THE EVENTS AND SERVICES FRAMEWORK

Courtesy of Prof. Carryer at Stanford University
Revised by Soja-Marie Morgens and Max Litchenstein
Second Generation, Documentation Rev 2.4 01/05/2013

BACKGROUND:
When writing event driven programs, there is a certain amount of code that is virtually identical in all applications. This code handles the creation, monitoring and manipulation of event queues, the regular execution of event checking routines, and the execution of service functions when there are events that need to be processed. The Events and Services Framework provides this code in a form that can easily incorporate your own code to quickly build an event driven application.

OVERVIEW OF THE FRAMEWORK:
In the terminology that we use here, an event driven application is made up of events and services. Events are changes in the state of the world, occurring either inside or outside of the device. Services are routines which respond to events. In a sense, the ES Framework’s purpose is simply to deliver events to services.

Events can represent external changes like “the front bumper just hit something” or “the IR detector circuit can no longer see the beacon.” They can also represent internal changes like “Timer 8 just expired” or “The reloader finished running.” Events only represent changes not states, so “The wall is 5 feet away” or “the front bumper is still hitting something” are not facts you can represent with events.

Events may originate in one of two ways. The ES framework periodically executes a list of event checkers, which are polling routines that check an external device and test for changes in its status. Alternatively, another service can generate an event. (Actually, there is a third way: a hardware interrupt can generate an event. However, you are unlikely to find this useful or use this in this class).

Services are routines which react to events. Each service is represented by a handful of functions. Services can be very simple or quite complex. The simplest service might be one that simply reports the events that arrive through its queue to a human user. This
kind of service is very useful both in testing the event checkers, and in building familiarity with the ES Framework. The most complex services are hierarchical state machines, such as the ones that govern the behavior of your robot. By default, the ES Framework runs two services: TimerService, which reports the setting and expiration of timers, and KeyboardInput, which manages keystrokes from a user. Developing other services to control your mechatronic system will occupy the majority of your time writing code in this course.

When run, the ES Framework proceeds in a fairly simple way, at least at the top level. First, each event checker is run. If an event checker detects an event, it puts that event into the event queue of the appropriate services. Then, if any services have events in their queues, the framework selects that service and runs it with the event as the input. The service may ignore the event, generate a new event, move a motor, switch to a different state, or some combination of the above. Once this is complete, the ES Framework deletes that event from the queue. Once the event queues are empty, the ES Framework’s loop is complete, and it again runs each event checker.

One of the most common ways in which frameworks are described is to say that they adhere to the Hollywood Principle: “Don’t call us, we’ll call you.” While not strictly true (your code will call some ES Framework functions), the description seeks to convey the idea that, at the highest level, your code does not call the functions that you write; rather the framework will call your functions when the time is right. Accepting this paradigm and trusting the framework to call your functions when it is necessary is fundamental to using the framework. This is not to say that you are totally giving up control. You will be writing the event checking routines that generate the events that will be responded to so you will be able to trigger a call to a service run function by generating an event. Also, you are responsible for writing the services’ response (or lack thereof) to an event. What you will not do is to call your service directly.

FEATURES OF THE FRAMEWORK:
- Up to 13 independent services each with its own event queue and a unique priority.
- Up to 250 user defined event types, each with a user defined 16-bit parameter.
- 16 independent 32 bit timers (based on a 1ms tick rate, which can run for up to 45 days) that generate ES_TIMEOUT, ES_TIMERACTIVE, and ES_TIMERSTOPPED events to your application.

OPERATION OF THE FRAMEWORK:
In an application, there are generally only two direct function calls to the framework, which correspond to the two phases of operation. (That is, your main() function will only contain two calls to the ES Framework). The first phase, initialization, is triggered by a call to ES_Initialize(). In this call, internal data structures and variables of the framework are initialized, the timer that is used to generate timeout events is initialized and an each services' initialization function is called. This service initialization function allows each service an opportunity to perform local initialization before the framework begins its main
The second phase of operation is the running of the framework, initiated by a call to ES_Run(). In the absence of errors, this call will never return. The call to ES_Run() enters into the main loop of the ES Framework, which checks event queues for events, passes those events to relevant service routines, checks for system-generated events, and then executes each event checker. This is shown in flowchart form below.

**OPERATION OF THE FRAMEWORK:**

When using the framework in your own code, you should only edit one file from the framework, which is ES_Configure.h. You will write hardware drivers, event checkers, and services, and then use ES_Configure.h to tie all the elements together.

