专题一、事件驱动调度机制参考源代码

1）文件 evt_driven_sched.c

```
/**
 * @file evt_driven_sched.c
 *
 * @brief  Event driven scheduler.
 * @author    Computer Science and Technology
 * @author    Wuhan University of Technology
 */

/* === INCLUDES ============================================================ */
#include "evt_driven_sched.h"
#include "timer.h"
#include "kernel.h"
#include "kdebug.h"
#include "board.h"

/* === TYPES =============================================================== */


/* === MACROS ============================================================== */


/* === GLOBALS ============================================================= */
/* Each bit in the task_flags corresponds to a task. */
uint16_t task_flags = 0;
/* task list, defined in file "demoTasks.c". */
extern tsk_handler_t tsk_hd_table[];

#if KDEBUG_DEMO
uint8_t nonRT_tskID = 0xFF;
#endif

/* === PROTOTYPES ========================================================== */


/* === IMPLEMENTATION ====================================================== */
/**
 * Once an event is generated, the related flag will be set.
 * For the event scheduler, it will poll the flags in loop.
 * In case that a flag is observed to be set, the related task will be executed.
 *
 * Task flags are polled byte by byte firstly, if some bits within a byte have
been set, the bits will be polled secondly.
```

```
 * Task have priorities, and the priorities are statically assigned offline.
 *
 * If no tasks are active, the sleeping directive will be executed to make the
sensor node fall asleep.
 *
 * After an event handler runs to completion, further action (memory resource
releasing, etc.) will be taken in terms of the execution results.
 *
 */
void
event_driven_scheduling(void)
{
    uint8_t i;
    uint16_t fb;
    tsk_handler_t tsk_exec = NULL;

    /* In case that no tasks are active, enter the idle status. */
    if(task_flags == 0)
    {
        //hardware_sleep();
        return;
    }

    /* Poll the task flags and execute the tasks if the task flag is set.
       Clear the task flag after a task runs to completion. */
    for(i = 0, fb = 0x01; i < 16; i++, fb = fb << 1)
    {
        if((task_flags&fb) != 0)
        {
            /* get the task handler address from the task table. */
            tsk_exec = tsk_hd_table[i];
            /* record the task ID for demo debug */
            #if KDEBUG_DEMO
            nonRT_tskID = i;
            #endif
            /* execute the task handler. */
            tsk_exec();
            /* clear the task flag. */
            task_flags &= ~(0x01 << i);
            /* break here, thus the high-priority task will always be executed in
advance. */
            break;
        }
    }
}
```

```
/**
 * @brief Post an event to a task to active it (by setting the task flag, 16bits).
 *
 * Once a task becomes active, it will be scheduled by the event-driven scheduler.
 */
INLINE void
eventPost(uint8_t task_ID)
{
    task_flags ^= (0x01 << task_ID);
}
```

2）文件 evt_driven_sched.h

```
/**
 * @file evt_driven_sched.h
 *
 * @brief  header for evt_driven_sched.c
 *
 * @author    Computer Science and Technology
 * @author    Wuhan University of Technology
 */

/* Prevent double inclusion */
#ifndef _EVENT_DRIVEN_SCHED_H_
#define _EVENT_DRIVEN_SCHED_H_

/* === Includes ============================================================ */
#include "board.h"
#include "typedef.h"
#include "os_start.h"

/* === Macros ============================================================== */

/* === Types =============================================================== */
/* task handler */
typedef uint8_t (*tsk_handler_t)(void);
extern uint8_t nonRT_tskID;

/* === Prototypes ========================================================== */
extern void event_driven_scheduling(void);
extern void taskPost(uint8_t task_ID);

#endif
```

