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

CMPE118(218)/L: INTRODUCTION TO MECHATRONICS

DEBUGGING THE EVENTS AND SERVICES FRAMEWORK[†]

PURPOSE:

State machines, and particularly hierarchical state machines, can be challenging to debug. The essential task is to keep track of which event/state pairs are occurring in which specific state of the state/sub-state machine. It is relatively easy to get lost, and become confused simply by observing the behavior of the code. We have provided some methods to track state/event pairs in a way that makes it easier to see what is happening in your code.

While all standard debugging techniques work (printf, reading through code etc), there are two specific functions within ES_FRAMEWORK that will significantly aide in your state machine debugging: TattleTale and ES_KEYBOARD_INPUT. These will allow you to determine the state of your state machine while your code runs as well as allowing you to virtually debug the state machine. Use them well. **DO NOT TRY TO REMOVE THEM FROM THE CODE**. If you don't want to use them, comment out the flags in ES_Configure.h that turn them off. This document outlines how to set them up and how to use them.

SETUP:

Open ES_CONFIGURE.h. Some of the first few lines are:

```
//defines for keyboard input
//#define USE_KEYBOARD_INPUT
//What State machine are we testing
//#define POSTFUNCTION_FOR_KEYBOARD_INPUT PostGenericService

//define for TattleTale
#define USE_TATTLETALE

//uncomment to supress the entry and exit events
//#define SUPPRESS_EXIT_ENTRY_IN_TATTLE
```

TattleTale is turned on by default. If you wish to turn it off, comment out the the "#define USE_TATTLETALE" line to disable it. **DO NOT COMMENT OUT TATTLETALE CALLS IN ANY OTHER FILES**. They are known respectively as ES_TATTLE() and ES_TALE(); ES_TATTLE() is found right before any state machine switch statement and ES_TALE() is found right after. These functions are included in the templates (both main and substate machine) so make sure you don't comment them out or erase them.

KEYBOARD INPUT:

The Keyboard_Input mode changes the way events are generated. Normally, events are generated by the code in your event checking module. When Keyboard_Input is enabled, events are instead generated by keystrokes entered into the terminal by a human (you). This lets you check your state machines independently from the hardware. To enable it, uncomment the "#define KEYBOARD_INPUT" line.

Using the Keyboard_Input mode is a bit counter-intuitive. The terminal will present you with a list of events (as defined in your ES_configure.h file). You then enter a number into the terminal input field followed by a semicolon and press enter. The ES framework will interpret this as an event and react accordingly (i.e.: the service designated by the `POSTFUNCTION_FOR_KEYBOARD_INPUT` function will get that event), then will wait for you to feed it the next event. You may wish to pass an event with a parameter. To trigger event "7" with a parameter of "00000111", type "B->7;" and press enter. The semicolon is required.

[†] Courtesy of Prof. Carryer of Stanford University

TATTLETALE:

TattleTale allows you to see not only the state(s) you are in but the events that trigger the transitions. This is extremely useful in debugging a running robot. In conjunction with ES_KEYBOARD_INPUT, this allows you to virtually run and debug your state machine before you have any working hardware (the software is always to more difficult to debug, so getting started on it early is helpful).

TattleTale will string together your state/event transitions using the form of output to the serial port as a verbose message: `state_machine_file[State(Event,EventParameter)]->`

When tattletale is enabled, it will print out the initial trace (for example a hierarchal roach):

```
RunFancyRoachHSM[InitPState(ES_INIT,0)]->RunSubStoppedHSM[InitPSubState(ES_INIT,0)]->
RunSubStoppedHSM[InitPSubState(ES_EXIT,0)]->RunSubStoppedHSM[Halted(ES_ENTRY,0)]->
RunFancyRoachHSM[InitPState(ES_EXIT,0)]->RunFancyRoachHSM[Stopped(ES_ENTRY,0)];
```

The first line, `RunFancyRoachHSM[InitPState(ES_INIT,0)]` translates as: the ES_Init event called the highest state machine (RunFancyRoachHSM) which starts in its initial state, InitPState. From its initial state, it activates the first sub state machine (RunSubStoppedHSM), passing an ES_Init event to the sub state machine.
The second line shows RunSubStoppedHSM changing its CurrentState from InitPSubState to Halted and exits the InitPSubState using an ES_EXIT (from InitPSubState) and ES_ENTRY to get into its new state, Halted.
The third line shows RunFancyRoachHSM changing its current state from InitPState to Stopped and exits the InitPSubState using an ES_EXIT and ES_ENTRY to get into Stopped.

