internal power management method of gOS. Invocating **gOS_SystemStart()** with parameter 1 means the target application may not use any OS or another scheduler to design its function. Therefore, target application shall use its own power management method.

3.2 Task

The task is the scheduling unit of gOS, so the application based on it shall register at least one task. This section describes how to register a task and how the task event handler is designed.

To register a gOS task, the following API shall be used:

```
BDE_STATUS gOS_AddTask(CONST ptrFn_TaskInit ptr_fn_init,
                       CONST ptrFn_TaskEventHandler ptr_fn_event_handler,
                       uint8 task_priority);
```

From the API declaration above, it's mandatory for each task to implement 2 functions. The 2 functions are task initialization routine and task events processing routine.

NOTE: Each time a new task is registered, gOS allocates a memory block from gOS memory pool to maintain the corresponding task. And the memory block will never be released.

The task initialization function must be declared as the following prototype:

```
void Task_Init(uint8 task_id);
```

In the prototype above, the task_id parameter value is assigned after **gOS_SystemStart()** is invocated. Target application shall define a global variable to store the assigned task id. Each task initialization function is called inside **gOS_SystemStart()**.

The task events processing function must be declared as the following prototype:

```
void Task_EventProcess(uint8 task_id, uint32 events);
```

The task_id parameter will be assigned when the task initialization function is invocated. The events parameter indicates what events happen. For more events information, see Section 3.3.

NOTE: Because gOS is a single-threaded event scheduler, each task must not occupy the MCU executing permission for a long time. A dead loop is never

permitted to exist in a task implementation. For example the following code is problematic.

```
void Task_EventProcess(uint8 task_id, uint32 events)
{
  uint8 i = 0
  if (events & SOME_EVENT)
  {
    for (i = 0; i < 100; i++) // This loop never breaks
    {
      if (i == 90)
      {
        i = 0;
      }
    }
  }
}
```

The following source code demonstrates how to register a user-implemented task.

```
void gOS_DoCBsInit(void)
{
  /**
   * Tasks Creating Function register
   */
  gOS_SetDriverFunc((void(*)(void))gOS_CreateTasksList, gOS_ADD_TASKLIST_CB);
}
void gOS_CreateTasksList(void)
{
  gOS_AddTask(LED1_InitTask,   LED1_EventProcessor, gOS_NORMAL_PRIORITY);
}
```

gOS_DoCBsInit() is called when gOS starts. See Section 3.1. This function is user-implemented. In this function implementation, a callback function named **gOS_CreateTasksList()** is registered by using the **gOS_SetDriverFunc()** API. Just as its name implies, the **gOS_CreateTasksList()** is used to register all user-defined tasks.

NOTE: gOS does not support registering and deleting tasks after running (after invocating **gOS_SystemStart()**).

The following is an example code for task definition and implementation. The code comes from the gOS demo project '*app_01_task_add_related*'. See its related project for more code.

```
static uint8 led1_task_id = INVALID_TASK_ID;


void LED1_InitTask(byte task_id)
{
  /**
   *  save the task_id that the gOS assigns
   */
  led1_task_id = task_id;
  /**
   *  Turn on the LED1
   */
  HAL_TurnOnLed1();
  /**
   *  Executed the LED1_FLASH_EVENT event immediately
   */
  gOS_EventSet(led1_task_id, LED1_FLASH_EVENT);
}

void LED1_EventProcessor(byte task_id, UINT32 events)
{
  /**
   *The event is bit-mask definition. So here do a '&' operation to determine whether
   * an event is triggered.
   *
   * If the LED1_FLASH_EVENT event happens
   */
  if (events & LED1_FLASH_EVENT)
  {
    /**
     * Toggle the LED1
     */
    HAL_ToggleLed1();

    /**
     * Set the LED1_FLASH_EVENT event happen LED1_FLASH_PERIOD ms later
     */
```

```
    gOS_StartTimer(task_id, LED1_FLASH_EVENT, LED1_FLASH_PERIOD);
  }
}

static uint8 led2_task_id = INVALID_TASK_ID;

void LED2_InitTask(uint8 task_id)
{
  /**
   *  save the task_id that the gOS assigns
   */
  led2_task_id = task_id;
  /**
   *  Turn on the LED2
   */
  HAL_TurnOnLed2();
   /**
   *  Executed the LED2_FLASH_EVENT event LED2_FLASH_PERIOD ms later
   */
  gOS_StartTimer(led2_task_id, LED2_FLASH_EVENT, LED2_FLASH_PERIOD);
}

void LED2_EventProcessor(uint8 task_id, uint32 events)
{
  /**
   * the LED2_FLASH_EVENT event has happened
   */
  if (events & LED2_FLASH_EVENT)
  {
    HAL_ToggleLed2();

    gOS_StartTimer(task_id, LED2_FLASH_EVENT, LED2_FLASH_PERIOD);
  }
}
```

