Events

There are 2 primary objects used in the event system: EventSource's and EventListener's. There will be a 1-to-1 relationship between EventSources and EventListeners. Moreover, for every EventSource a thread is interested in, a unique EventListener is required to associate the thread to the EventSource.

EventSource

An EventSource is a fairly simple/dumb object whose only function is to keep track of all the EventListeners that have been registered with it.

EventListener

An EventListener is a more complex object that allows a thread to be notified whenever the EventSource it is registered with has signaled. Every thread can 'listen' for (sizeof(eventmask_t) * 8) EventSources.

In Use

The general usage of events is as follows:

EventListener Actions

- Associate an EventListener to an EventSource
- Wait for one/all/any EventSource to signal the thread
- Determine the active EventSource, extract flags and perform action.

EventSource Actions

- Signal a specific thread or broadcast to all threads. Supply additional flags if desired.

EventListener Actions

Association

Before a thread can listen for any events an EventListener must first be registered with the EventSource it is interested in. This is accomplished by calling: chEvtRegister(). See below for the header definition/documentation:

chEvtRegister

```
/**
 * @brief   Registers an Event Listener on an Event Source.
 * @note    Multiple Event Listeners can use the same event identifier, the
 *          listener will share the callback function.
 *
 * @param[in] esp       pointer to the  @p EventSource structure
 * @param[out] elp      pointer to the @p EventListener structure
 * @param[in] eid       numeric identifier assigned to the Event Listener. The
 *                      identifier is used as index for the event callback
 *                      function.
 *                      The value must range between zero and the size, in bit,
 *                      of the @p eventid_t type minus one.
 *
 * @api
 */
chEvtRegister(esp, elp, eid)
```

The first thing to discuss is the @note that states that multiple EventListeners can use the same Event Identifier (eid). When multiple threads are interested in the same EventSource, they are free to use the same ‘eid’. However, when a single thread is interested in multiple EventSources, a unique ‘eid’ must be used for each EventListener.

Secondly, the 'eid' parameter represents a THREAD-SPECIFIC value that associates an EventListener to an EventSource. Valid values are generally 0 - 31. When the EventSource signals, this value will be OR'd into a bit mask of all active events currently signaling the thread associated with the EventListener. Because this value is thread-specific, different threads can assign a different eid when listening to the same EventSource.

Waiting

There are a variety of waiting API's provided. Each of these API's will suspend the thread and allow waiting for either:

- A specific EventSource: **chEvtWaitOne**
- Any EventSouces specified by a mask of the eid's assigned to EventListeners: **chEvtWaitAny**
- All EventSources specified by a mask of the eid's assigned to EventListeners: **chEvtWaitAll**

All of these functions take an eventmask_t parameter. This parameter is an OR'ed set of the eid's of the EventListeners of the EventSources you are interested in waiting for. Generally, you should create this using the EVENT_MASK macro (example below):

- EVENT_MASK(eid1) | EVENT_MASK(eid2)

chEvtWaitOne

This is a simple macro for chEvtWaitOneTimeout. This function will wait for ANY of the events specified by the eventmask_t. It will return ONLY 1 of the active events at a time. Therefore, it must be called in a loop to service all the active events.

```
EVENTSOURCE_DECL(eventSource1);
EVENTSOURCE_DECL(eventSource2);


void
Thread (void *arg)
{
    EventListener es1Listener;
    EventListener es2Listener;
    eventmask_t   activeEvents;
    flagmask_t    flags;


    enum { es1ID = 7, es2ID = 3 };


    /*
     * Register the event listener's with the event source's.  The eid's can
     * be any unique number between 0 - 31.
     */
    chEvtRegister(&eventSource1, &es1Listener, es1ID);
    chEvtRegister(&eventSource2, &es2Listener, es2ID);

    while (1) {
        /* Wait for one of the events to signal */
        activeEvents = chEvtWaitOne(EVENT_MASK(es1ID) | EVENT_MASK(es2ID));

        if (activeEvents == EVENT_MASK(es1ID)) {
            /* eventSource1 has signalled.  See if it posted any flags. */
            flags = chEvtGetAndClearFlags(&es1Listener);

            /* Do something else */
        }
        else if (activeEvents == EVENT_MASK(es2ID)) {
            /* eventSource2 has signalled.  See if it posted any flags. */
            flags = chEvtGetAndClearFlags(&es2Listener);

            /* Do something else */

        }
    }
}
```