专题二、多线程调度机制参考源代码

1）文件 multithreading_sched.c

```
/**
 * @file multithreading_sched.c

 * @brief  multi-threading scheduler to schedule the real-time tasks.
 *
 *  All the RT tasks in will be scheduled by the multi-threading scheduler.
 *  If common_thread is executing and any RT thread becomes active during this executing process,
 *  common_thread will be preempted and then the OS will switch
 *  from the event-driven scheduling model to the multithreaded scheduling model.
 *
 * @author    Computer Science and Technology
 * @author    Wuhan University of Technology
 */

/* === INCLUDES ============================================================ */
#include "evt_driven_sched.h"
#include "multithreading_sched.h"
#include "kernel.h"
#include "kdebug.h"
#include "usart.h"

#if RT_SUPPORT
/* === TYPES =============================================================== */


/* === MACROS ============================================================== */


/* === GLOBALS ============================================================= */
/* thread control tables (TCB) to store the thread specific information.
   Since the thread number is small (only for RT tasks),
   the TCB is pre-reserved other than dynamically allocated. */
thrd_tcb_t thrd_TCB[MAX_THREAD_NUM], *thrd_lstQ= NULL;

/* === PROTOTYPES ========================================================== */



/* === IMPLEMENTATION ====================================================== */
/**
 * @brief created a thread to execute the RT task.
 *
 * This function will create a new thread, establish the thread run-time context,
 * and then force the thread switch.
 *
 * \param thrd_tsk     The RT task which will be executed by the thread.
 *
 */
thrd_tcb_t*
thread_create(tsk_handler_t thrd_tsk, uint16_t tsk_period)
{
    thrd_tcb_t *thrd = NULL;

    /* thread context creation. */
    thrd = thrd_contextPrep(thrd_tsk, tsk_period);
    /* force the thread switch. */
    thread_dispatcher();

    return thrd;
}

/**
 * @brief Create the thread context.
 *
 * Firstly, get a thread control table (TCB).
 * Then, allocate a thread stack.
 * Finally, initialize the thread stack.
 *
 * \param thrd_tsk     The RT task which will be executed by the thread.
 */
thrd_tcb_t*
thrd_contextPrep(tsk_handler_t thrd_tsk, uint16_t tsk_period)
{
    HAS_CRITICAL_SECTION;
    uint8_t i, id;
    thrd_tcb_t *thrd, *thrd_prev;

    /* get a thread control table (TCB) for this thread. */
    for(id = 0; id < MAX_THREAD_NUM; id++)
        if(thrd_TCB[id].status == THRD_UNUSED) break;
    if(id == MAX_THREAD_NUM)
    {
        /* maximum threads have been created. */
        sendUsartByte(USART_CHANNEL_1, 'M');
        return NULL;
    }
    /* allocate a thread run-time stack.
       Note that the stack is used from high address to low
       thus, should move the stack pointer "thrd_sp" to the stack bottom. */
    heapSaddr += THREAD_CONTEXT_SIZE;
    thrd_TCB[id].thrd_sp = (uint8_t *)(heapSaddr - 1);
    /* init the thread TCB. */
    thrd_TCB[id].next = NULL;
    thrd_TCB[id].thrd_tsk = thrd_tsk;
    thrd_TCB[id].thrd_period = tsk_period;
    thrd_TCB[id].status = THRD_ACTIVE;

    /* add this thread into the thread queue in the order of the thread priority.
       thread with the smallest "thrd_period" (highest priority) will be put at the
queue header. */
    if(thrd_lstQ == NULL)
        thrd_lstQ = &thrd_TCB[id];
    else
    {
        for(thrd_prev = thrd = thrd_lstQ; thrd != NULL; thrd = thrd->next)
        {
            /* insert new thread before "thrd". */
            if(thrd_TCB[id].thrd_period < thrd->thrd_period)
            {
                ENTER_CRITICAL_SECTION;
                /* insert to the head. */
                if(thrd == thrd_lstQ)
                {
                    thrd_TCB[id].next = thrd_lstQ;
                    thrd_lstQ = &thrd_TCB[id];
                }
                else
                {
                    thrd_prev->next = &thrd_TCB[id];
                    thrd_TCB[id].next = thrd;
                }
                LEAVE_CRITICAL_SECTION;
                break;
            }
            /* update "thrd_prev", the previous thread after which the new thread
will be inserted. */
            thrd_prev = thrd;
        }

        /* insert to the tail. */
        if(thrd == NULL)
            thrd_prev->next = &thrd_TCB[id];
    }

    /* curThrd should not be NULL. */
    if(curThrd == NULL)
    {
        /* ERROR here. */
        sendUsartByte(USART_CHANNEL_1, 'E');
        return NULL;
    }

    /*
    * The sequence of the following part should
    * correspond to that in "thrdContextRestore".
    */
    asm volatile(               \
    "in %A0, __SP_L__\n\t"              \
    "in %B0, __SP_H__\n\t"              \
    : "=r" (curThrd->thrd_sp) : );      \
        /* save current thread's stack */

    asm volatile(               \
    "out __SP_H__, %B0\n\t"         \
    "out __SP_L__, %A0\n\t"         \
    :: "r" (thrd_TCB[id].thrd_sp) );    \
        /* switch to the new created thread's stack */

    asm volatile(               \
    "push %A0\n\t"                  \
    "push %B0\n\t"                  \
    :: "r" (thrd_start_wrapper) );         \
        /* Push address of "thrd_start_wrapper" onto the stack.
           "thrd_start_wrapper" is the thread's entry function.
           After this operation, when the "RET" instruction is executed in
"thread_dispatcher",
           this function "thrd_start_wrapper" will be executed.
           */

    for(i = 0; i < 33; i++)             \
        asm volatile("push __zero_reg__\n\t" ::); \
        /* Reserve and init the registers.
           33 bytes: 32 for the registers and 1 for the sreg.
           use "__zero_reg__" as the operation of initialization. */

    asm volatile(               \
    "in %A0, __SP_L__\n\t"              \
    "in %B0, __SP_H__\n\t"              \
    : "=r" (thrd_TCB[id].thrd_sp) : );          \
        /* save the new created thread's sp pointer into "thrd_sp",
           it will be used in "thread_dispatcher". */

    asm volatile(                   \
    "out __SP_H__, %B0\n\t"         \
    "out __SP_L__, %A0\n\t"         \
    :: "r" (curThrd->thrd_sp) );        \
        /* recover the current thread's stack. */

    return &thrd_TCB[id];
}


/**
 * @brief Start executing a thread.
 *
 * Call the related system handlers for this HRT event.
 * Once a thread runs to completion, its run-time context can be released, and
 * this thread will also be deleted.
 */
void
thrd_start_wrapper(void)
{
    /* execute the current thread's handler. */
    curThrd->thrd_tsk();

    /* thread dispatcher, switch to execute the other threads. */
    thread_dispatcher();
}


/**
 * @brief  get the next thread to be scheduled, according to RMS algorithm.
 *
 * Since event-driven scheduler is implemented as a thread "common_thread",
 * if the "common_thread" is scheduled, the OS will switch back to event-driven
scheduling model.
 *
 * \return    Return the next thread to be executed.
 */
thrd_tcb_t*
getNextThread(void)
{
    thrd_tcb_t *thr;
    /* Implementation of RMS scheduling algorithm here.
       Since threads are ordered in the thread queue in terms of the priorities
(from highest priority to lowest priority).
       Thus, scan the thread queue from the header, the first active thread will
be the next one to be scheduled. */
    for(thr = thrd_lstQ; thr != NULL; thr = thr->next)
        if(thr->status == THRD_ACTIVE)  return thr;

    /* Scheduler switch if here.
       If all the threads are inactive, return the common_thread.
       After this, the context of the common_thread (that is, the event-driven
scheduling) will be restored.
       And then, the scheduler will switch to the event-driven model. */
    return &common_thread;
}



/**
 * @brief Force the thread switch.
 *
 * Save the current thread's run-time context,
 * and then recover the next thread's context.
 *
 * At the end of this function, the "RET" instruction
 * will be called in default, this instruction will pop the next thread's
 * execution address to the "PC (program counter)", and then
 * the next thread will start execution.
 */
void
thread_dispatcher(void)
{
    /* save current thread's context. */
    {
        thrdContextSave();
        asm volatile(           \
        "in %A0, __SP_L__\n\t"          \
        "in %B0, __SP_H__\n\t"          \
        : "=r" (curThrd->thrd_sp) : );  \
    }

    /* get the next thread to be scheduled in terms of the RMS scheduling algorithm,
       and assign this thread to "curThrd". */
    curThrd = getNextThread();

    /* recover the context of the next thread. */
    {
       asm volatile(            \
           "out __SP_H__, %B0\n\t"     \
           "out __SP_L__, %A0\n\t"     \
           :: "r" (curThrd->thrd_sp));
       thrdContextRestore();
   }

   /* "RET" directive is executed here in default:
    * This first time a thread is created, this "RET" will pop out
    * the address of "thrd_start_wrapper" to Program Counter, and
    * then the execution of a thread can be processed.
    */
}


/**
 * @brief Save the task context
 *
 * Save the task context data over the run-time stack
 * Save SREG
 * Save all the registers
 */
void
thrdContextSave(void)
{
    asm volatile(           \
    "push r24\n\t"              \
    "in r24, __SREG__\n\t"          \
    "cli\n\t"           \
    "push r24\n\t"              \
    );    /* save R24 and sreg */

    asm volatile(           \
    "push r31\n\t"              \
    "push r30\n\t"              \
    "push r29\n\t"              \
    "push r28\n\t"              \
    "push r27\n\t"              \
    "push r26\n\t"              \
    "push r25\n\t"              \
    "push r23\n\t"              \
    "push r22\n\t"              \
    "push r21\n\t"              \
    "push r20\n\t"              \
    "push r19\n\t"              \
    "push r18\n\t"              \
    "push r17\n\t"              \
    "push r16\n\t"              \
    "push r15\n\t"              \
    "push r14\n\t"              \
    "push r13\n\t"              \
    "push r12\n\t"              \
    "push r11\n\t"              \
    "push r10\n\t"              \
    "push r9\n\t"            \
    "push r8\n\t"            \
    "push r7\n\t"            \
    "push r6\n\t"            \
    "push r5\n\t"            \
    "push r4\n\t"            \
    "push r3\n\t"            \
    "push r2\n\t"            \
    "push r1\n\t"            \
    "push r0\n\t"            \
    );     /* save all registers */
}


/**
 * @brief Restore the task context
 *
 * the first time this function is called,
 * it will pop out the initialized values of
 * the registers that are assigned in "thrd_contextPrep".
 */
void
thrdContextRestore(void)
{
    /* restore the context data */
    asm volatile(            \
    "pop r0\n\t"             \
    "pop r1\n\t"             \
    "pop r2\n\t"             \
    "pop r3\n\t"             \
    "pop r4\n\t"             \
    "pop r5\n\t"             \
    "pop r6\n\t"             \
    "pop r7\n\t"             \
    "pop r8\n\t"             \
    "pop r9\n\t"             \
    "pop r10\n\t"            \
    "pop r11\n\t"            \
    "pop r12\n\t"           \
    "pop r13\n\t"           \
    "pop r14\n\t"           \
    "pop r15\n\t"           \
    "pop r16\n\t"           \
    "pop r17\n\t"           \
    "pop r18\n\t"           \
    "pop r19\n\t"           \
    "pop r20\n\t"           \
    "pop r21\n\t"           \
    "pop r22\n\t"           \
    "pop r23\n\t"           \
    "pop r25\n\t"           \
    "pop r26\n\t"           \
    "pop r27\n\t"           \
    "pop r28\n\t"           \
    "pop r29\n\t"           \
    "pop r30\n\t"           \
    "pop r31\n\t"           \
    "pop r24\n\t"           \
    "out __SREG__, r24\n\t"    \
    "pop r24\n\t"           \
    "sei\n\t"           \
    );
}

/**
 * @brief Set the status of this thread to ACTIVE.
 * \param thrd   The thread TCB to be operated.
 */
INLINE void
active_Thread(thrd_tcb_t *thrd)
{
    if(thrd != NULL)
        thrd->status = THRD_ACTIVE;
    /* yield the control to the others. */
    thread_dispatcher();
}

/**
 * @brief Set the status of this thread to SUSPENDED.
 * \param thrd   The thread TCB to be operated.
 */
INLINE void
yield_Thread(thrd_tcb_t *thrd)
{
    if(thrd != NULL)
        thrd->status = THRD_SUSPENDED;
    /* yield the control to the others. */
    thread_dispatcher();
}
#endif // RT_SUPPORT
```