For the code executing phenomenon above, see Section 4.1.

3.3 Event

Event is the kernel concept of gOS. According to event setting, gOS schedules

all its tasks. Each task has an internal uint32 variable to store the events bit-mask set by the following API:

```
BDE_STATUS gOS_EventSet(byte task_id, uint32 event_flag);
```

Each task has up to 32 events. And among the 32 events, one event (0x800000) is occupied by gOS message management. Thus, up to 31 events can be used in a task implementation.

When gOS scheduler is running, it polls the entire task list to check whether a task has events triggered. If some events are triggered, gOS scheduler will invocate the corresponding task events processing function to perform the user implementation code. See Section 2 for the scheduling method of gOS.

The following is a demonstration code about how to define gOS events:

```
#define  EVENT1        0x00000001
#define  EVENT2        0x00000002
#define  EVENT3        0x00000004
#define  EVENT4        0x01000000
#define  EVENT5        0x40000000
```

Because each task has its own variable to maintain its events scheduling, one bit-mask can be used in 2 or more tasks implementations. For example the following code is correct.

```
//... for task 1
#define LED_OFF_EVENT   0x1
void Task1_EventProccess(uint8 task_id, uint32 events)
{
  if (events & LED_OFF_EVENT)
  {
    HAL_TurnOffLed1();
  }
}

//... for task 2
#define LED_ON_EVENT   0x1
void Task1_EventProcess(uint8 task_id, uint32 events)
{
  if (events & LED_ON_EVENT)
  {
    HAL_TurnOnLed1();
```

```
  }
}
```

NOTE: Using **gOS_EventSet()** can set 1 or more events at a time. For example the following code shows 2 events processing logics are performing at a time in a task scheduling. We strongly recommend that DO NOT set more than 1 event at a time.

```
#define LED_OFF_EVENT   0x2
#define LED_ON_EVENT    0x4

//.. Something triggers the following statement to execute
gOS_EventSet(task1_id, 0x6);
...
void Task1_EventProcess(uint8 task_id, uint32 events)
{
  if (events & LED_OFF_EVENT)
  {
    HAL_TurnOffLed1();
  }

  if (events & LED_ON_EVENT)
  {
    HAL_TurnOnLed1();
  }
}
```

The following code is from the gOS demo project '*app_02_event_related*'. See Section 4.2 for its performing effect.

```
void LED_EventProcessor(byte task_id, uint32 events)
{
  /**
   * LED_ON_EVENT event has happened
   */
  if (events & LED_ON_EVENT)
  {
    HAL_TurnOnLed1();
  }

  /**
```

```
    * LED_OFF_EVENT event has happened
    */
  if (events & LED_OFF_EVENT)
  {
    HAL_TurnOffLed1();
  }
}

/**
 * When the key is pressed,the LED_OFF_EVENT event will trigger
 * wgen the key is released ,the LED_ON_EVENT event will trigger
 */
void LEDTask_OnKeyClick(uint8 key)
{
  switch (HAL_GetKeyState(key))
  {
  case HAL_KEY_PRESSED:
  {
    gOS_EventSet(led_task_id, LED_OFF_EVENT);
    break;
  }
  case HAL_KEY_RELEASED:
  {
    gOS_EventSet(led_task_id, LED_ON_EVENT);
    break;
  }
  default:
    break;
  }
}
```

3.4 Timer

Here the timer is not a hardware resource, and it is simulated by software and provides timer function to the target application. In order to use gOS timer, a hardware tick counter source shall be provided to gOS. See Section 2.2 and Section 3.2 in the document '*BDSE Driver Hardware Abstract Layer*'. To start and stop a timer, the following API shall be used:

```
BDE_STATUS gOS_StartTimer(byte task_id, uint32 event_id, uint16 timeout_value);
```

```
BDE_STATUS gOS_StopTimer(byte task_id, uint32 event_id);
```

To start a timer, the task which wants to use the timer shall be specified. And event_id parameter is used to specify what events are triggered when specified timeout (in millisecond) expires. Before the specified timeout events are triggered, target application can invocate **gOS_StopTimer()** to stop an unexpired timer.

