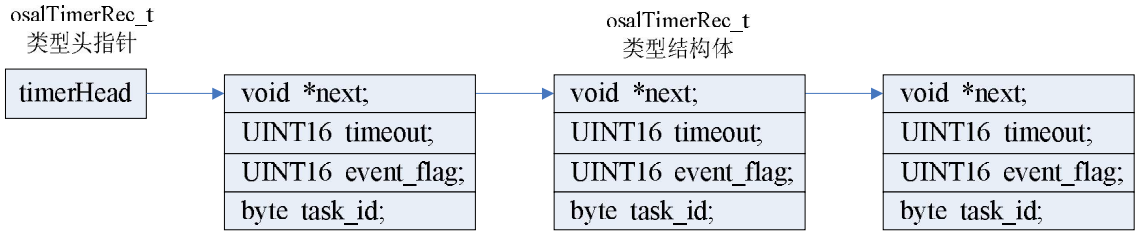
关于 ZStack OSAL 中的系统定时任务处理

与系统定时器相关的任务，并不是一个独立的任务抽象，其只是用来设置与时间相关的任务事件的一种机制。所有的定时任务使用系统定时器（在 ZStack 中使用定时器 4）定时。

定时任务链表：



链表中每个节点的类型为：

```
typedef struct
{
  void *next;
  UINT16 timeout;
  UINT16 event_flag;
  byte task_id;
} osalTimerRec_t;
```

可见链表其实为一个任务（task）和事件（event）的链表，并不是定时器的链表，而是系统定时器的任务链表，timeout 就是定时的时间值。当定时时间到，就会设置 task_id 对应的任务。定时如何实现见下文讲述。

首先看该链表是如何创建的，也就是如何向链表中添加一个新的定时任务。使用下列函数：

```
byte osal_start_timerEx( byte taskID, UINT16 event_id, UINT16 timeout_value )
{
  halIntState_t intState;
  osalTimerRec_t *newTimer;

  HAL_ENTER_CRITICAL_SECTION( intState );  // Hold off interrupts.

  // Add timer
  newTimer = osalAddTimer( taskID, event_id, timeout_value );   //添加定时任务
  if ( newTimer )
  {
#ifdef POWER_SAVING
    // Update timer registers
    osal_retune_timers();
    (void)timerActive;
#endif
    // Does the timer need to be started?
```

```
    if ( timerActive == FALSE )   //若系统定时器之前未启动，则启动之
    {
      osal_timer_activate( TRUE );
    }
  }

  HAL_EXIT_CRITICAL_SECTION( intState );  // Re-enable interrupts.

  return ( (newTimer != NULL) ? ZSUCCESS : NO_TIMER_AVAIL );
}
```

其中调用了函数 salAddTimer(byte task_id, UINT16 event_flag, UINT16 timeout) 来添加定时任务，该函数首先查找表中是否已经有包含 task_id 任务和 event_flag 事件的定时任务，若没有，便分配一个 osalTimerRec_t 类型的空间，根据函数实参设置各成员，并追加的链表尾。并且若系统定时器之前未启动，则启动之

函数 void osalDeleteTimer(osalTimerRec_t *rmTimer)用来从链表中删除一个定时任务。

上文已经讲了任务链表的创建和系统定时器的启动，现在再回过头来看系统定时器的初始化和配置。

在 main 函数中，会调用 osal_init_system()函数初始化操作系统，在此初始化函数中，就会调用系统定时器初始化函数 osalTimerInit()：

```
void osalTimerInit( void )
{
  // Initialize the rollover modulo
  tmr_count = TICK_TIME; //系统定时器的定时值，宏 TICK_TIME=1000，单位 us
  tmr_decr_time = TIMER_DECR_TIME; //链表中各任务的定时值 timeout 在每次系
    //统定时器比较事件发送时的递减值，宏 TIMER_DECR_TIME=1，单位为 ms
  osal_timer_activate( false );     //初始化系统定时器为关闭状态
  timerActive = false;
  osal_systemClock = 0;
}
```