2）文件 multithreading_sched.h

```
/**
 * @file multithreading_sched.h
 *
 * @brief  header for multithreading_sched.c
 *
 * @author    Computer Science and Technology
 * @author    Wuhan University of Technology
 */

/* Prevent double inclusion */
#ifndef _MULTITHREADING_SCHED_H_
#define _MULTITHREADING_SCHED_H_

/* === Includes ============================================================ */
#include "board.h"
#include "evt_driven_sched.h"


/* === Types =============================================================== */
/* thread status */
typedef enum
{
    THRD_UNUSED,
    THRD_ACTIVE,
    THRD_SUSPENDED,
    THRD_SLEEPING
} thrd_status_t;

/* thread TCB structure */
__ALIGNED2 typedef struct thrd_tcb
{
    struct thrd_tcb *next;
    uint8_t *thrd_sp;        /* stack's run-time address. */
    tsk_handler_t thrd_tsk;  /* pointer to the task executed by this thread. */
    struct thrd_tcb *semQ_next; /* queue for the resource semaphore. */
    uint16_t thrd_period;       /* period of the task, will determine the priority
of this thread. */
```

```
    uint8_t status;             /* ″UNUSED, SUSPENDED, ACTIVE, etc.″ */
    #if KDEBUG_DEMO
    uint8_t thrd_id;        /* used for demo. */
    #endif
} thrd_tcb_t;


/* === Macros ============================================================== */
#define MAX_THREAD_NUM      8
#define   THREAD_CONTEXT_SIZE  128


/* === GLOBALS ============================================================= */
extern thrd_tcb_t *thrd_lstQ;


/* === Prototypes =========================================================== */
extern thrd_tcb_t* thread_create(tsk_handler_t thrd_tsk, uint16_t tsk_period);
extern thrd_tcb_t* thrd_contextPrep(tsk_handler_t thrd_tsk, uint16_t tsk_period);
extern void thrd_start_wrapper(void);
extern thrd_tcb_t* getNextThread(void);
extern void thread_dispatcher(void);
extern void thrdContextSave(void);
extern void thrdContextRestore(void);
extern void active_Thread(thrd_tcb_t *thrd);
extern void yield_Thread(thrd_tcb_t *thrd);

#endif
```

