UNIVERSITY OF CALIFORNIA, SANTA CRUZ
BOARD OF STUDIES IN COMPUTER ENGINEERING

CMPE118(218)/L: INTRODUCTION TO MECHATRONICS

THE EVENTS AND SERVICES FRAMEWORK[†]

PURPOSE:

The primary purpose of the Events and Services Framework is to support the creation of efficient event driven programs with a minimum of effort spent on boiler-plate code that is necessary to all event driven applications.

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 (polling) routines to detect non-interrupt generated events and the execution of *service* functions when there are events that need to be processed. The Events and Services Framework provides all of these functions and a straightforward way for you to incorporate your code into the framework 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* (things that need to be responded to) and *services* (the responses to the events). Events are detectable changes in 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 may originate with polling routines that test for changes external to the microcontroller, with interrupt response routines triggered from within the microcontroller, or from other parts of the application (the services). Services have a minimum defined set of functions to provide a standard, flexible interface between your code and the framework. On the simple end are services that simply report the occurrence of the events that arrive through its queue. This kind of service is very useful both in testing the event detection functions and debugging at later stages of development. The most complex service functions may be those that implement orthogonal regions and include multiple hierarchical statecharts. The simpler flat state machine implementations fall somewhere in between.

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.

[†] Courtesy of Prof. Carryer at Stanford University

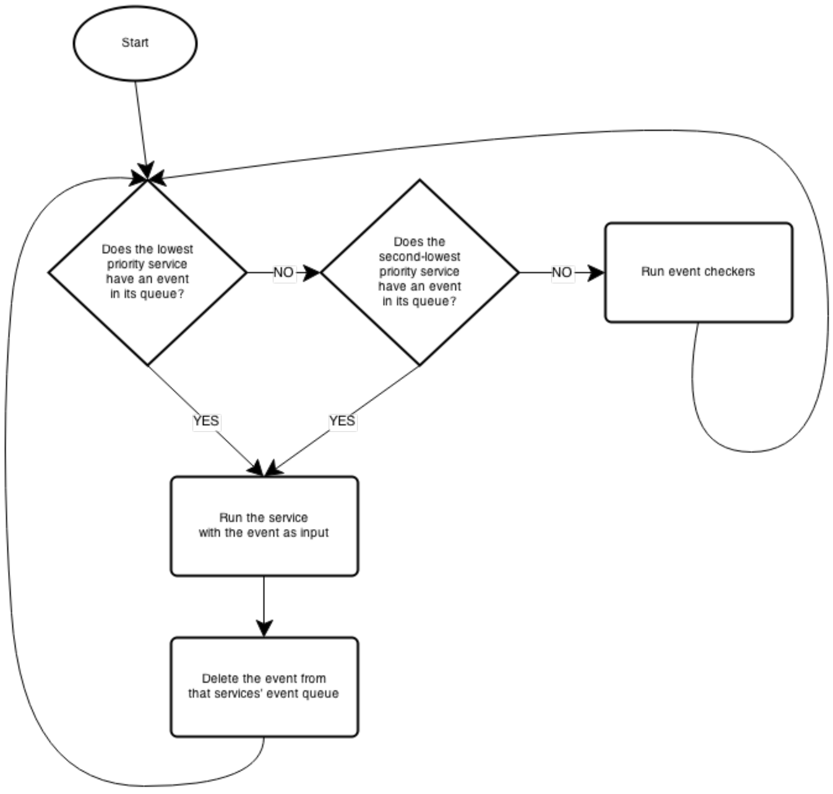
Start
Does the lowest priority service have an event in its queue?
NO
Does the second-lowest priority service have an event in its queue?
NO
Run event checkers
YES
YES
Run the service with the event as input
Delete the event from that services' event queue

Figure 1: ES_Framework Flow Chart

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 (you will call the 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. Thus you will be able to trigger a call to a service run function by generating an event. What you will not do is to call your service run function directly.

FEATURES OF THE FRAMEWORK:

- Up to 8 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 32bit timers (based on a 1ms tick rate, up to 45days) that generate ES_TIMEOUT, ES_TIMERACTIVE, and ES_TIMERSTOPPED events to your application.

OPERATION OF THE FRAMEWORK:

The framework provides for up to 8 independent services each with its own event queue and a unique priority. The framework supports up to 250 user defined event types and provides 16 independent timers (based on a common 1ms tick rate) that are used to generate ES_TIMEOUT events to your application. In addition there is a 32bit free running counter. In an application, there are generally only two direct function calls to the framework and these correspond to the two phases of operation. The first phase, initialization, is triggered by a call to `ES_Initialize()`. During initialization the internal data structures and variables of the framework are initialized, the timer that is used to generate timeout events is initialized and an initialization function associated with each service is called. This service initialization function allows each service an opportunity to perform local initialization before the framework begins its main loop.

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 a loop that takes care of testing the event queues for events that are ready to be processed, when detected, passing those events to the relevant service routines, testing for system generated events (such as timers expiring) and calling your event checking functions to detect the occurrence of external events. In flowchart form, the basic `ES_Run()` operation is depicted in Figure 1.

