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

CMPE118(218)/L: INTRODUCTION TO MECHATRONICS

CREATING A PROJECT USING THE EVENTS AND SERVICES FRAMEWORK†

PURPOSE:

The ES_Framework is simplified software structure that will allow you to check for events and dispatch them to appropriate services (simple, finite state machines, or hierarchical state machines). The underlying ideas come from Miro Samek's "Practical Statecharts in C/C++," and the implementation can be a bit daunting. This document runs through how to set up a the ES_Framework in a project such that you can run and test your code to ensure proper functionality. It is recommended you read the EventsServicesFramework document, and the Debugging_ES_FRAMEWORK document as well.

SETUP:

Begin by creating a new project in MPLAB-X as you would from the NewMPLABXProjectInstructions document. In addition to the usual header and source files you need, you will also need all of the header and source files associates with the ES_Framework (these all start with ES_).

In addition to the usual includes (both source and header files) there are a few additional steps that create an ES_Framework project: (1) You must have a valid ES_Configure.h file associated with your project, (2) You will need a valid ES_Run.c file associated with your project, and (3) You will need to set up the enum_to_string.py program to run before you build your project. Each of these steps will be detailed below.

ES_Configure.h is where you define your events, point the ES_Framework at your event checker routines, and associate your services (simple services, FSM, and HSMs) with their respective event queues. It is also where both the use of keyboard input (to generate events to send to the framework) and tattletale (to generate a trace of the event/state pairs) are enabled. These functions are detailed in the debugging the ES_Framework document. In general, you will need an ES_Configure.h for each of your projects (kept in the local directory), though sometimes it is preferable to work with only one master one.

ES_Run is the main function for the ES_Framework. You will need a local copy in order to perform any hardware initialization that you need specifically for your own hardware. For instance, if running on a roach, you would need a call to Roach_Init() to initialize the hardware.

Lastly, in order to run the TattleTale utility (which is very helpful in debugging state machines) there needs to exist a string version of the enumerations for both events and states. This is done using a python script that is set to run before build in MPLAB-X. This is set under the project properties -> Conf -> Building and checking the "Execute this line before build" box, and inserting: python C:\CMPE118\scripts\Enum_To_String.py

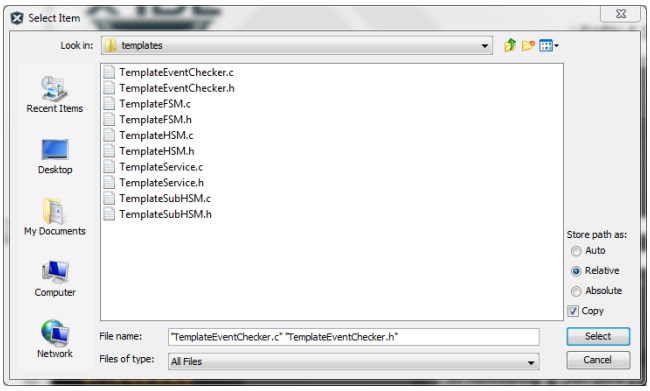

Figure 1: Adding the TemplateEventChecker files

Methods for testing each of the standard types of ES_Framework projects using the templates are detailed in the next sections. These are event checkers, simple services (which are most often

† Courtesy of Prof. Carryer of Stanford University

event checkers that run periodically), FSMs, and HSMs.

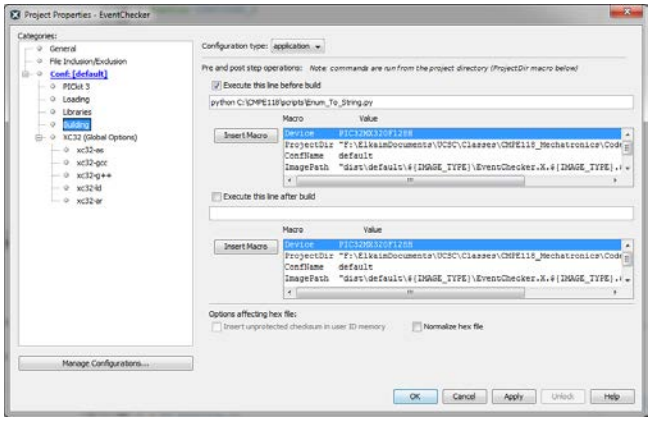

Figure 2: Setting Enum_To_String to run

EVENTCHEKING:

Event checking is a key part of the events and services framework. An event is a detectable change (not just some sensor being above a threshold). For example, the roach light sensor being below the LIGHT_THRESHOLD is not an event unless the roach was in the dark before. These can be a bit tricky to get right (especially with compound detections that need multiple things to happen for an event to be generated). In our experience, most of the problems students have had with the ES_Framework have been problems with event detectors, rather than problems with the state machines (though it is *very* important to match your state machine drawings to your code).

This section of the document will lead you through setting up an event checking project, correctly configure the ES_Configure.h file, and run the test harness. Unlike most of the other template files provided, the TemplateEventChecker.h/.c module will compile, run, and demonstrate a simple event (BATTERY_CONNECTED and BATTERY_DISCONNECTED) that you can trigger yourself on either the roach or the Uno32 stack hardware.

Begin by following the NewMPLABXProjectInstructions document to set up a new project in your local directory with the correct paths and includes from the C:\CMPE118 directory. The first thing you will do is to copy the TemplateEventChecker.c and TemplateEventChecker.h files to your local project. These can be found in the templates folder of the C:\CMPE118 directory. This can be done by right clicking on the "Source Files" icon in MPLAB-X and selecting "add existing item…" and using the "Copy" checkbox in the window (see *Figure 1*) This will create a copy of these files within your project folder. Note that you can add both the .h and the .c files at the same time, and simply drag the appropriate one to its correct folder within MPLAB-X.