In order to set up the ES Framework in MPLABX, you should use absolute includes to
include all ES_Framework files from the CMPE118 folder, with the exception of ES_Configure.h. Use relative includes to include files you need from your own code folder, such as drivers, event checkers, and services. Finally, make a copy of ES_Configure.h and put it in your MPLABX project folder, then use a relative include to include that copy of ES_Configure. Each project should have its own copy of ES_Configure.

It is recommended that you create a separate MPLABX project for each element of your system. For example, you should have one project for the bumpers, one project for the wheel motors, one project for a wall-following state machine, one project for your top-level state machine, etc. MPLABX has a utility to efficiently duplicate projects. Use relative includes to include files from other projects, rather than copying them, so that fixes propagate throughout all of your code.

**DESIGNING AN APPLICATION WITH THE ES FRAMEWORK**

Writing the code for a complicated system can be quite daunting, even with a pre-made framework. Fortunately, the process can be broken into simpler steps. The following strategy is recommended:

1. Start by listing the events that will be of useful for your system, and the responses that you would like the program to perform. For example, “Hit a wall” is likely to be of interest to a mobile robot, and “stop wheel motors” is a sensible response. Feel free to create your own special purpose events and don’t forget that each event has an associated 16-bit parameter that you can use to carry information about the event.

Remember that events are not only generated by the microcontroller hardware or by sensors. A service can also generate events either for themselves or for other services. When you need to communicate with another service, you can post an event to that service’s event queue by calling its public post function.

2. Write the event checkers that your application will require. See the sections below for more information about how to implement these.

3. Write the service functions that will handle the response to the events that get deposited into the service’s queue. For most functions of your application, the service will likely take the form of a state machine. However, you may also want to use simpler, non-state machine services, such as a controller for a blinking LED. There are eight possible services and there is very little overhead involved in using each additional service up to the limit, so you shouldn’t be shy about using them.

4. The final step in creating the application (actually getting it to the point of compiling) is to edit the ES_Configure.h file to tell the framework where to find your event checkers, service routines, timer associations and other bits. Later in this document you will find a tour through the ES_Configure.h file with instructions for how to edit it to customize it for
your application.

**EVENT CHECKING FUNCTIONS:**

The fundamental behavior of an event checking function is to detect when a change has happened that should trigger the generation of an event. An event represents the point of transition in a bit state or variable value. Therefore, almost every event checking function will require a static variable to hold the last tested value of the bit or variable. While it is possible to use a static local variable for the last state, doing so will almost invariably generate a spurious event the first time it is tested. To avoid the generation of a spurious event on startup, you can make the “last state” variable a module level variable and provide an initialization function to perform an initial read of the tested port/register/variable and assign that value to the “last state” variable. When this is done, the first call to the event checking function will have valid values for both the “last state” and the “current state”. The last act that every event checking function must perform is to update the value of the “last state” variable with its current state/value. The return value from an event checking function should be TRUE, if a new event was detected and FALSE otherwise.

**Example Event Checker Function:**

In this example, the event checker uses a module-level variable (lastBumpState) to keep track of the previous readings from the bumpers. Each time it runs, it compares the current readings with the previous readings, and posts an event if and only if the readings have changed. Notice that the post function is PostRoachFSM and that the event light to dark vs. dark to light is encoded within the parameter. A final note is that the parameter must be cast to a uint16_t. You're free to use the parameter to encode any information you like in any way you like -- here we use it to record the readings from all the bumpers.

```c
uint8_t CheckBumps(void)
{
    static int lastBumpState = 0;
    int currentBumpState;
    ES_Event thisEvent;

    uint8_t returnVal = FALSE;
    // check the light level and assign LIGHT or DARK
    currentBumpState = Roach_ReadBumpers();
    if (currentBumpState != lastBumpState) { //event detected
        thisEvent.EventType = BUMPED;
        thisEvent.EventParam = (uint16_t) currentBumpState;
        PostRoachFSM(thisEvent);
        returnVal = TRUE;
        lastBumpState = currentBumpState;
    }

    return returnVal;
}
```

**SERVICE FUNCTIONS:**

Services are more complex than event checking functions. A service actually consists of
at least 3 public functions:

• An *initialization* function that will be called during the framework initialization.
• A *post* function that can be called by other services to place an event into that service’s queue.
• A *run* function that will be called by the framework whenever it detects that there is an event ready in a service’s queue. This is the code that responds to events.