专题三、固定大小块动态内存管理

1）文件 mem_SFL.c

```
/**
 * @file mem_SFL.c
 *
 * @brief  segregated free list (SFL) allocator.
 *     Segregated free list (SFL) allocator divides the memory space into segregated partitions.
 *     Each partition holds a set of specified size blocks, and a free list is used for the allocation in each partition.
 *     Upon allocation, a block is deleted from the free list which matches the allocation size.
 *     Upon releasing, the released block will be added to the matching free list.
 *
 *     To avoid the heap insufficiency problem, the sizes of the partitions are not reserved to the largest sizes that they may be required.
 *     Instead, the moderate sizes by which the requirements of most WSN applications can be met are assigned for the partitions.
 *     And in case a partition is memory overflowed, the allocation will not be failed but continue to be allocated in the extended heap space.
 *
 * @author    Computer Science and Technology
 * @author    Wuhan University of Technology
 */

/* === INCLUDES ============================================================ */
#include "typedef.h"
#include "kernel.h"
#include "kdebug.h"
#include "multithreading_sched.h"
#include "timer_ACV.h"
#include "timer_RCV.h"
#include "mem_SFL.h"
#include "mem_SFL_extHeap.h"
#include "ipc.h"

#if MEM_SFL
/* === TYPES =============================================================== */


/* === MACROS ============================================================== */
/* create partitions here: MEM_PARTITION_CREATE(name, struct_name, blkNum) */
MEM_PARTITION_CREATE(timer, timer_t, 2);  /* create partition for software timer, with 10 blocks. */
MEM_PARTITION_CREATE(ipc, ipc_msgQ_t, 2); /* create partition for IPC
sending/recving, with 30 blocks. */

/* === GLOBALS ============================================================ */


/* === PROTOTYPES ========================================================= */


/* === IMPLEMENTATION ===================================================== */
/**
 * @brief Initialization of SFL partition.
 *
 *  This function is used to link all the free blocks to the free list
 *  after a partition is created.
 *
 * \param *pt Header of a partition.
 * \return NULL.
 */
void
memSFL_partition_init(partition_t *pt)
{
    uint8_t    i;
    memblk_t *blk;
    /* link all the free blocks at the init stage. */
    for(i = 0, blk = pt->ptFreeQ; i < pt->blk_num; i++, blk = blk->next)
    {
        blk->next = (void *)blk + pt->blk_size + sizeof(memblk_t);
        blk->pt = pt;
        /* the last one */
        if(i == (pt->blk_num -1))
            blk->next = NULL;
    }
}
/**
 * @brief SFL allocation.
 *
 *  If SFL allocation from a given partition is failed, this allocation will be
done continuously inside the extended heap.
 *  For the extended heap, the SF allocation mechanism will be used.
 *
 * \param pt  The header of a given partition.
 * \return    Return the address of the allocated object.
 */
void*
mem_alloc(partition_t *pt)
```

```
{
    memblk_t *mem_blk = NULL;

    /* If partition free list is not NULL, allocate from this partition. */
    if(pt->ptFreeQ != NULL)
    {
        /* delete from the header directly, allocation can be completed in constant time. */
        mem_blk = pt->ptFreeQ;
        pt->ptFreeQ = pt->ptFreeQ->next;
        pt->blk_num--;
        /* return. */
        return ((void *)mem_blk + sizeof(memblk_t));
    }

    /* If partition free list is NULL, allocate from the extended heap space. */
    else
        return memSFL_extHeap_alloc(pt->blk_size);
}

/**
 * @brief Free a chunk
 *
 * If object to be released is inside the extended heap space, add it into the free list of hpFreeQ. In this case,
 * if two freed objects are adjacent, coalesce them.
 * If object to be released is inside a partition, add it into the partition free list ptFreeQ. In this case,
 * no memory coalescence will be needed.
 *
 * \param chkMem  Address of the chunk to be released.
 *
 */
void
mem_free(void *chkMem)
{
    memblk_t *mem_blk = NULL;

    /* if object locates insides a partition, add this object to the header of partition free list. */
    if(chkMem < heapSaddr)
    {
        mem_blk = (memblk_t *)((void *)chkMem - sizeof(memblk_t));
        /* insert into the header of list ptFreeQ. */
        mem_blk->next = mem_blk->pt->ptFreeQ;
        mem_blk->pt->ptFreeQ = mem_blk;
        /* update the available number. */
        mem_blk->pt->blk_num++;
    }

    /* if object locates in the extended heap, add this object to the free list
hpFreeQ. */
    else
        memSFL_extHeap_free(chkMem);
}
#endif
```

2）文件 mem_SFL.h

```
/**
/* Prevent double inclusion */
#ifndef _mem_SFL_H_
#define _mem_SFL_H_

/* === Includes ============================================================ */
#include "board.h"
#include "evt_driven_sched.h"
#include "qlist_proc.h"
#include "mem_SFL_extHeap.h"
#include "timer_ACV.h"
#include "timer_RCV.h"

#if MEM_SFL
/* === Types =============================================================== */
typedef struct partition partition_t;
/* block structure in each partition.
   Maximum size of each block will be: 0xFF-sizeof(memblk_t) = 251. */
__ALIGNED2 typedef struct memblk {
    struct memblk *next; /* single link list to link all the free blocks. */
    partition_t *pt;     /* link to the partition header, for block collection when
this block is released. */
} memblk_t;

/* header for each partition. */
__ALIGNED2 struct partition {
    uint8_t blk_size;    /* blk_size is commonly the size of the struct. */
    uint8_t blk_num;     /* "blk_num/blk_size" will be used for memory reservation
at initialization stage. */
    memblk_t *ptFreeQ;      /* Qhead to link all the free blocks, init to the
partition's starting address at the system startup. */
};
```

```
/* === Macros ============================================================ */
/**
 * preprocessing macro for concatenating to strings.
 *
 * We need use two macros (CC_CONCAT and CC_CONCAT2) in order to allow
 * concatenation of two #defined macros.
 */
#define CC_CONCAT2(s1, s2) s1##s2
#define CC_CONCAT(s1, s2) CC_CONCAT2(s1, s2)

/*
 * This macro is used to create a new partition for SFL allocation.
 * pre-reserve the partition memory space firstly,
 * and then define and initialize the partition header.
 *
 * \param name
 *     The name of this memory partition (later used with memb_init(), memb_alloc()
and memb_free()).
 * \param struct_name
 *     The name of the struct that this memory partition will hold.
 * \param blkNum
 *     The total number of memory blocks in this partition.
 */
#define MEM_PARTITION_CREATE(name, struct_name, blkNum) \
   uint8_t  CC_CONCAT(name,_blks)[(sizeof(struct_name)+sizeof(memblk_t))*blkNum];
\
   partition_t CC_CONCAT(name,_pt) = {    \
      sizeof(struct_name), \
      blkNum, \
      (void *)CC_CONCAT(name,_blks)}

/* === GLOBALS =========================================================== */
extern uint8_t timer_blks[];
extern partition_t timer_pt;
extern uint8_t ipc_blks[];
extern partition_t ipc_pt;

/* === Prototypes ======================================================== */
/* memory management */
extern void memSFL_partition_init(partition_t *pt);
extern void* mem_alloc(partition_t *pt);
extern void mem_free(void *chkMem);

#endif
#endif
```