NOTE: gOS takes the pair of task_id and event_id parameters as an identifier of timer. Each timer has its unique task_id and event_id parameters pair value. To stop a timer, the pair value must be same as the pair value of **gOS_StartTimer()**. For example, the following two examples are problematic. The **gOS_StopTimer()** invocating is no effect.

Example 1:

```
#define EVENT1      0x1
#define EVENT2      0x2

gOS_StartTimer(task_id, EVENT1 | EVENT2, 500);

//... Before 500ms expired
gOS_StopTimer(task_id, EVENT1); // This invocating has no effect, the return
                                // value is stateInvalidTimer
```

Example 2:

```
#define EVENT1      0x1
#define EVENT2      0x2
gOS_StartTimer(task_id, EVENT1, 500);

//... Before 500ms expired
gOS_StopTimer(task_id, EVENT1 | EVENT2); // This invocating has no effect,
                                // the return value is stateInvalidTimer
```

We strongly recommend that DO NOT start a timer with multiple events at a time.

NOTE: Before a timer expires, re-starting the same gOS timer updates the timeout value to the new invocating of **gOS_StartTimer()**. Figure 3.1 illustrates this case.

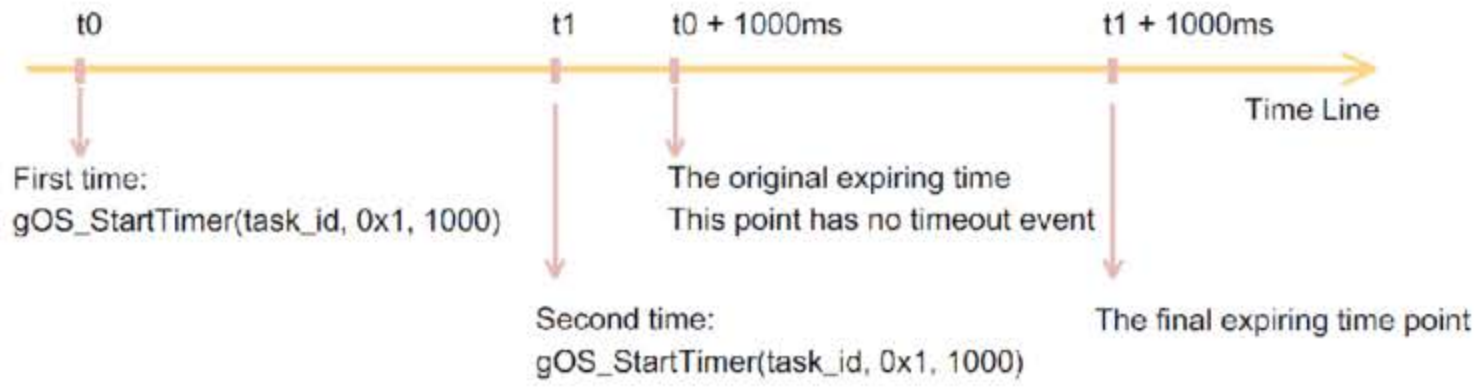

Figure 3.1 Updating the expiring time of gOS timer

NOTE: **gOS_StartTimer()** allocates a temporary memory block from gOS memory pool. The temporary memory pool is released when the specified timeout event happens or when **gOS_StopTimer()** is invocated to stop the specified timer earlier.