While you might expect the third line to appear first (logically) sub-state machines actually transition *before* their supers. Again, sub-state machines get called before their supers. This has to occur so that events get consumed by the lowest level state machine that will handle it; this allows for more generic events to be handled at a higher level and more specific events handled at the lowest level.

Tattletale will output a trace at the end of every state machine run. For example, if you have an event (e.g.: INTO_LIGHT) while in the sub state "Halted", the output would look like this:

```
RunFancyRoachHSM[Stopped(INTO_LIGHT,8)]->RunSubStoppedHSM[Halted(INTO_LIGHT,8)]
        ->RunFancyRoachHSM[Stopped(ES_EXIT,0)]->RunSubStoppedHSM[Halted(ES_EXIT,0)]
        ->RunSubStoppedHSM[InitPSubState(ES_INIT,0)]->RunSubStoppedHSM[InitPSubState(ES_EXIT,0)]
        ->RunSubStoppedHSM[Halted(ES_ENTRY,0)]->RunFancyRoachHSM[Fleeing(ES_ENTRY,0)]
        ->RunSubRunHSM[Waiting(ES_ENTRY,0)];
```

The framework has passed the event down to the RunSubStoppedHSM, which has generated an exit event. This, in turn, resets the sub state machine (optional) and transitions RunFancyRoachHSM to its next state, Fleeing. Fleeing in turn initiates its first substate machine RunSubRunHSM which transitions to its first state, Waiting.

While ES_ENTRY and ES_EXIT events lead to these traces being somewhat verbose they will help you quickly find errors in your state machines transitions and events. You can turn them off by uncommenting #define SUPPRESS_EXIT_ENTRY_IN_TATTLE in ES_CONFIGURE.h

Tattletale operates under the assumption that your events and state names are declared with a certain consistency. This allows it to 'tattle' on the current states and events as they happen in the code. To ensure that the tattletale output is readable in human form (as opposed to simply state/event pairs as integers), we must translate the enum types to string arrays. This is done with a python script that runs before compilation.

EVENT AND STATE SETUP:

The events and states are set up using enumeration types standard within C, but must conform to some specific naming conventions in order for the python script to generate the string array that tattletale requires to print out the verbose names of state/event pairs. Previously we have used a custom script that required you to paste in the string array, and also some pre-processor hacks to generate the string array, however both methods had problems. Students forgot to run the script and would have their state/event names go out of sync with what was in the file (making tattletale more confusing), and the pre-processor tricks were very fragile and easy to break (throwing very hard to decipher errors in the compiler).

While we believe the new method very much superior, we are sure you will break it. Report any misbehavior to the teaching staff via piazza, and we will push updated versions as fast as we can fix what you break. The new python script will run automatically when you compile (you will set this in your project). It will search through the project files and create the string array and paste it in below your enum declaration. If something fails, it will break before the code can compile.

Events:
Events are available for all state and substate machines. All events are declared in the ES_CONFIGURE.h file. You will find them listed as (for example):

```
typedef enum {
    ES_NO_EVENT, ES_ERROR, /* used to indicate an error from the service */
    ES_INIT, /* used to transition from initial pseudo-state */
    ES_ENTRY, /* used to enter a state*/
    ES_EXIT, /* used to exit a state*/
    ES_KEYINPUT, /* used to signify a key has been pressed*/
    ES_LISTEVENTS, /* used to list events in keyboard input, does not get posted to fsm*/
    ES_TIMEOUT, /* signals that the timer has expired */
    ES_TIMERACTIVE, /* signals that a timer has become active */
    ES_TIMERSTOPPED, /* signals that a timer has stopped*/
    /* User-defined events start here */
    BATTERY_CONNECTED,
    BATTERY_DISCONNECTED,
    NUMBEROFEVENTS,
} ES_EventTyp_t;
```

Each user event needs to have a unique name, and should be in the comma separated list following the ES_TIMERSTOPPED event. There is no requirement to have them on their own line, as long as they are separated by commas. **DO NOT ALTER EVENTS ABOVE** /* User-defined events start here */ comment. While comments after the event name are not required, they are useful to include for code readability.