chEvtWaitAny

This is a simple macro for chEvtWaitAnyTimeout. This function will wait for ANY of the events specified by the eventmask_t. It will return ALL of the active events. Therefore, you must mask the returned value to determine which of the EventSources has been signaled.

```
EVENTSOURCE_DECL(eventSource1);
EVENTSOURCE_DECL(eventSource2);


void
Thread (void *arg)
{
    EventListener es1Listener;
    EventListener es2Listener;
    eventmask_t   activeEvents;
    flagmask_t    flags;


    enum { es1ID = 0, es2ID = 3 };


    /*
     * Register the event listener's with the event source's.  The eid's can
     * be any unique number between 0 - 31.
     */
    chEvtRegister(&eventSource1, &es1Listener, es1ID);
    chEvtRegister(&eventSource2, &es2Listener, es2ID);

    while (1) {
        /*
         * We can now wait for our events. We will wait for either of them to
         * signal.
         */
        activeEvents = chEvtWaitAny(EVENT_MASK(es1ID) | EVENT_MASK(es2ID));

        /*
         * At this point, 'activeEvents' holds a bitmask of all events which
         * have signaled.  Because we waited for either of them, we need to
         * test each independently to determine which (or both) has signaled.
         */

        /* See which (or both) events has signalled */
        if (activeEvents & EVENT_MASK(es1ID)) {
            /* The event has signalled.  See if it posted any flags. */
            flags = chEvtGetAndClearFlags(&es1Listener);
        }

        if (activeEvents & EVENT_MASK(es1ID)) {
            /* The event has signalled.  See if it posted any flags. */
            flags = chEvtGetAndClearFlags(&es2Listener);
        }
    }
}
```

chEvtWaitAall

This is a simple macro for chEvtWaitAllTimeout. This function will wait for ALL of the events specified by the eventmask_t. It will return once ALL of the EventSources have been signaled.

```
EVENTSOURCE_DECL(eventSource1);
EVENTSOURCE_DECL(eventSource2);


void
Thread (void *arg)
{
    EventListener es1Listener;
    EventListener es2Listener;
    eventmask_t   activeEvents;
    flagmask_t    flags;


    enum { es1ID = 4, es2ID = 3 };


    /*
     * Register the event listener's with the event source's.  The eid's can
     * be any unique number between 0 - 31.
     */

    chEvtRegister(&eventSource1, &es1Listener, es1ID);
    chEvtRegister(&eventSource2, &es2Listener, es2ID);

    while (1) {
        /* Now will wait for both of them to signal. */
        activeEvents = chEvtWaitAll(EVENT_MASK(es1ID) | EVENT_MASK(es2ID));
        assert(activeEvents == EVENT_MASK(es1ID) | EVENT_MASK(es2ID))

        /*
         * We know that both of them have signaled, get the flags.  In this
         * example we don't do anything with the flags, but you generally would
         * need to use a separate variable for each.
         */
        flags = chEvtGetAndClearFlags(&es1Listener);

        /* Do Something */

        flags = chEvtGetAndClearFlags(&es2Listener);

        /* Do Something */
    }
}
```

EventSource Actions

Signaling

There are a variety of API's provide that allow a specific EventListener's thread to be made ready or all EventListeners' threads associated with the EventSource be made ready. The API is as follows:

- Ready a specific thread with **chEvtSignal**
- Ready all threads with **chEvtBroadcastFlags**

chEvtSignal

This function provides a way to make ready a specific thread with a specific set of event flags. Moreover, the caller of this function must have specific knowledge of the thread it is making ready. Specifically, the caller must have a pointer to the thread it wishes to make ready, as well as the eventID's that the thread has enabled. It is meant to be a lightweight signaling mechanism generally used in conjunction with **chEvtAddEvents**.

```
#define THREADS_STACK_SIZE      (512u)

typedef struct _EventContext EventContext_t;
struct _EventContext
{
    Thread      *pThread;
    eventmask_t eventMask;
    void        (*threadFunc)(void);
};

static WORKING_AREA(thread2_wa, THREADS_STACK_SIZE);


void
TestFunc(void)
{
   /* Do something more useful, but as an example just sleep */
    chThdSleepMilliseconds(50);
}

void
Thread2(void *arg)
{
    EventContext_t *pContext = (EventContext_t *)arg;


    if (NULL == pContext) {
        return 0;
    }

    if (NULL != pContext->threadFunc) {
        pContext->threadFunc();
    }

    /* Tell the caller that we are done */
    chEvtSignal(pContext->pThread, pContext->eventMask);

    return 0;
}


void
Thread1(void *arg)
{
    EventContext_t context;
    enum { eMask = 1, eID = 0 };


    /* Configure our event context */
    context.pThread    = chThdSelf();
    context.eventMask  = eMask;
    context.threadFunc = TestFunc;

    /* Configure our event mask */
    chEvtAddEvents(eMask);

    /* Start the worker thread */
    chThdCreateStatic(thread2_wa, sizeof(thread2_wa),
                      chThdGetPriority() - 1, thread2, &context);

    /* Do More Work */

    /* Wait for the other thread to finish */
    chEvtWaitOne(EVENT_MASK(eID));

    return 0;
}
```

chEvtBroadcastFlags