专题四、主动碎片回收动态内存管理

1）文件 mem_proactive_SF.c

```
/**
 * @file mem_proactive_SF.c
 *
 * @brief  sequential fit allocator.
 * The fragment assembling mechanism has been implemented for the sequential fit
allocator.
 *         And currently, two assembling approaches have been realized: the
proactive fragment assembling and
 *         the reactive fragment assembling (concepts motivated by the proactive
routing protocol, e.g. the DSDV,
 *         and the reactive routing protocol, e.g. the AODV).
 *
 *         For proactive fragment assembling, the memory fragments are assembled
once they are appeared.
 *         By this way, the fragment problem can be prevented to occur,
 *         and the new allocation can be done immediately with constant response
time.
 *
 * @author    Computer Science and Technology
 * @author    Wuhan University of Technology
 */

/* === INCLUDES ============================================================ */
#include "board.h"
#include "kernel.h"
#include "sys_config.h"
#include "kdebug.h"

#if MEM_PROACTIVE_SF
#include "mem_proactive_SF.h"
/* === TYPES =============================================================== */


/* === MACROS ============================================================== */


/* === GLOBALS ============================================================= */
/* references for the allocated chunks. */
uint16_t* proSF_Ref[REF_NUM];

/* starting address of left heap. */
void* leftHpSaddr;
```

```
/* === PROTOTYPES ========================================================== */


/* === IMPLEMENTATION ====================================================== */
/**
 * @brief Memory allocated by sequential fit (proactive fragments assembled).
 *        If a new object needs to be allocated, it will be done from the starting address of the free memory space.
 *        And if an allocated object needs to be de-allocated, the coalescence will be done to the adjacent chunks of this object.
 *        As the addresses of some objects will change after the coalescing operation,
 *        all the allocated objects in this allocation method need to be accessed indirectly by the reference pointers.
 *
 * \param reqSize Required size to be allocated.
 * \return        Address of the reference header for the allocated chunk.
 */
uint16_t*
mem_alloc(uint8_t reqSize)
{
    proSF_chk_hdr_t* chk_alloc = NULL;
    uint8_t chkSize = reqSize + sizeof(proSF_chk_hdr_t);
    uint8_t id;

    /* Check if we have enough memory left for this allocation. */
    if((uint16_t)((uint16_t)leftHpSaddr+chkSize) > HEAP_EADDR)  return NULL;

    /* allocate a reference for this chunk. */
    for(id = 0; id < REF_NUM; id++)
        if(proSF_Ref[id] == NULL)   break;
    /* references are used up, maximum allocation. */
    if(id == REF_NUM) return NULL;

    /* memory space to be allocated. */
    chk_alloc = (proSF_chk_hdr_t *)leftHpSaddr;
    /* init this new chunk. */
    chk_alloc->chk_size = chkSize;
    proSF_Ref[id] = (void *)chk_alloc + sizeof(proSF_chk_hdr_t);
    chk_alloc->chk_ref = (uint16_t *)&proSF_Ref[id];

    /* update new heap starting address. */
    leftHpSaddr += chkSize;

    /* return chunk address. */
    return chk_alloc->chk_ref;
}

/**
 * @brief Free a chunk
 *      For proactive fragment assembling, the memory fragments are assembled once
they are appeared.
 *      By this way, the fragment problem can be prevented to occur,
 *      and the new allocation can be done immediately with constant response time.
 *
 * \param chkMem  Address of the chunk to be released.
 *
 */
void
mem_free(uint16_t *chkMem)
{
    proSF_chk_hdr_t *m;
    uint8_t *mvSaddr, *mvTo;
    uint16_t i, sft_size;

    /* "*chkMem" will point to the data payload,
       assign "mvTo" and "mvSaddr". */
    mvTo = (uint8_t *)(*chkMem - sizeof(proSF_chk_hdr_t));
    sft_size = ((proSF_chk_hdr_t *)mvTo)->chk_size;
    mvSaddr = (uint8_t *)(mvTo + sft_size) ;

    /* release the reference. */
    *(((proSF_chk_hdr_t *)mvTo)->chk_ref) = 0;

    /* update the references of the other chunks before memory coalescence. */
    for(m = (proSF_chk_hdr_t *)mvSaddr; m != leftHpSaddr; m = (proSF_chk_hdr_t
*)((uint16_t)m + m->chk_size))
        *(m->chk_ref) -= ((proSF_chk_hdr_t *)mvTo)->chk_size;

    /* memory coalescence. */
    for(i = 0; i < ((uint16_t)leftHpSaddr - (uint16_t)mvSaddr); i++)
        *(mvTo + i) = *(mvSaddr + i);

    /* update "leftHpSaddr". */
    leftHpSaddr -= sft_size;
}

#endif
```

```
/**
 * @file mem_proactive_SF.h
 *
 * @brief  header for mem_proactive_SF.c
 *
 * @author    Computer Science and Technology
 * @author    Wuhan University of Technology
 */

/* Prevent double inclusion */
#ifndef _MEM_PROACTIVE_SF_H_
#define _MEM_PROACTIVE_SF_H_

/* === Includes ============================================================= */
#include "board.h"
#include "kernel.h"
#include "sys_config.h"
#include "qlist_proc.h"
#include "typedef.h"

#if MEM_PROACTIVE_SF
/* === Macros =============================================================== */
/* reference number, maximum allocation. */
#define REF_NUM      20


/* === Types ================================================================ */
typedef __ALIGNED2 struct proSF_chk_hdr
{
   uint16_t *chk_ref;          /* pointer linked to reference. */
   uint16_t chk_size;          /* chunk size, including header "proSF_chk_hdr_t",
used when chunk removing, etc. */
} proSF_chk_hdr_t;

/* === GLOBALS ============================================================= */
extern void* leftHpSaddr;
extern uint16_t* proSF_Ref[];

/* === Prototypes =========================================================== */
extern uint16_t* mem_alloc(uint8_t reqSize);
extern void mem_free(uint16_t *chkMem);


#endif
#endif
```