After the python script runs, there will be a string array inserted into the file below the typedef enum that has the name of every event as a string, one to a line:

```
static const char *EventNames[] = {
       "ES_NO_EVENT",
       "ES_ERROR",
       "ES_INIT",
       "ES_ENTRY",
       "ES_EXIT",
       "ES_KEYINPUT",
       "ES_LISTEVENTS",
       "ES_TIMEOUT",
       "ES_TIMERACTIVE",
       "ES_TIMERSTOPPED",
       "BATTERY_CONNECTED",
       "BATTERY_DISCONNECTED",
       "NUMBEROFEVENTS",
};
```

This will be regenerated each time you compile, and does not need to be deleted or altered. However, it is very important that you reformat the code using ALT-SHIFT-F before you compile. This ensures the python script will run without errors.

States:
State names are declared within each state machine (or sub-state machine) within the .c file. This means that the names do not have to be unique across all files in the project. See the template files to see where to put them in which type of state machine file. Again they are an enumerated type and once again they are typedef'd. In order for the python script to catch them, the typdef must end in "State_t" or it will break. A typical one will look like:

```
typedef enum {
    InitPSubState,
    Waiting,
    SpinningLeft,
    SpinningRight,
} SubDanceState_t;
```

Again, once the python script runs, it will insert a state array (local to the file) below the enumeration. Once again, make sure you auto-format using ALT-SHIFT-F before compiling (it is a good idea to do it all the time). The generated string array will look like:

```
static const char *StateNames[] = {
       "InitPSubState",
       "Waiting",
       "SpinningLeft",
       "SpinningRight",
};
```

The enums get used by your state machines, and the strings get used by tattletale. Again, put these in your .c file. Remember that the typdef enum needs to end in State_t or the python code won't find it. Finally, uncomment #define USE_TATTLETALE inside of ES_CONFIGURE.h if you want to turn it off.

ES_KEYBOARD_INPUT:

ES_KEYBOARD_INPUT allows you to artificially post events and parameters to your bot. It can be used in conjunction with TattleTale or alone. This is an excellent way to walk through your state machines and ensure it works the way you think it does.

Follow the instructions in TattleTale to setup the events correctly. Uncomment #define USE_KEYBOARD_INPUT inside of ES_CONFIGURE.h. Also make sure to designate the correct post function in the #define POSTFUNCTION_FOR_KEYBOARD_INPUT line. This tells the Keyboard Input where to send events.

When you initialize your program with activated KEYBOARD_INPUT, you should see output that look like this:

```
Printing all events available in the system

 0: ES_NO_EVENT        1: ES_ERROR        2: ES_INIT             3: ES_ENTRY
 4: ES_EXIT            5: ES_KEYINPUT     6: ES_LISTEVENTS       7: ES_TIMEOUT
 8: ES_TIMERACTIVE     9: ES_TIMERSTOPPED 10: INTO_LIGHT         11: INTO_DARK
12: BUMPED            13: DONE_EVADING
```

When keyboard input is active, no other events except timer activations will be processed. You can redisplay the event list by sending a 6 event.

This gives you a list of all of the events available to inject into the system. Note that while operating with keyboard input, the framework will not process any other events from the framework. To create an event, you need to type (for example) "10->0;" into the input and press enter. This creates a INTO_LIGHT event and the hex parameter (in this case 0). If you wanted to see what happens when timer 8 stops, you would type "9->8;"

Keyboard input gives you a complete method to test out your state machines before you test them on your robots. Use it to debug your state machine and ensure that you have coded exactly what you have drawn on your state machine diagram. This will save you untold hours in the lab.