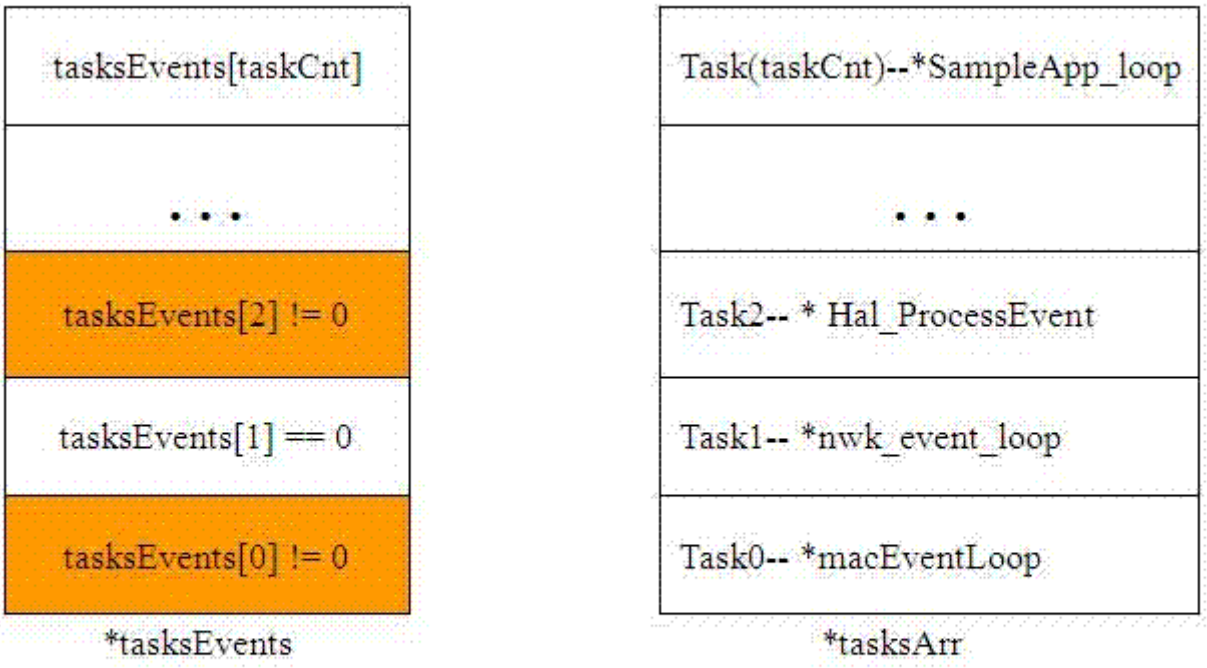
OSAL 系统框架专题 2010-4-15 2:47:00

9. TI 协议栈所用系统框架探讨。
51 的系统往往不是太大，但是几十 K 的程序，也足以让一个初学者望而却步。我们首先忽略 C 语言本身的难度，光是系统框架也让生手读起来很吃力，再加上这种到处是 API 跟"define"的程序，还没有正式学习协议部分就已经让人在丛林中“迷路”了。

在接下来的一段时间内，我会以 TI 所用的系统框架为主线进行学习，希望大家共同探讨。。。

在层层迷雾中摸索了两天，终于拨云见日，那个心情啊，怎一个“爽”字了得~~~

可是怎么能把这么复杂的一个问题讲得清楚呢？嗯。。。还是先上图吧



图一、 主循环函数处理机制

注：为了便于直观，以下涉及到数据地址的地方都是由上而下，地址由高变低

第 1 节、各个任务是如何被调用到的？
我们还是先从 main()函数开始，看看各个任务之间是如何协调工作的。
插播一句广告：在一切都看不清的时候，忽略次要，看主要因素 —— by outm

an from Zigbeetech

我们直接进入主循环的核心部分，看一下系统中的几个主要的任务是如何被调用，并开始自己的使命的？

看一段程序的时候，往往要从它的数据结构入手。我们先看一下，主循环中的两个关键数组，*tasksEvents 与*tasksArr，从图一中我们可以看出来，tasksEvents 这个数组存放的是从序号为 0 到 tasksCnt，每个任务在本次循环中是否要被运行，需要运行的任务其值非 0（用橙色表示），否则为 0。而 tasksArr 数组则存放了对应每个任务的入口地址，只有在 tasksEvents 中记录的需要运行的任务，在本次循环中才会被调用到。——这节讲完了。。。