其中，osal_timer_activate(false)最后调用的是 hal_timer.c 源文件中的 HalTimerStop (OSAL_TIMER)函数，并可见，系统定时器 OSAL_TIMER 实际上是 8 位的物理定时器 4。

上面的函数只对系统定时器初始化了，但并未配置。

在 main 函数中，还调用了 InitBoard(OB_COLD)函数，在 InitBoard 函数中，便对系统定时器进行了配置：

```
OnboardTimerIntEnable = FALSE;
HalTimerConfig (OSAL_TIMER,                    // 8bit timer2
          HAL_TIMER_MODE_CTC,                 // Clear Timer on Compare
          HAL_TIMER_CHANNEL_SINGLE,           // Channel 1 - default
          HAL_TIMER_CH_MODE_OUTPUT_COMPARE,   // Output Compare mode
          OnboardTimerIntEnable,              // Use interrupt
          Onboard_TimerCallBack);             // Channel Mode
```

配置了通道和工作模式，不使用中断，设置了回调函数 Onboard_TimerCallBack，这是定时任务链表实现定时功能的关键函数：

```
void Onboard_TimerCallBack ( uint8 timerId, uint8 channel, uint8 channelMode)
```

```
{
  if ((timerId == OSAL_TIMER) &&
                 (channelMode == HAL_TIMER_CH_MODE_OUTPUT_COMPARE))
  {
    osal_update_timers();
  }
}
```

其中就调用了任务定时值更新函数 osal_update_timers()，其又调用了函数 osalTimerUpdate()：

```
static void osalTimerUpdate( uint16 updateTime )
{
  halIntState_t intState;
  osalTimerRec_t *srchTimer;
  osalTimerRec_t *prevTimer;
  osalTimerRec_t *saveTimer;

  HAL_ENTER_CRITICAL_SECTION( intState );  // Hold off interrupts.

  // Update the system time
  osal_systemClock += updateTime;   //xl:更新系统时钟，即加上系统定时器的定时周期

  // Look for open timer slot
  if ( timerHead != NULL )
  {
    // Add it to the end of the timer list
    srchTimer = timerHead;
    prevTimer = (void *)NULL;

    // Look for open timer slot
    while ( srchTimer )
    {
      // Decrease the correct amount of time
      if (srchTimer->timeout <= updateTime)
        srchTimer->timeout = 0;
      else
        srchTimer->timeout = srchTimer->timeout - updateTime;

      // When timeout, execute the task
      if ( srchTimer->timeout == 0 )
      {
        osal_set_event( srchTimer->task_id, srchTimer->event_flag );    //xl:若定时时间到了，便设置对应的任务和事件

        // Take out of list
        if ( prevTimer == NULL )
          timerHead = srchTimer->next;
```

```
      else
        prevTimer->next = srchTimer->next;

      // Next
      saveTimer = srchTimer->next;

      // Free memory
      osal_mem_free( srchTimer );

      srchTimer = saveTimer;
    }
    else
    {
      // Get next
      prevTimer = srchTimer;
      srchTimer = srchTimer->next;
    }
  }

#ifdef POWER_SAVING
  osal_retune_timers();
#endif
  }

  HAL_EXIT_CRITICAL_SECTION( intState );  // Re-enable interrupts.
}
```

在该函数中，首先查询链表中的每个任务，将每个任务的 timeout 减去系统定时器的定时周期 tmr_decr_time（1 ms），若不够减，则置 0。然后判断 timeout 是否等于 0（即定时时间是否到了），若是，调用函数 osal_set_event(srchTimer->task_id, srchTimer->event_flag)设置相应的任务和事件，指示系统处理之。

现在系统定时器是配置好了，但还有个问题，由于系统定时器未使用中断方式（OnboardTimerIntEnable = FALSE），那回调函数在什么时候被调用来更新 timeout 呢？