专题五、被动式碎片回收动态内存管理

1）文件 mem_reactive_SF.c

```
/**
 * @file mem_reactive_SF.c
 *
 * @brief  sequential fit allocator.
 *  The fragment assembling mechanism has been implemented for the sequential fit allocator.
 *          Currently, two assembling approaches have been realized: the proactive fragment assembling and
 *          the reactive fragment assembling (concepts motivated by the proactive routing protocol, e.g. the DSDV,
 *          and the reactive routing protocol, e.g. the AODV).
 *
 *          For reactive fragment assembling mechanism, the fragments are not assembled every time an allocated object is released.
 *          Instead, they are assembled only when the 1st time SF allocation is failed, that is,
 *          when there is no enough continuous free memory left for the new allocation.
 *
 *          The same as proactive fragment assembling mechanism, reference pointers need to be used in
 *          reactive fragment assembling mechanism as well,
 *          and the update to the related reference pointers must be done after the fragments are assembled.
 *
 * @author    Computer Science and Technology
 * @author    Wuhan University of Technology
 */

/* === INCLUDES ============================================================ */
#include "board.h"
#include "kernel.h"
#include "sys_config.h"
#include "kdebug.h"

#if MEM_REACTIVE_SF
#include "mem_reactive_SF.h"
/* === TYPES =============================================================== */


/* === MACROS ============================================================== */
/* === GLOBALS ============================================================ */
/* references for the allocated chunks. Reference is 16-bit on AVR platform
(uint16_t).
   Reference will point to the data payload starting address (exclude the header)
*/
uint16_t* reSF_Ref[REF_NUM];

/* free memory list for reactive SF allocator. */
reSF_chk_hdr_t* reSF_freeQ = NULL;

#if KDEBUG_DEMO
reSF_chk_hdr_t* reSF_allocQ = NULL;
#endif

/* === PROTOTYPES ========================================================= */

/* === IMPLEMENTATION ===================================================== */
/**
 * @brief Memory allocated by sequential fit (reactive fragments assembled).
 *        If the 1st time SF allocation is failed, that is, when there is no enough
continuous free memory left for the new allocation,
 *        the assembling operation will be done to the different memory fragments.
 *        After this, the new allocation will be tried for another time.
 *        References need to be updated after the fragments are assembled.
 *
 * \param objSz      Required size to be allocated.
 * \return       Address of the reference header for the allocated chunk.
 */
uint16_t*
mem_alloc(uint8_t objSz)
{
    reSF_chk_hdr_t *alloc = NULL;
    uint8_t id;

    /* allocate a reference for this chunk firstly. */
    for(id = 0; id < REF_NUM; id++)
        if(reSF_Ref[id] == 0)    break;
    /* references are used up, maximum allocation. */
    if(id == REF_NUM) return NULL;

    /* Compute the required length. */
    objSz = ALIGN(objSz+sizeof(reSF_chk_hdr_t), ALIGN_SIZE);

    /* 1st time allocation from the free memory list.
       If failed, need to assemble all the memory fragments. */
    if((alloc = mem_alloc_proc(objSz)) == NULL)
    {
        /* assemble all the fragments. */
        fragment_assemble();

        /* 2nd time allocation, after fragments are assembled.
           From the header of free memory list "reSF_freeQ" directly. */
        if(reSF_freeQ->ckSize < objSz)  return NULL;
        else
        {
            /* update reSF_freeQ. */
            reSF_freeQ->ckSize -= objSz;
            /* create new allocated chunk. */
            alloc = (void *)reSF_freeQ + reSF_freeQ->ckSize - objSz;
            alloc->ckSize = objSz;
            alloc += sizeof(reSF_chk_hdr_t);
        }
    }

    /* allocation successfully, init reference and return.
       Reference will point to the starting address of data payload. */
    reSF_Ref[id] = (void *)alloc + sizeof(reSF_chk_hdr_t);
    alloc->ckRef = (uint16_t *)&reSF_Ref[id];

    /* Add this allocated object into the list */
    #if KDEBUG_DEMO
    /* store the thread id into the high byte of "ckSize".
      Note that "ckRef" in "reSF_chk_hdr_t" is needed for the memory free operation.
*/
    alloc->thrd_id = curThrd->thrd_id;
    /* if reSF_allocQ is empty. */
    if(reSF_allocQ == NULL)
    {
        reSF_allocQ =  alloc;
        alloc->next = NULL;
    }
    else
    {
        /* insert into the header */
        alloc->next = reSF_allocQ;
        reSF_allocQ = alloc;
    }
    #endif // KDEBUG_DEMO

    return alloc->ckRef;
}
```

```
/**
 * @brief Memory allocated by sequential fit (reactive fragments assembled).
 *  1st time SF allocation from the free memory list.
 *
 * \param objSz      Required size to be allocated.
 * \return       Starting address of the allocated chunk.
 */
void*
mem_alloc_proc(uint8_t objSz)
{
   reSF_chk_hdr_t *ck, *alloc = NULL;
   uint8_t splitSz = 0;

   /* get the split size. If the free chunk size is larger then this, split it.
*/
   #if !KDEBUG_DEMO
   splitSz = objSz+sizeof(reSF_chk_hdr_t)+MIN_PAYLOAD_SIZE;
   #else
   splitSz = objSz;
   #endif

   /* no available memory left. */
   if(reSF_freeQ == NULL)   return NULL;

   /* start allocation from the queue tail. */
   ck = reSF_freeQ->prev;
   /* search from the freed chunk queue. */
   do
   {
      /* find an available entry. */
      #if !KDEBUG_DEMO
      if((ck->ckSize >= objSz) && (ck->ckSize <= splitSz))
      {
         /* initialization and return.Total size "splitSz" will be allocated. */
         ck->ckSize = splitSz;
         /* delete ck from the freed chunk queue. */
         dlst_del((dlist **)(&reSF_freeQ), (dlist *)ck);
         return ck;
      }
      #else
      if(ck->ckSize == objSz)
      {
         /* delete ck from the freed chunk queue. */
         dlst_del((dlist **)(&reSF_freeQ), (dlist *)ck);
         return ck;
      }
```

```
        #endif

        /* split operation is required as the chunk size
           is larger than the required one. */
        if(ck->ckSize > splitSz)
        {
            /* split a piece from the bottom of this chunk. */
            alloc = (reSF_chk_hdr_t *)((uint16_t)ck + ck->ckSize - objSz);
            alloc->ckSize = objSz;
            /* new chunk update. */
            ck->ckSize -= objSz;
            return alloc;
        }

        /* check the next one */
        ck = ck->prev;
    } while(ck != reSF_freeQ->prev);

    return NULL;
}



/**
 * @brief fragment assembling for SF allocation
 * Assemble all the fragments and update the references.
 */
void
fragment_assemble(void)
{
    HAS_CRITICAL_SECTION;
    reSF_chk_hdr_t *frgmCk = reSF_freeQ->prev, *p;
    uint8_t *mvFrom, *mvTo;
    uint8_t mvSize, i;

    /* if no fragments, return directly. */
    if(frgmCk == reSF_freeQ) return;

    /* assign initialized "mvTo" address. */
    mvTo = (uint8_t *)((uint16_t)frgmCk + frgmCk->ckSize - 1);

    /* assemble all fragments one by one. */
    for(; frgmCk != reSF_freeQ; )
    {
        /* assign mvStart. */
        mvFrom = (uint8_t *)((uint16_t)frgmCk - 1);
        /* get the size to be moved. */
       mvSize = (uint16_t)frgmCk - ((uint16_t)frgmCk->prev + frgmCk->prev->ckSize);

        ENTER_CRITICAL_SECTION;
        /* chunks will be moved.
           Starting from chunk "frgmCk - mvSize", Ending to chunk "frgmCk".
           update the references of these chunks.
    Update value is the size between "mvTo" and "frgmCk", but not the "mvSize". */
        for(p  =  (reSF_chk_hdr_t  *)((uint16_t)frgmCk  -  mvSize);  p  !=  frgmCk;  p 
=(reSF_chk_hdr_t *)((uint16_t)p + p->ckSize))
            *(p->ckRef) = *(p->ckRef) + ((uint16_t)mvTo + 1 - (uint16_t)frgmCk);

        /* get next free chunk before data shifting. */
        frgmCk = frgmCk->prev;

        /* shift the chunks to clean up the memory fragments.
           don't change the value of "to" since it will be used for the next step.
           value of "mvTo" has been updated after this shifting.  */
        for(i = 0; i < mvSize; i++)
            *mvTo-- = *mvFrom--;
        LEAVE_CRITICAL_SECTION;
    }
}


/**
 * @brief Free a chunk
 *     Add the freed chunk into the free list reSF_freeQ.
 *     If two freed objects are adjacent, coalesce them.
 *     Release the reference after a chunk is released.
 *
 * \param memRF  Reference of the object to be released.
 *
 */
void
mem_free(uint16_t *memRF)
{
    reSF_chk_hdr_t *chuk, *ck = reSF_freeQ;

    /* locates the chunk header position. */
    chuk = (reSF_chk_hdr_t *)(*memRF - sizeof(reSF_chk_hdr_t));

    /* remove this one from the allocated list "reSF_allocQ". */
    #if KDEBUG_DEMO
    reSF_chk_hdr_t *lst;
    /* delete from the header */
    if(reSF_allocQ == chuk)  reSF_allocQ = chuk->next;
    /* delete the free item */
    for(lst = reSF_allocQ; lst != NULL; lst = lst->next)
    {
        if(lst->next == chuk)
            lst->next = chuk->next;
    }
    #endif // KDEBUG_DEMO


    /* If the queue is empty. */
    if(reSF_freeQ == NULL)  {
        reSF_freeQ = chuk;
        reSF_freeQ->prev = reSF_freeQ->next = reSF_freeQ;
        return;
    }

    /* locate the insertion position. */
    while(ck < chuk)  {
        /* check next until find the available position. */
        ck = ck->next;
        /* if comes to the ends, then break, and will insert to the queue tail. */
        if(ck == reSF_freeQ) break;
    }

    /* insert "chuk" in front of "ck". */
    dlst_insert((dlist *)chuk, (dlist *)ck);
    /* update the queue  head. */
    if(chuk < reSF_freeQ)       reSF_freeQ =  chuk;

    /* debug information, to clear the data of the freed chunk. */
    #if DEBUG_SUPPORT
        memSFfree_debug(chuk);
    #endif

    /* coalesce two free chunks if they are adjacent.
       firstly, upper-address merging, merge "chuk & chuk->next".
       later, lower-address merging, merge "chuk->prev & chuk". */
    dlst_merge((dlist **)(&reSF_freeQ), (dlist *)chuk, (dlist *)chuk->next);
    dlst_merge((dlist **)(&reSF_freeQ), (dlist *)chuk->prev, (dlist *)chuk);

    /* release the reference. */
    *memRF = 0;
}
```

```
#if DEBUG_SUPPORT
#if MEM_REACTIVE_SF
/**
 * @brief Debug for SF allocation, clear the data in the new freed chunk.
 */
void
memSFfree_debug(reSF_chk_hdr_t *chuk)
{
   uint8_t i, *p = (uint8_t *)((uint16_t)chuk + sizeof(reSF_chk_hdr_t));
   for(i = 0; i < (chuk->ckSize - sizeof(reSF_chk_hdr_t)); i++)
      *p++ = 0;
};

#endif
#endif
#endif
```