又有同学举手？什么？还没明白？恩。。。好像是不能讲这么短的。。。
那好吧，把 main 函数贴过来，我们一点一点看
初始化过程“先不管”，我们先看主循环（dead loop）

```
for(;;)    // Forever Loop
{
      uint8 idx = 0;
      Hal_ProcessPoll();    // 先不管 1

      do {
          if (tasksEvents[idx])    // 寻找最高优先级的任务来运行
          {
              break;
          }
      } while (++idx < tasksCnt);

      if (idx < tasksCnt)
      {
          uint16 events;
          halIntState_t intState;

          HAL_ENTER_CRITICAL_SECTION(intState);
          events = tasksEvents[idx];
          tasksEvents[idx] = 0;    // 本任务运行完了，要对其清空，为后面要运行的任务让路
          HAL_EXIT_CRITICAL_SECTION(intState);

          events = (tasksArr[idx])( idx, events );//最关键的一句话，如图一中，运行对应的任务

          HAL_ENTER_CRITICAL_SECTION(intState);
          tasksEvents[idx] |= events;    // 本任务可能没完全完成，如果
```

是这样，再次设置标志位，在下一次循环中继续执行
```
          HAL_EXIT_CRITICAL_SECTION(intState);
          }
      }
```

第 2 节、系统时间

我们知道，每个操作系统（虽然我不认为 OSAL 是一个标准的操作系统，但我们先这么叫着吧）都有一个“节拍”－tick，就像每一个“活人”都有心跳一样。那么 OSAL 的心跳有多快呢？－－1ms。当然这个速度是可以设置的，在 osal_timer_activate 函数中开启了系统节拍，用 TICK_TIME 来定义其速度
```
#define TICK_TIME     1000     // Timer per tick - in micro-sec
```
注意：这个 1000 是 micro-sec（微秒），而不是 milli－sec（毫秒）！我刚开始的时候就是误以为是 1000ms 而耽误了不少时间。
那这个心脏是怎么跳动起来的呢？

这得从“定时器”说起，由于本文的重点不是讲单片机基础的，如果对这个名字还陌生的同学,那还是回去先看看基础再来看这个吧。2430 有 4 个定时/计数器，其中 timer4 用来做系统计时。如果认为是 timer2 的同学请看一下 halTimerRemap 这个函数。在上述 osal_timer_activate 函数中，开启了系统计时，并将 timer4 的初始设为 TICK_TIME（1000），这样 timer4 就开始了从 1000 开始的减计数，减到 0 以后呢？寄存器 TIMIF 会产生一个溢出标志，那么它会立即产生中断并进入中断服务程序吗？不会的。

我们看一下第 1 节主函数里的“Hal_ProcessPoll();   // 先不管 1”（不管的东西早晚要管的，只是时间的问题而已）   这个函数里调用了 HalTimerTick，这个函数就是专门来检查是否有硬件定时器溢出的，如果有的话会调用 halTimerSendCallBack 这个函数，对溢出事件做处理。

回过头来说系统节拍，那 timer4 在计数满 1000（即 1ms）后做了些什么事呢，那我们看一下 halTimerSendCallBack 这个函数
```
void halTimerSendCallBack (uint8 timerId, uint8 channel, uint8 channelMode)
{
    uint8 hwtimerid;

    hwtimerid = halTimerRemap (timerId);

    if (halTimerRecord[hwtimerid].callBackFunc)
      (halTimerRecord[hwtimerid].callBackFunc) (timerId, channel, channelMode);
}
```
这里面调用了“callBackFunc”函数，也就是说每个定时器溢出后都有一个 callBackFunc 函数，它在哪里呢，我们再看一下 HalTimerConfig 这个函数，它可

以对每个定时器进行定义。那什么时候定义的呢？－－InitBoard，即板子上电初始化的时候就做了这个定义的。我说什么来？“先不管”的东西，“后要管”的。。。

我们看到 timer4 的 callBackFunc 函数是 Onboard_TimerCallBack，最终指向 osalTimerUpdate，这个函数厉害了~
从上面的分析中我们知道它是每 1ms 被调用一次的，这样它就为应用程序提供了一个 ms 计时器，应用程序所用的定时往往以 ms 为单位足够了，这样的话就不用另外再占用硬件计时器了，毕竟只有 4 个嘛。。。同时这个函数还提供了一个系统时钟－osal_systemClock，看看它能计时多久吧，它是“uint32”型的，也就是 2^32ms=49.7 天，怎么样？你不会让这个系统时钟 overflow 吧：）