When the framework detects a new event, it will remove that event from the queue and pass it to the service run function associated with that queue. Each service is assigned a unique priority number that for this implementation lies between 0 and 7 with 0 being the lowest priority. When events are ready in more than one event queue, the highest priority service is processed first. Note that priority 0 is already in use for the User Timer functions.

**The Service Initialization Function:**

To facilitate the easy rearrangement of priorities in the configuration file, the service initialization function is passed a parameter to indicate the priority assigned to the service. The basic operation that every service initialization function must do is to save that priority into a module level variable. This value will be needed by the post function to post to the correct queue.

Beyond the saving of the priority, the work done by the initialization function will vary by application. Operations might include calling the function to initialize the “last value” variable in any associated event checking function(s), initializing a state variable, posting an ES_INIT event to itself to trigger an initial transition in a state machine that is implemented by the service or any other initialization that is needed by the service. The initialization function should return TRUE if no errors were detected during the initialization and FALSE if any error was detected.

**Example Initialization Function:**

```c
uint8_t InitTimerService(uint8_t Priority) {
  ES_Event ThisEvent;
  MyPriority = Priority;
  // in here you write your initialization code
  // post the initial transition event
  ThisEvent.EventType = ES_INIT;
  if (ES_PostToService(MyPriority, ThisEvent) == TRUE) {
    return TRUE;
  } else {
    return FALSE;
  }
}
```
The Service Post Function:
The primary purpose of the post function is to provide an interface to event checking functions and other services to allow them to post events to this particular service without needing to know what priority level has been assigned to this service. At this time, I cannot think of a reason that you would need to modify (other than to re-name) the post function provided in the service and state machine templates provided.

The Service Run Function:
The run function is the core of a service. It will be called by the framework whenever an event is found in its event queue. The framework will remove the event from the queue and pass it as a parameter to the run function. The major services of an application will most commonly implement state machines and state charts. Support services might be used, for example, to simulate external events during debugging, report the generation events or periodically perform some operation triggered by a timer or other event.

The run function return value is an event. In the case of a major error that should cause the ES_Run() function to terminate, the service run function should return an ES_ERROR event. In the absence of serious errors, the service run function should return an ES_NO_EVENT event.

Example Run Function:
This service simply updates an array called UserTimerStates whenever the status of a timer changes.

```
ES_Event RunTimerService(ES_Event thisEvent) {
    ES_Event returnEvent;
    returnEvent.EventType = ES_NO_EVENT; // assume no errors
    switch (thisEvent.EventType) {
        case ES_INIT: // do nothing
            break;
        case ES_TIMEOUT:
        case ES_TIMERACTIVE:
        case ES_TIMERSTOPPED:
            UserTimerStates[thisEvent.EventParam] = thisEvent.EventType;
            break;
        default:
            returnEvent.EventType = ES_ERROR;
            break;
    }
    return returnEvent;
}
```

OTHER FRAMEWORK FUNCTIONS:
In addition to the work on initialization and running of the main event loop, the framework provides a number of other functional blocks that are available to you in designing and
writing your service code:

• Timers. This version of the framework provides for up to sixteen (numbered 0-15) 32-bit timers that all tick at a 1ms rate. This allows events from 1ms to 45 days to be triggered. The timers are segregated into system timers (0-1) and 14 user timers (2-15). For the lower 8 user timers, functions are provided to test if the timer is active, stopped, or expired. For timers 2-15, you select a post function to be called when a timer expires via editing ES_Configure.h.

If you wish for a particular timer to post to multiple services, you can use a distribution list and put the associated ES_PostListxx function name as the posting function for that timer.

• FreeRunningTimer. A free running 32 bit timer (also at 1ms) is available to your application. ES_Timer_GetTime() returns the current 32bit free running counter value.

• Event Queues. While the main event queues of the framework are created, monitored and manipulated by the framework, the underlying queue library is available for your application to use to create its own private event queues.

• Distribution lists. There may be times when you would like to post an event to more than one service’s queue. While this could be accomplished by using multiple calls to the individual post functions, the framework provides a cleaner, easier way in the form of distribution list functions.

In its current incarnation, the framework provides for up to 8 unique distribution lists. Distribution lists are accessed through a set of post functions, ES_PostList00(), ES_PostList01(),... ES_PostList07(). These post functions can be used in place of any single post function and will result in the event being posted to each of the service queues in the distribution list. The contents of each of the distribution lists is configured through the ES_Configure.h file.