USING THE FRAMEWORK:

The work of building an application using the Events and Services Framework falls into a few general categories that also fit nicely as the steps to writing the application:

1. Design the application as a set of events of interest and the responses that you would like the program to perform when these events are detected. In this step it is important to keep in mind that events are not only generated by the microcontroller hardware or things external to the microcontroller. Your service code can also generate events either for itself (the same service) or for other services. When you need to communicate with another service (or possibly with a state machine that is being run as a service) you should post an event to that service's event queue by calling its public post function. 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 modify the way that the event is processed in the receiving service run function.

2. Write the event checkers that your application will require. These routines should detect when an event has occurred (**note**: events represent transition points, detectable changes in a variable or externally, not a persistent state) and then post an event to one or more service's queues. Posting to a single service's queue is done simply by calling the public post function for that service. Posting the same event to more than one service queue is done via distribution lists (see the section on *Distribution Lists* below).

3. Write the service functions that will respond to the events that get deposited into the service's queue. For the major functions of your application the service run function will likely implement a state machine or state chart. It will also be likely that you will want to use simpler, non-state machine services. There are eight possible services and there is very little overhead involved in using each additional service up to the limit. Services that are simply waiting for events to appear in their queues take no execution time until an event is deposited by calling the post function 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 about your event checkers, service routines, timer associations and other bits. The `ES_Configure.h` file is included in each of the modules of the framework and it is the only file in the framework that you should need to edit to make a new application. A tour through the `ES_Configure.h` file with instructions for how to edit it to customize it for your application appears later in this document.

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, 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 `lastLightState` variable is a module level variable defined outside the scope of the event checking function. Note the use of a `static enum` type for the `lastLightState` variable, as well as the `#defines` for the constants that define light levels. This is important for code maintainability (do not put magic numbers in your code). Notice that the post function is to `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 from an `enum` type to a `uint16_t`.

```
#define DARK_THRESHOLD 234
#define LIGHT_THRESHOLD 88
static enum {DARK, LIGHT} lastLightState;


uint8_t CheckLightLevel(void) {

    enum {DARK, LIGHT} currentLightState;
    unsigned int currentLightValue;
    uint8_t returnVal = FALSE;
    // check the light level and assign LIGHT or DARK
    currentLightValue = LightLevel();
    if (currentLightValue > DARK_THRESHOLD) {
        currentLightState = DARK;
    }
    if (currentLightValue < LIGHT_THRESHOLD) {
        currentLightState = LIGHT;
    }
    if (currentLightState != lastLightState) { //event detected
        ES_Event ThisEvent;
        ThisEvent.EventType = LIGHTLEVEL;
        ThisEvent.EventParam = (uint16_t) currentLightState;
        PostRoachFSM(ThisEvent);
        returnVal = TRUE;
    }
    lastLightState = currentLightState;
    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, before the framework begins its run loop.
- 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. 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:
```
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.

Example Post Function:
```
uint8_t PostTimerService(ES_Event ThisEvent) {
    return ES_PostToService(MyPriority, ThisEvent);
}
```

The Service Run Function:
As you might expect, 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:
```
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 synchronized 1ms rate. This allows events from 1ms to 45days to be triggered. When a timer expires, a posting function will be called to post an ES_TIMEOUT event with the event parameter set to the number of the timer. Which post function will be called when a timer expires is configured 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 will be times when you would like to post an event to more than one service's queue. While in your event checkers 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 that can be used anywhere a single post function is required. 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 are configured through the `ES_Configure.h` file.

EDITTING 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.

`MAX_NUM_SERVICES`	This is the maximum number of serviced allowed. At this time, the only valid value for this definition is 8, so leave it alone.
`NUM_SERVICES`	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

`SERV_1_HEADER`
`SERV_1_INIT`
`SERV_1_RUN`
`SERV_1_QUEUE_SIZE`
	These are the definitions for Service 1, the lowest priority service available (Service 0 is the user timer functions). Every Events and Services application must have a Service 0. Further services are added in numeric sequence (1,2,3,...) with increasing priorities. You may not skip numbers in the sequence.
	
	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 a `#include` statement.
	
	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. 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.

`SERV_2_HEADER`
`SERV_2_INIT`
`SERV_2_RUN`
`SERV_2_QUEUE_SIZE`
.
.
.
`SERV_7_HEADER`
`SERV_7_INIT`
`SERV_7_RUN`
`SERV_7_QUEUE_SIZE`
	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 start here */`
`  LIGHTLEVEL,`
`} ES_EventTyp_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).

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

`DIST_LIST0`
`DIST_LIST1`
`DIST_LIST2`
`DIST_LIST3`
`DIST_LIST4`
`DIST_LIST5`
`DIST_LIST6`
`DIST_LIST7`
	These are the definitions for the actual distribution lists. Each definition should be a comma separated list of post functions that will be executed when that particular distribution list is invoked.

`EVENT_CHECK_HEADER`	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 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.
`EVENT_CHECK_LIST`	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.

`TIMER8_RESP_FUNC`
`TIMER9_RESP_FUNC`
`TIMER10_RESP_FUNC`
`TIMER11_RESP_FUNC`
`TIMER12_RESP_FUNC`
`TIMER13_RESP_FUNC`
`TIMER14_RESP_FUNC`
`TIMER15_RESP_FUNC`
	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.