还是回到 main 函数，其中最后调用了 osal_start_system()系统启动函数：

```
void osal_start_system( void )
{
#if !defined ( ZBIT )
  for(;;)  // Forever Loop
#endif
  {
    uint8 idx = 0;

    Hal_ProcessPoll();  // This replaces MT_SerialPoll() and osal_check_timer().
```

```
    do {
      if (tasksEvents[idx])  // Task is highest priority that is ready.
      {
        break;
      }
    } while (++idx < tasksCnt);
//    IndexSta=idx;
    if (idx < tasksCnt)
    {
      uint16 events;
      halIntState_t intState;

      HAL_ENTER_CRITICAL_SECTION(intState);
      events = tasksEvents[idx];
      tasksEvents[idx] = 0;  // Clear the Events for this task.
      HAL_EXIT_CRITICAL_SECTION(intState);

      events = (tasksArr[idx])( idx, events );

         HAL_ENTER_CRITICAL_SECTION(intState);
      tasksEvents[idx] |= events;  // Add back unprocessed events to the current
task.
         HAL_EXIT_CRITICAL_SECTION(intState);
      }
#if defined( POWER_SAVING )
    else  // Complete pass through all task events with no activity?
    {
      osal_pwrmgr_powerconserve();  // Put the processor/system into sleep
    }
#endif
    }
}
```

此函数不会返回，包含一个永久 for 循环，是整个 ZStack 的主循环。for 循环做两个工作：一是调用函数 Hal_ProcessPoll()轮询定时器、串口等设备；二是监听任务事件。

关于后者操作系统的任务事件和消息队列的处理见笔者的另一篇文档。

此处要讲述的是前者 Hal_ProcessPoll()函数的定时器轮询，该函数调用了定时器轮询函数 HalTimerTick()：

```
void HalTimerTick (void)
{
  if (!halTimerRecord[HW_TIMER_1].intEnable)
  {
    halProcessTimer1 ();
  }

  if (!halTimerRecord[HW_TIMER_3].intEnable)
```

```
  {
    halProcessTimer3 ();
  }

  if (!halTimerRecord[HW_TIMER_4].intEnable)
  {
    halProcessTimer4 ();
  }
}
```

该函数只会处理中断未使能的定时器。

据上文知道，系统定时器使用的是物理定时器 4 即 HW_TIMER_4，由于系统定时器未使能中断，所以最后一个 if 语句会调用定时器 4 的处理函数 halProcessTimer4 ()：

```
void halProcessTimer4 (void)
{
  if (halTimerRecord[halTimerRemap(HAL_TIMER_2)].channelMode ==
HAL_TIMER_CH_MODE_OUTPUT_COMPARE)
  {
    if (TIMIF & TIMIF_T4CH0IF)
    {
      TIMIF &= ~(TIMIF_T4CH0IF);
      halTimerSendCallBack (HAL_TIMER_2, HAL_TIMER_CHANNEL_A,
HAL_TIMER_CH_MODE_OUTPUT_COMPARE);
    }
    if (TIMIF & TIMIF_T4CH1IF)
    {
      TIMIF &= ~(TIMIF_T4CH1IF);
      halTimerSendCallBack (HAL_TIMER_2, HAL_TIMER_CHANNEL_B,
HAL_TIMER_CH_MODE_OUTPUT_COMPARE);
    }
  }
  else if (halTimerRecord[halTimerRemap(HAL_TIMER_2)].channelMode ==
HAL_TIMER_CH_MODE_OVERFLOW)
  if (TIMIF & TIMIF_T4OVFIF)
  {
    TIMIF &= ~(TIMIF_T4OVFIF);
    halTimerSendCallBack (HAL_TIMER_2, HAL_TIMER_CHANNEL_SINGLE,
HAL_TIMER_CH_MODE_OVERFLOW);
  }
}
```

定时器 4 即系统定时器使用的是通道 1 的输出比较模式，所以在定时器比较发生置通道 1 的中断标志后，if (TIMIF & TIMIF_T4CH1IF)语句中就会调用系统定时器的回调函数，即前文讲到的函数 Onboard_TimerCallBack()，来更新定时任务链表中的各个任务的 timeout 值，这就实现了定时任务链表中各任务的计时。

功德圆满了！

本文讲述了，定时任务链表，系统定时器的初始化、配置、启动和轮询系统定时器，以及没纸了写不