第 3 节、系统的消息处理机制

结合第 1、2 节中的内容，让我们一起进入到系统最核心的部分－消息处理中来吧（这个句型怎么这么耳熟~~~）

第 1 节中我们说了，tasksEvents 数组存放了一个任务是否该被运行的序列，但是这个序列是如何产生的呢？如果了解了这个问题，那也就知道了 OSAL 系统的运作方式。
再插句广告：在浩如烟海的程序中搜索最重要的东西，就像大浪淘沙，其实也是蛮享受的一件事情－－by outman from zigbeetech

source insight "ctr+/"，整个项目搜索，我们发现了一个“osal_set_event”的函数是专门来设置 tasksEvents 的，但是似乎并不能帮到我们。继续搜！有两个很重要的地方引起了我们的注意：osalTimerUpdate 和 osal_msg_send 这两个函数

osalTimerUpdate，这个称得上“厉害”的函数还记得吧？它会去设置 tasksEvents？那不就是说，它可以让任务在主循环中被运行到？答对了，这就是它“厉害”的地方。。。那看看它运行任务的条件吧？

```
          // When timeout, execute the task
          if ( srchTimer->timeout == 0 )
          {
              osal_set_event( srchTimer->task_id, srchTimer->event_fla
g );
... ...
```

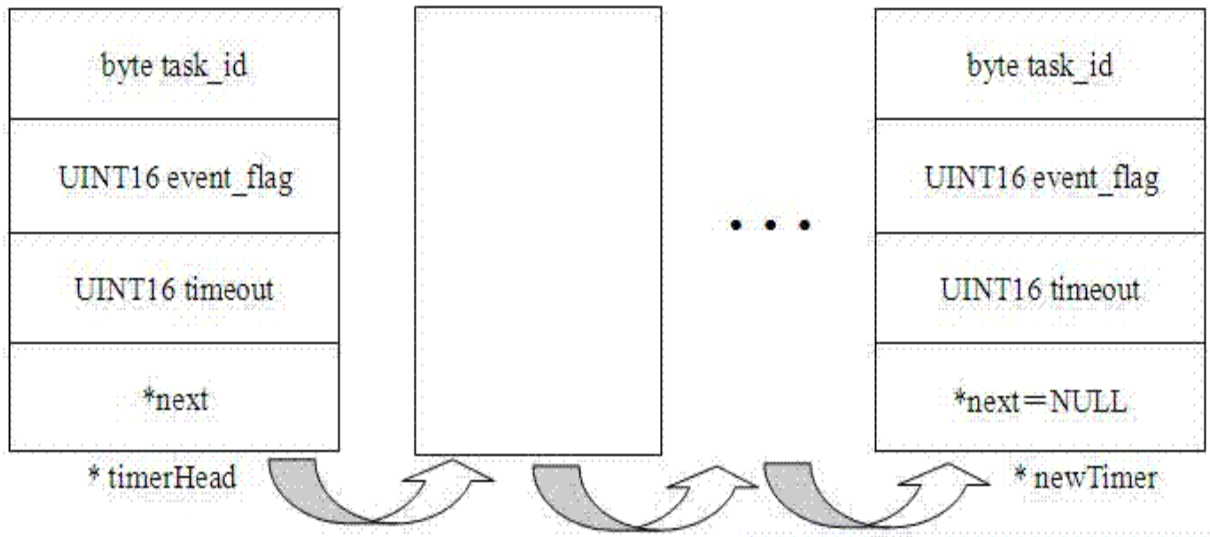
也就是说，计时器溢出－－恩。。。不多说了，我们埋个伏笔，先介绍另一个朋友－osal_start_timerEx，先看下它的自我介绍

```
/*********************************************************************
*
* @fn          osal_start_timerEx
*
* @brief
```

```
*
*     This is called to start a timer to expire in n mSecs.
*     When the timer expires, the calling task will get the specified
 event.
*
* @param     byte taskID - task id to set timer for
* @param     UINT16 event_id - event to be notified with
* @param     UNINT16 timeout_value - in milliseconds.
*
* @return    ZSUCCESS, or NO_TIMER_AVAIL.
*/
byte osal_start_timerEx( byte taskID, UINT16 event_id, UINT16 timeout
_value )
```