2）文件 mem_reactive_SF.h

```
/**
 * @file mem_reactive_SF.h
 *
 * @brief  header for mem_reactive_SF.c
 *
 * @author    Computer Science and Technology
 * @author    Wuhan University of Technology
 */

/* Prevent double inclusion */
#ifndef _MEM_REACTIVE_SF_H_
#define _MEM_REACTIVE_SF_H_

/* === Includes ============================================================= */
#include "board.h"
#include "kernel.h"
#include "sys_config.h"
#include "qlist_proc.h"

#if MEM_REACTIVE_SF
/* === Macros =============================================================== */
/* minimum size of a new split chunk should be larger than MIN_PAYLOAD_SIZE. */
#define MIN_PAYLOAD_SIZE 8
/* reference number, maximum allocation. */
#define REF_NUM      20
/* debug option for fragment assembling test. */
#define FRAG_ASSMBL_DEBUG   0
```

```
/* === Types ================================================================ */
/* double link list is used as memory coalescence is needed. */
typedef __ALIGNED2 struct reSF_chk_hdr
{
    struct reSF_chk_hdr *prev;  /* to previous free chunk */
    struct reSF_chk_hdr *next;  /* to next free chunk */
    uint16_t ckSize;         /* size of whole chunk including the chuk_hdr */
    uint16_t *ckRef;         /* the address of the reference to this chunk. */
    #if KDEBUG_DEMO
    uint8_t thrd_id;
    #endif
} reSF_chk_hdr_t;

/* === GLOBALS ============================================================= */
extern reSF_chk_hdr_t* reSF_freeQ;
extern uint16_t* reSF_Ref[];
extern reSF_chk_hdr_t* reSF_allocQ;

extern uint16_t* mem_alloc(uint8_t objSz);
extern void* mem_alloc_proc(uint8_t objSz);
extern void fragment_assemble(void);
extern void mem_free(uint16_t *memRF);
extern void memSFfree_debug(reSF_chk_hdr_t *chuk);

/* === Prototypes =========================================================== */

#endif
#endif
```

专题六、时钟管理

1）文件 timer_RCV.c

```
/**
 * @file timer_RCV.c
 *
 * @brief software timers implemented by using RCV (relative counter value).
 *
 * The timers can be classified into two modes: one-slot timer and periodic timer.
 * For one-slot timer, it will be deleted once it is expired.
 * For periodic timer, it will be reset and restarted after expired.
 *
 * Since callback function is called from the interruption service routine (ISR),
 * its execution time should be short. If the execution time of a callback is long,
 * this callback needs to be executed outside the interruption context. That is,
 * executed in asynchronous mode. In this case, only a execution request is posted
inside the ISR.
 *
 * @author    Computer Science and Technology
 * @author    Wuhan University of Technology
 */

/* === INCLUDES ============================================================ */
#include "typedef.h"
#include "sys_config.h"
#include "evt_driven_sched.h"
#include "multithreading_sched.h"
#include "kdebug.h"

#if TIMER_RCV
#include "timer_RCV.h"
/* === TYPES =============================================================== */


/* === MACROS ============================================================== */


/* === GLOBALS ============================================================= */
timer_t *sysTimerQhead = NULL; // head of the system timer queue.

/* === PROTOTYPES ========================================================== */


/* === IMPLEMENTATION ====================================================== */
```

```
/**
 * @brief Hardware Periodical Interrupt Timer (PIT) handler
 *
 * This is the interrupt service routine for hardware PIT.
 * It is triggered when a timer expires.
 */
ISR(TIMER4_COMPA_vect)
{
    HAS_CRITICAL_SECTION;
    ENTER_CRITICAL_SECTION;
    /* SysTimer Service. */
    timerService();
    LEAVE_CRITICAL_SECTION;
}

/**
 * @brief Interrupt service routine for the system timers.
 *
 * This is the interrupt service routine for timer.
 * It checks all the timers in the system timer queue,
 * if a timer is fired, the related timer callback will be executed.
 */
void
timerService(void)
{
    timer_t *t;

    /* search for expired timers and take actions */
    for(t = sysTimerQhead; t != NULL; t = t->next)
    {
        /* if counter value smaller than APPTIMERINTERVAL, timer will be fired. */
        if(t->interval < APPTIMERINTERVAL)
        {
            /* remove this fired timer from the timer queue. */
            stopTimer(t);
            /* if the timer is a periodical one, add it into the timer queue again.
*/
            if (t->mode == TIMER_REPEAT_MODE)  startTimer(t);

            /* When timer is fired, call the timer callback function. */
            t->callback(t->cb_data);
        }
        else  /* decrease the counter value. */
            t->interval -= APPTIMERINTERVAL;
    }
}
```

```
/**
 * @brief Starts a timer.
 */
int
startTimer(timer_t *Timer)
{
    if (!Timer)
    return -1;
    if (true == isAlreadyInQueue((sQList *)sysTimerQhead, (sQList *)Timer))
    return 0;

    /* insert the new timer to the head of timer queue. */
    Timer->next = sysTimerQhead;
    sysTimerQhead = Timer;

    return 0;
}

/**
 * @brief Stops the timer.
 */
int
stopTimer(timer_t *Timer)
{
    timer_t *prev = 0;
    timer_t **t = &Timer;

    if (!Timer)   return -1;

    /* when "Timer" is not header, get its previous one. */
    if (sysTimerQhead != *t)
    {
        if  (!(prev  =  (timer_t  *)findPrevEntry((sQList  *)sysTimerQhead,  (sQList
*)Timer)))
        return -1;
    }
   RemoveEntryFromQ((sQList **)(&sysTimerQhead), (sQList *)prev, (sQList *)Timer);
    return 0;
}
#endif /* #if TIMER_RCV */
```