NOTE: **gOS_StartTimer()** and **gOS_EventSet()** APIs have the similar function. Both the 2 APIs are designed to trigger some events. The difference between them is: **gOS_EventSet**() triggers the event immediately, but **gOS_StartTimer**() triggers the event only when the specified time expires.

```
void LED_EventProcessor(byte task_id, uint32 events)
{
  if (events & LED_FLASH_EVENT)
  {
    HAL_ToggleLed1();
    gOS_StartTimer(led_task_id, LED_FLASH_EVENT, LED_FLASH_INTERVAL);
  }
}

// This function invocating is triggered when KEY is pressed
void LED_StopFlashing(void)
{
  gOS_StopTimer(led_task_id, LED_FLASH_EVENT);
}
```

The code above is extracted from the gOS demo project '*app_04_timer_related*'. See Section 4.4 for its performing effect.

3.5 Message

Messages help to exchange multiple bytes contents between tasks. To use messages, the following 4 APIs shall be used:

```
BDE_STATUS gOS_MsgAlloc(uint8 **ptr_msg, uint8 size);
BDE_STATUS gOS_MsgSend(uint8 dest_task_id, uint8 *ptr_msg, uint8 size);
gOS_RcvdMsgSavedStruc *gOS_MsgReceive(uint8 task_id);
```

```
BDE_STATUS gOS_MsgFree(uint8 *ptr_msg);
```

gOS_MsgAlloc() is used to allocate message managing buffer and contents buffer from gOS memory pool. It returns the pointer to the message contents buffer. If the allocating is successful, a valid pointer will be returned through the output function parameter ptr_msg. After the message buffer allocating, valid contents shall be filled into the message content buffer before the message sending.

NOTE: 0 size message is not supported by gOS. Users shall determine whether the allocating operation is successful before using the message.

gOS_MsgSend() is used to send the message to the specified task. Invocating **gOS_MsgSend()** pushes the sent message to a message managing queue. The queue is based on FIFO (first in and first out) method to maintain all tasks messages scheduling. As mentioned in Section 3.3, the **gOS_MsgSend()** API takes **SYS_EVENT_MSG** (0x80000000) as its scheduling event.

gOS_MsgReceive() is used to receive messages for a specified task. Its return value indicates the message contents length, message contents pointer, message sending task, and some other information.

gOS_MsgFree() is used to release the message managing buffer and its contents buffer. If the receiving messages are of no use, release the message buffer by invocating this API.

The following code demonstrates how to allocate a message and how to send it.

```
uint8* ptr_msg = 0;
if (gOS_MsgAlloc(&ptr_msg, 8) == stateSuccess)
{
   // If allocating is successful, the ptr_msg will pointer to the message content buffer
   ptr_msg[0] = (uint8)'h';
   ptr_msg[1] = (uint8)'e';
   ptr_msg[2] = (uint8)'l';
   ptr_msg[3] = (uint8)'l';
   ptr_msg[4] = (uint8)'o;
   ptr_msg[5] = (uint8)'m';
   ptr_msg[6] = (uint8)'s';
   ptr_msg[7] = (uint8)'g';

   gOS_MsgSend(task_id, ptr_msg, 8);
}
```

The following code shows how to receive a gOS message and how to release its memory.

```
void TEST_TaskInit(uint8 task_id)
{
  test_task_id = task_id;
}

void TEST_TaskEventProcess(uint8 task_id, uint32 events)
{
  gOS_RcvdMsgSavedStruc* msg = 0;
  uint8* ptr_msg = 0;
  uint8 msg_size = 0;
  uint8 sending_task_id;

  if (events & SYS_EVENT_MSG)
  {
    /**
     * If SYS_EVENT_MSG event is triggered, receiving message just
     * like the following code
     */
    while (1)
    {
      msg = gOS_MsgReceive(test_task_id);l
      if (!msg)
      {
        break;
      }
      ptr_msg = msg->ptr_msg; // Pointer to message contents buffer
      msg_size = msg->hdr.send_len;
      sending_task_id = msg->hdr.task_id;
      //.. Do something according to the message contents
      ...
      /**
       * Release the message buffer if the message is no use
       */
      gOS_MsgFree(ptr_msg);
    }
  }
}
```

From the source code above, the following notes must be paid attention to:

1) If **SYS_EVENT_MSG** is triggered, the target task shall pop all the messages from the message queue at a time in the invocating of event processing function.