EDITING ES_CONFIGURE.H:
The ES_Configure.h file is the only framework file that you will need to edit to adapt the framework to your application. The file consists of a number of #define definitions that are then used in modules throughout the framework. The ES_Configure.h file should be the first #include file in each of your modules.

The process of editing ES_Configure.h consists entirely of changing the definitions of compiler macros. The relevant compiler macros are listed below:
<table>
<thead>
<tr>
<th>MAX_NUM_SERVICES</th>
<th>This is the maximum number of services allowed. At this time, the only valid value for this definition is 8, so leave it alone.</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM_SERVICES</td>
<td>This macro determines that number of services that are actually used in a particular application. It will vary in value from 1 to MAX_NUM_SERVICES.</td>
</tr>
<tr>
<td>SERV_1_HEADER</td>
<td>These are the definitions for Service 1, the lowest priority service available (Service 0 is the user timer functions. Further services are added in numeric sequence 1,2,3,... with increasing priorities. You may not skip numbers in the sequence.)</td>
</tr>
<tr>
<td>SERV_1_INIT</td>
<td>SERV_1_HEADER is the name of the header file for the service 1 module. It should be entered with leading and trailing double quotes, just as you would in an #include statement.</td>
</tr>
<tr>
<td>SERV_1_RUN</td>
<td>SERV_1_INIT and SERV_1_RUN are the names of the initialization routine and run routine for the service. These names should be entered without any quotes.</td>
</tr>
<tr>
<td>SERV_1_QUEUE_SIZE</td>
<td>SERV_1_QUEUE_SIZE is a number to indicate the size of the event queue to be reserved for this service. The minimum size is 1. There is no predetermined maximum size. However, the need for a queue of greater than 3 should be viewed with skepticism. If there is an interrupt response (e.g. timer expiration) that can post to this event queue then add 1 entry per possible interrupt generated event. If another higher priority service will post events to this service, add 1 entry per other service. An event queue size of 3 would allow for an interrupt to post an event, another service to post an event and this service to post an event to itself all happening during the same time.</td>
</tr>
</tbody>
</table>
These are the definitions for services 2 through 7. They follow the same pattern as for service 1.

typedef enum {
    ES_NO_EVENT = 0,
    ES_ERROR,
    ES_INIT,
    ES_TIMEOUT,
    ES_TIMERACTIVE,
    ES_TIMERSTOPPED,
    /* User-defined events here */
} ES_EventType_t

Here is where you define the symbolic names for the events used in your application. The first few entries are reserved for system-defined events. Beyond that, you are free to add your own. There is a limit of 256 on the total number of events (system + user-defined).

This entry defines the number of distribution lists that will be created. It ranges from 0 to 8.

These are the definitions for the distribution lists. Each definition should be a comma-separated list of post functions that will be executed when that particular distribution list is invoked.

This is the name of the Event checking function header file. If your event checkers are not all in one file, then you will need to create a “wrapper” header file that includes the
<table>
<thead>
<tr>
<th></th>
<th>actual header files for all of the event checking modules. You should place the name of that “wrapper” header (with leading and trailing double-quotes) in this definition.</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVENT_CHECK_LIST</td>
<td>This definition is a comma separated list of the event checking functions. There should be no quotes, nor leading or trailing commas in this definition. The event checking functions will be executed in the order given in this list so placing an event checker earlier in the list will give it a higher priority. When an event checking function indicates that an event was found, the processing of the list breaks and the new event is processed from the queue into which it was deposited. The next time checking for user events begins, it will begin again from the start of the list.</td>
</tr>
<tr>
<td>TIMER8_RESP_FUNC TIMER9_RESP_FUNC TIMER10_RESP_FUNC TIMER11_RESP_FUNC TIMER12_RESP_FUNC TIMER13_RESP_FUNC TIMER14_RESP_FUNC TIMER15_RESP_FUNC</td>
<td>These are the names of the post functions that will be called when each of the timers expires. You may direct any number of timers to post their events to the same service. The service can differentiate between the ES_TIMEOUT events by examining the event parameter. The parameter will contain the number of the timer that generated that particular ES_TIMEOUT event. All of these definitions must exist. If you are not using a particular timer, you can list its definition as TIMER_UNUSED. If you wish for a timer to post to multiple services you can use a distribution list and put the associated ES_PostListxx function name here.</td>
</tr>
</tbody>
</table>