2）文件timer_RCV.h

```
#if TIMER_RCV
/* === Macros ================================================================ */
/* timer modes.
   There are two timer modes: periodical timer and one-slot timer.
   For one-slot timer, after it is fired, it will be deleted from the timer queue.
   For periodical timer, after fired, it will be added into the timer queue again.
*/
typedef enum
{
  TIMER_REPEAT_MODE,
  TIMER_ONE_SHOT_MODE,
} TimerMode_t;


/** \brief RCV Timer structure */
typedef void (*time_cb_t)(void *data);
__ALIGNED2 typedef struct _Timer_t
{
    struct _Timer_t *next;
    uint32_t interval;      /* timer counter. */
    time_cb_t callback;     /* callback function when timer is fired. */
    void *cb_data;          /* data used by timer callback. */
    uint8_t mode;        /* timer mode: TIMER_ONE_SHOT_MODE or TIMER_REPEAT_MODE.
*/
} timer_t;

/* === Types ================================================================= */


/* === GLOBALS ============================================================== */
extern timer_t *timerQhead; // head of Timer list

/* === Prototypes =========================================================== */
extern void timerService(void);
extern bool isTimerAlreadyStarted(timer_t *Timer);
extern int startTimer(timer_t *Timer);
extern int stopTimer(timer_t *Timer);

#endif /* TIMER_ACV */
#endif
```

专题七、链表操作库函数

1）文件 qlist_proc.c

```
/**
 * @file qlist_proc.c
 *
 * @brief  kernel library for the queue list operation.
 *
 * @author    Computer Science and Technology
 * @author    Wuhan University of Technology
 */

/* === INCLUDES ============================================================ */
#include "board.h"
#include "typedef.h"
#include "kdebug.h"
#include "qlist_proc.h"

/* === IMPLEMENTATION ====================================================== */
/**
 * @brief Double link queue delete operation.
 * \param Qhead     Pointer to the head of the queue.
 * \param list      The entry to be deleted from this queue.
 *
 * Update the queue header when required.
 */
INLINE void
dlst_del(dlist **Qhead, dlist *list)
{
    HAS_CRITICAL_SECTION;
    /* if only one exist, the queue becomes empty. */
    if(list->next == list)
    {
        /* update the header. */
        *Qhead = NULL;
        return;
    }

    ENTER_CRITICAL_SECTION;
    list->prev->next = list->next;
    list->next->prev = list->prev;
    LEAVE_CRITICAL_SECTION;
    /* update the queue header. */
    if(*Qhead == list)
        *Qhead = (*Qhead)->next;
}

/**
 * @brief Double link queue insertion operation.
 * \param list      The entry to be inserted into.
 * \param pos   The position for "list" to be inserted.
 *
 * Insert "list" to the front of "pos".
 * The queue head is not updated in this function,
 * users should be aware of this.
 */
void
dlst_insert(dlist *list, dlist *pos)
{
    HAS_CRITICAL_SECTION;
    ENTER_CRITICAL_SECTION;
    list->next = pos;
    list->prev = pos->prev;
    pos->prev->next = list;
    pos->prev = list;
    LEAVE_CRITICAL_SECTION;
}

/**
 * @brief Merge two queue lists if they are adjacent.
 * \param Qhead     Pointer to the head of the queue.
 * \param listA     The 1st entry to be merged.
 * \param listB     The 2nd entry to be merged.
 *
 * Merge listB into listA, and then delete listB.
 */
void
dlst_merge(dlist **Qhead, dlist *listA, dlist *listB)
{
    /* merge listB into listA, and then delete listB. */
    if(((uint16_t)listA + listA->size) == (uint16_t)listB)  {
        /* delete "listB" */
        dlst_del(Qhead, (dlist *)listB);
        /* update "listA" size */
        listA->size += listB->size;
    }
}

/**
 * @brief Search "item" from the single link queue with the header "Qhead".
 * \param Qhead     Pointer to the head of the queue.
 * \param item      Search this item from this queue.
 * \return   Return true if found, or false if not found.
 *
 */
bool
isAlreadyInQueue(sQList *Qhead, sQList *item)
{
   bool result = false;
   sQList *p = Qhead;

   while (NULL != p) {
      if (p == item)  {
         result = true;
         break;
      }
      p = p->next;
   }
   return result;
}

/**
 * @brief Find the entry previous to "item" in the single link queue "Qhead".
 * \param Qhead     Pointer to the head of the queue.
 * \param item      Return the entry ahead of "item" in this queue.
 * \Return      Return the previous entry.
 */
sQList
*findPrevEntry(sQList *Qhead,  sQList *item)
{
   sQList *t = Qhead;

   for (; t ;)
   {
      if (t->next == item)
      return t;
      t = t->next;
   }
   return NULL;
}

/**
 * @brief Removes the entry "item" from the single link queue.
 * \param head      Pointer to the head of the queue.
 * \param prev      The previous entry before "item".
 * \param item      Entry to be removed.
 *
 * Use "findPrevEntry" to get the entry previous to "item" firstly.
 */
void
RemoveEntryFromQ(sQList **head, sQList *prev, sQList *item)
{
    if (item == *head)
        /* removing first element of list */
        *head = item->next;
    else
        prev->next = item->next;
    item->next = 0;
}
```

2）文件 qlist_proc.h

```
/* Prevent double inclusion */
#ifndef _QLIST_PROC_H_
#define _QLIST_PROC_H_

/* === INCLUDES ============================================================ */
#include "board.h"
#include "typedef.h"
#include "kdebug.h"

/* === TYPES =============================================================== */
/* double link queue list. */
typedef struct dList
{
    struct dList *prev;
    struct dList *next;
    uint16_t size;
} dlist;

/* single link queue list. */
typedef struct sList
{
    struct sList *next;
} sQList;
```