2) All gOS message managing buffers are dynamically allocated from gOS memory pool. So, if the message is of no use, release it by invocating **gOS_MsgFree()**. And when a message memory buffer is released, the parameter passed to **gOS_MsgFree()** is the message contents pointer instead of the pointer returned by **gOS_MsgReceive()**.
3) To allocate a message, **gOS_MsgAlloc()** must be used. And to release a message memory, **gOS_MsgFree()** must be used. It is worth reminding that DO NOT use **gOS_MemoryAlloc()** and **gOS_MemoryFree()** instead of **gOS_MsgAlloc()** and **gOS_MsgFree()**.

In addition, it is possible to forward a gOS message to another task which needs to process the message. See the following example code.

```
void TEST1_TaskEventProcess(uint8 task_id, uint32 events)
{
  if (events & SYS_EVENT_MSG)
  {
    while (1)
    {
      msg = gOS_MsgReceive(test_task_id);l
      if (!msg)
      {
        break;
      }
      //.. Do something according to the message contents
      ...
      /**
       * Forward the message to another task.
       * If the message is determined to be no use in next task, its shall be released
       * in next task by invocating gOS_MsgFree().
       */
      gOS_MsgSend(the_next_task_id, msg->ptr_msg, msg->hdr.msg_size);

      /**
       * DO NOT invocating gOS_MsgFree() to release the message memory in current
       * forwarding task.
       */
    }
  }

}
```

The following code comes from the gOS demo project '*app_03_msg_related*'. See Section 4.3 for its performing effect.

```
//.. Message Sending
```

```
void MSGTEST_EventProcessor(byte task_id, uint32 events)
{
  byte* ptr_msg = NULL;

  if (events & KEY_PRESSED_EVENT)
  {
    static uint8 flash_cnt = 1;
    static uint16 flash_interval = 500; /*500 ms*/

    if (gOS_MsgAlloc(&ptr_msg, 3) == stateSuccess)
    {
      if (flash_cnt == 5)
      {
        flash_cnt = 1;
      }

      ptr_msg[0] = flash_cnt;
      memcpy(ptr_msg + 1, &flash_interval, sizeof(uint16));
      gOS_MsgSend(led_task_id, ptr_msg, 3);

      flash_cnt++;
    }
  }
}

//.. Message receiving and processing
void LED_EventProcessor(byte task_id, uint32 events)
{
  gOS_RcvdMsgSavedStruc *ptr_msg;
  byte* ptr_msg_content;

  /**
   * LED_FLASH_EVENT event has happened
   */
  if (events & LED_FLASH_EVENT)
  {
    HAL_ToggleLed1();

    if (flash_cnt)
    {
      flash_cnt--;
      gOS_StartTimer(led_task_id, LED_FLASH_EVENT, flash_interval);
    }
```

```
    else
    {
      HAL_TurnOffLed1();
    }
}

/**
 * System event message has happened
 */
if (events & SYS_EVENT_MSG)
{
  do
  {
    /**
     * Get the msg
     */
    ptr_msg = gOS_MsgReceive(led_task_id);

    if (ptr_msg)
    {
      ptr_msg_content = ptr_msg->ptr_msg;

      flash_cnt = ptr_msg_content[0];
      flash_cnt *= 2;
      flash_interval = *(uint16*) &ptr_msg_content[1];

      /**
       * Make the LED_FLASH_EVENT event happens immediately
       */
      gOS_EventSet(led_task_id, LED_FLASH_EVENT);

      /**
       *   It must to free the msg buffer
       */
      gOS_MsgFree(ptr_msg_content);
    }
  } while (ptr_msg);
 }
}
```

3.6 Memory

gOS has its internal memory management method .But it also allows users to specify a third party of memory management method. The gOS internal memory management method is based on First Fit algorithm.

There are three APIs related to gOS memory management:

```
void gOS_SetMemMgr(void     (*mem_mgr_init)          (void),
                   void*    (*mem_alloc)             (uint16),
                   void     (*mem_free)              (void*),
                   uint8*   default_mem_mgr_pool,
                   uint32   defualt_mem_mgr_pool_size)
void *gOS_MemoryAlloc(uint16 size)
byte gOS_MemoryFree(void *ptr);
```

gOS_SetMemMgr() specifies which memory managing method will be used, the internal First Fit method or a third party method. No matter what method is specified, the related APIs to allocate and release memory are **gOS_MemoryAlloc()** and **gOS_MemoryFree()** respectively.