也就是说，它会开始一个 timeout_value(ms)的计时器，当这个计时器溢出时，则会对 taskID 这个 task，设置一个 event_id，让这个任务在后面的主循环中运行到，但是是怎么实现的呢？还是要请 osalTimerUpdate 来帮忙。。。

那位同学说啥？复杂了，听不懂？
唉，还是上图吧



图二、软件定时器数据链表

还是先从数据结构说起吧，不知道啥是“数据链表”的同学，把谭老师的书拿过来再读几遍。。。
这个表就是 osalTimerUpdate 函数的“任务表”，上面不是说过这个函数给应用程序提供了“软计时”了吗？就是体现在这里，osal_start_timerEx 通过 osalAddTimer 向链表里添加了“定时任务”，由 osalTimerUpdate 来以 ms 为单位对

这些“软定时器”减计数，溢出时，即调用 osal_set_event，实现主循环里对任务的调用。

好了，到此讲了上面提到的"set event"函数中的一个 osal_start_timerEx，还有一个更厉害的还在外面呢，osal_msg_send，这就渐入佳境，进入最重要的消息处理机制了。。。

-- by outman 2010-4-14 18:00　　下班啦，老婆在家等着回去一起做饭哪~~~晚上见~~~

为了更好地说明这个问题，还是拿一个具体的例子来讲比较直观。不过在这个笔记中，我尽量不涉及具体开发板，而讲一些通用的知识，因为这样会让更多的人受益。在 TI 官方 zstack 2006 中有 4 个例子，其中一个叫 GenericApp 最基本的通信的例程，如果没有安装 zstack 的同学可以到“本站专用下载贴”中下载。当然由于讲的是些比较通用的东西，所以手头有开发板的同学可以用自己的开发板来试验，效果更好。。。

在这样的通信例程中，一般会有一个按键触发，然后会和相邻的模块进行通信，当然由于这部分是讲 OSAL 的系统框架的，我们先不涉及通信的内容，只是看一下按键是如何产生的，及如何调用相应的接口程序。

按 OSAL 的模块定义，按键可能在哪层来？硬件服务相关的，恩。。。是不是在 HAL 层呢？到 Hal_ProcessEvent 看看？有个 HalKeyPoll 函数不是？恩，这就是检测按键的地方~~不过，我可不是像上面这样这么容易猜出来的，这几句话足足用了我大半个钟头呢。。。过程我不细说了，有兴趣的话我可以再补充一下。