The following code is 3 examples for setting memory managing method:

Example 1 uses the internal memory management:

```
static uint8 mem_pool[1024];

int main(void)
{
  ...
  /**
   * Set the internal method as the default memory managing method.
   */
  gOS_SetMemMgr(0, 0, 0, mem_pool, sizeof(mem_pool));
  ...
  return 0;
}
```

Example 2 uses TLSF (Two-Level Separated Fit) allocator:

```
static uint8 mem_pool[8*1024];
void TLSF_Init(void)
{
  init_memory_pool(sizeof(mem_pool), mem_pool);
}

void* TLSF_Malloc(uint16 size)
{
  return tlsf_malloc(size);
}
```

```
void TLSF_Free(void* pointer)
{
   tlsf_free(pointer);
}

int main(void)
{
   ...
   /**
    * Set TLSF as the default memory managing method.
    */
   gOS_SetMemMgr(TLSF_Init, TLSF_Malloc, TLSF_Free, 0, 0);
   ...
   return 0;
}
```

Example 3 uses the complier defined memory management.

```
#include <stdlib.h>

void ThirdParty_Init(void)
{

}

void* ThirdParty_Malloc(uint16 size)
{
   return malloc(size);
}

void ThirdParty_Free(void* pointer)
{
   free(pointer);
}

int main(void)
{
   ...
   /**
    * Set Complier defined memory management as the default memory managing method.
    */
   gOS_SetMemMgr(ThirdParty_Init, ThirdParty_Malloc, ThirdParty_Free, 0, 0);
   ...
```

```
    return 0;
}
```

NOTE: If complier memory management is selected, the heap (memory pool) size may need to be specified in IDE project setting.

NOTE: All the gOS kernel features, task registering, timer scheduling and message scheduling, are designed based on gOS memory management. If permitted, we strongly recommend the applications DO NOT share the memory pool if the pool size is set as 1Kbytes or smaller.

3.7 Power management

Most embedded applications care their power consumption. gOS has an internal power managing mechanism that can be enabled or disabled dynamically. There is only 1 API about the power management:

```
void gOS_SetSleepParam(uint16 min_dev_slp_time, uint16 max_dev_slp_time);
```

In order to use the gOS power management, an MCU sleep function needs to be registered to gOS. See Section 2.2 and Section 3.3 in the document *'BDSE Driver Hardware Abstract Layer'*.

If **gOS_SystemStart()** is invocated with its parameter equal to 0 and **gOS_SetSleepParam()** is invocated with its first parameter equal to 1, the power managing will be enabled. Otherwise, the power managing feature is disabled.

The flowchart in Figure 3.2 shows the power management method.

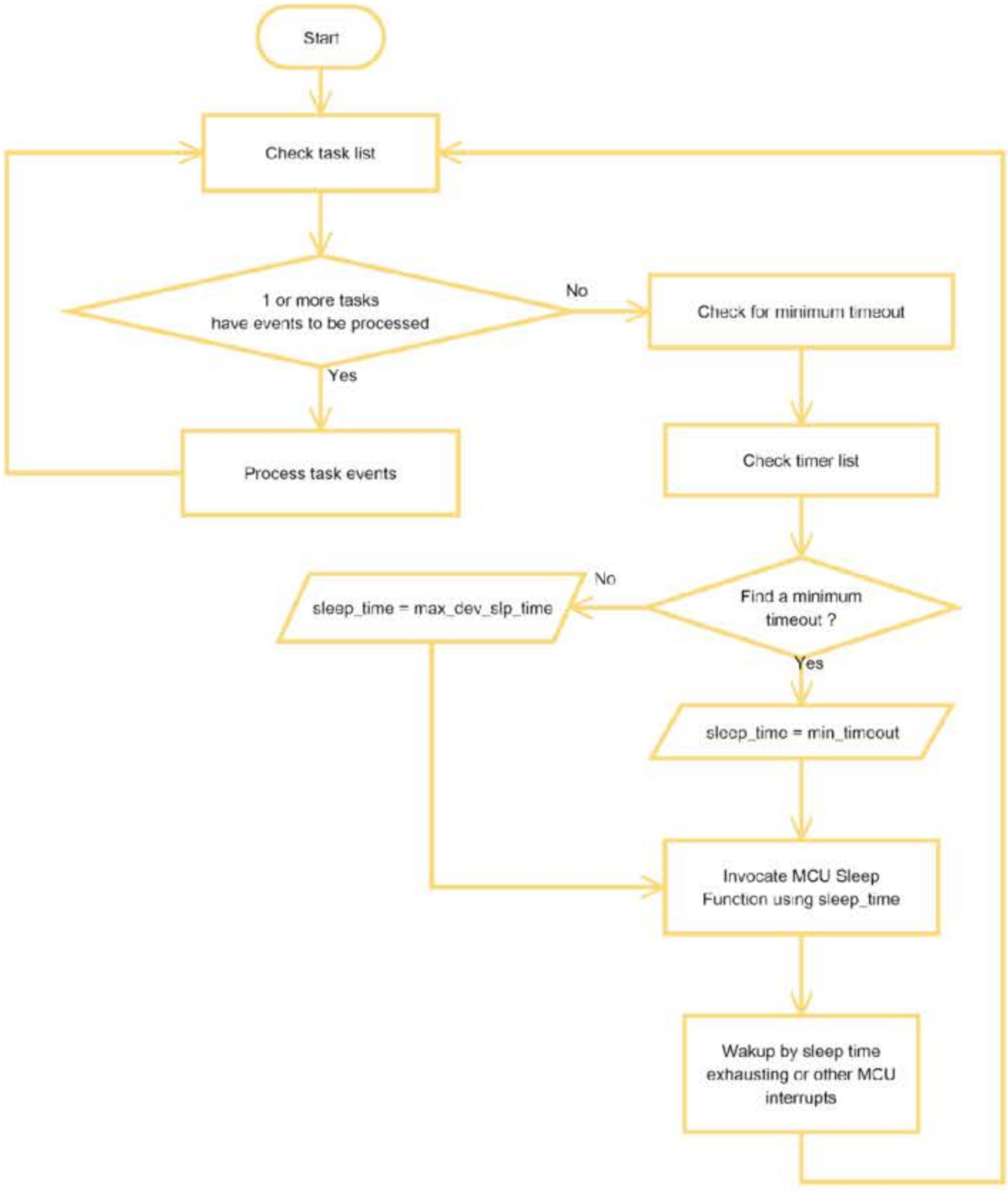

Figure 3.2 gOS power managing method

NOTE: The variable **max_dev_slp_time** is set by the invocating of **gOS_SetSleepParam()**. This variable is taken as the default sleep expired time when gOS timer list is empty.

In addition, to verify the power managing effect, see the gOS demo project '*App_05_lower_power_related*'. And see Section 4.5 for the project performing effect.

4. gOS demo project performing effect

To verify the gOS demo project performing effect, HAL must be implemented at the specified hardware target. For each gOS demo project, there are some relative hal implementation files located in each demo project directory.

4.1 App_01_task_add_related

LED1 and LED2 blink alternately at 1Hz frequency.

4.2 App_02_event_related

LED1 is turned on by default. When KEY1 is pressed, LED1 will be turned off. When KEY1 is released, LED1 will be turned on.

4.3 App_03_msg_related

When any KEY is pressed, the application sets an event (0x00000001) to the msgtest_task, and then the task sends a message to led_flash_task which contents are 3 bytes in length.

There are two fields in the simple gOS message. The first 1 byte is an integer which increases by one when any KEY is pressed and rolls over when it counts up to 4. The second part is 2 bytes in length. It indicates the LED toggle interval.

When led_flash_task receives the message, it toggles the LED for specified times and at specified interval included in the message.

4.4 App_04_timer_related

The LED toggles at 500 milliseconds interval. And if any KEY is pressed, the LED stops toggling and re-toggles when the pressed key is released.

When stop toggling, the LED's status may be OFF or ON.

4.5 App_05_lower_power_related

The performing procedure of this project is as following:

Turn Off LED -> Sleep for 5000 milliseconds -> Wake up to turn on LED -> Sleep 100 milliseconds -> Wake up to turn off LED -> ...

Pressing KEY will wake up MCU if it is in sleep state. And MCU will check whether there are some events waiting for processing. And if no event needs to be processed, it will re-enter sleep mode.