在 HalKeyPoll 函数中，无论按键是 ADC 方式，或者是扫描 IO 口的方式，最后都会生成一个键值 keys，然后通过下面的语句来调用按键服务程序

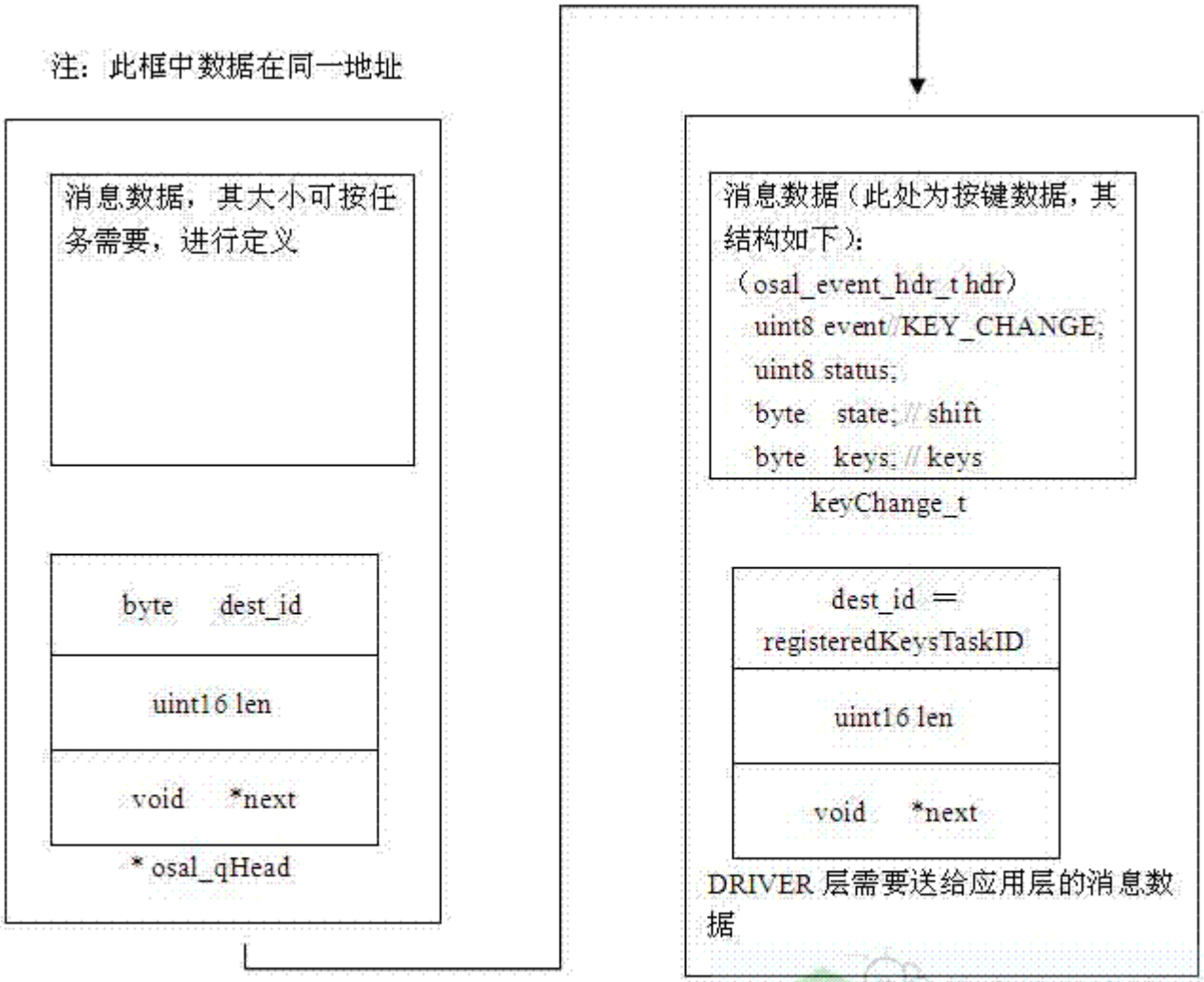
```
    /* Invoke Callback if new keys were depressed */
    if (keys && (pHalKeyProcess))
    {
      (pHalKeyProcess) (keys, HAL_KEY_STATE_NORMAL);
    }
```

这里调用的服务程序，在 InitBoard 中被初始化为 OnBoard_KeyCallback，这个函数又通过 OnBoard_SendKeys 运行下面语句

```
    {
      // Send the address to the task
      msgPtr = (keyChange_t *)osal_msg_allocate( sizeof(keyChange_t) );
      if ( msgPtr )
      {
          msgPtr->hdr.event = KEY_CHANGE;
          msgPtr->state = state;
          msgPtr->keys = keys;
```

```
        osal_msg_send( registeredKeysTaskID, (uint8 *)msgPtr );
      }
      return ( ZSuccess );
    }
```

下面我们就看下 osal_msg_send 是如何向上级应用程序发送消息的。终于要讲消息量的数据结构了，好像绕得有点远。。。。还是先上图



图三、消息数据链表

在理解了消息量的数据链表后，再来理解 osal_msg_send 里的语句就不难了

OSAL_MSG_ID(msg_ptr) = destination_task;//设置消息数据对应是属于哪个任务的

```
// 将要发送的消息数据链接到以 osal_qHead 开头的数据链表中
osal_msg_enqueue( &osal_qHead, msg_ptr );
```

```
// 通知主循环有任务等待处理
osal_set_event( destination_task, SYS_EVENT_MSG );
```

这样用户任务 GenericApp_ProcessEvent 就收到一个按键的处理任务，并通过 GenericApp_HandleKeys 来执行相应的操作。

好了，现在应该对 OSAL 的消息处理机制有个了解了吧？我们再来复习一下这个按键的处理过程：任务驱动层 Hal_ProcessEvent 负责对按键进行持续扫描，发现有按键事件后 OnBoard_KeyCallback 函数向应用层 GenericApp_ProcessEvent 发送一个有按键需要处理的消息，最终由 GenericApp_HandleKeys 来负责执行具体的操作。

让我们再回到最初的问题，任务处理表 tasksEvents 是怎么被改动的呢？初始化程序、其他任务或者本任务主要通过下面几种方式对其操作：
1、设置计时器，当其溢出时，触发事件处理
2、直接通过任务间的消息传递机制触发
3、...(等我想到了再补充)