The next step is to repeat the process with ES_Configure.h (also in the templates folder), and create a local copy of it in your project folder using the same method. In order to run the EventCheckers, you will also need AD.c/h, BOARD.c/h, ES_Events.h, and Serial.c/h in the project. These are linked in using absolute paths, not local copies.

In order to run the test harness, which will test the event checker code two additional steps are required. The first is to set the python script that will do the string conversions from the enum to automatically run. Go to the "Customize …" pulldown on the toolbar and select "Conf:default->Building" and check the box for "Execute this line before build" and type in: python C:\CMPE118\scripts\Enum_To_String.py (see *Figure 2*).

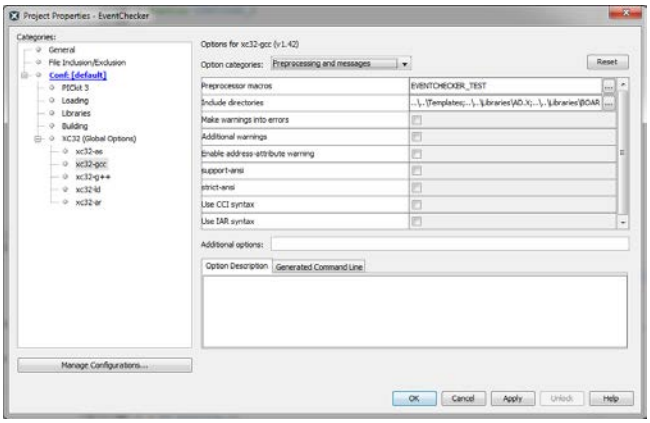

Figure 3: Macro EVENTCHECKER_TEST

The second step is to set a project macro to turn on the test harness. A project macro looks the same as a #define in the code, but it is not actually defined within the code itself. This allows you to conditionally compile parts of the code within one project, but link those files in another project and have different code conditionally compile. This is extremely powerful as it allows you to have your project with its own test harness but use the same code without the test harness in a different project. This means that you can always reload the first project and rerun your test harness when you discover you want to make changes to that code, but still use it within your other projects without a need to change the files once you have tested them.

In this case, we are going to add "EVENTCHECKER_TEST" to the project to enable the test harness. This is again done through the "Customize …" pulldown, and selecting "xc32 gcc." Select the "Preprocessing and messages" option category, and add "EVENTCHECKER_TEST" to the preprocessor macros (see *Figure 3*).

At this point, you should be able to build your project and have the code compile (note that you have not changed anything within any of the files, yet). Load your code onto a roach (or Uno32 stack) using the ds30Loader. Have a battery connected, but the power switch in the OFF position. Once you reset the roach, you should see on the serial port:

```
Event checking test harness for TemplateEventChecker.c
```

Now, when you switch the power switch to the ON position, and then to the OFF position, and back to the ON position you should get an event each time the switch changes positions. The output will look something like:

```
Func: TemplateCheckBattery   Event: BATTERY_DISCONNECTED   Param: 0xAF
Func: TemplateCheckBattery   Event: BATTERY_CONNECTED      Param: 0xB1
Func: TemplateCheckBattery   Event: BATTERY_DISCONNECTED   Param: 0xAF
```

Now, the first thing to do is to rename TemplateEventChecker.c/h to something more appropriate to your own code. This will break several things within ES_Configure.h and you will need to change those as well. Note that you should change the name of the function TemplateCheckBattery to something better, and make the changes in the header file and ES_Configure.h to match. Recompile and verify the test harness still works.

The next step is to add in your own event checker. Use the TemplateCheckBattery as a template, and make the appropriate modifications. Leave the lines with the #ifndef untouched except to change the Post function to the correct one for your project:

```
#ifndef EVENTCHECKER_TEST           // keep this as is for test harness
        PostGenericService(thisEvent);
#else
        SaveEvent(thisEvent);
#endif
```

This is needed for the test harness to be able to display events. If EVENTCHECKER_TEST is not defined (either in the file or in the project) then the event detector will post the event to the service post function. Otherwise, the event is saved into a module level variable for later printing.

SIMPLE SERVICE:

The simple service is simply a file that runs every time a specific event happens. While you could put a state machine inside the simple service, these are better handled with FSM and HSM template structures below. The most common use of a simple service is as a periodic function that is fired by an ES_TIMEOUT event so that it runs every x msec. Very often the underlying service is in fact a type of event checker that is slaved to run periodically for filtering purposes (e.g.: debouncing a switch, synchronous filtering on tape detectors). As such, the simple service is going to post an event to some other state machine once the event is correctly detected.

This section will lead you through the setup of a simple service project using the templates, and have you test it using both the Keyboard Inputs as well as the timeout events. The event detector inside the simple service is the same exact one as used in the EventCheckers section above.

To build a simple service project (which uses the full ES_Framework machinery), you will need to begin a new project following the same instructions in the NewMPLABXProject document. For the base project, you will need to add AD.c/h, BOARD.c/h, serial.c/h, as well as all of the ES_*.c/h files in the CMPE118 directory. All of these files should be added with an absolute link to their location (which is to say that you are using these files, not creating local copies of them).

Next you will create local copies of TemplateService.c/h exactly like you did for the EventChecker project above (see Figure 1). You will also need local copies of ES_Configure.h and TemplateES_Main.c, which you will need to modify (hence why you need local copies).

As you did in the EventChecker project, you will need to add the Enum_to_String.py script to the build chain (see Figure 2), and you will need to add a project macro SIMPLESERVICE_TEST in order to put the test code into the simple service (this will result in the service posting an event to itself and printing it out). This is done exactly like it was done in the EventChecker project (see Figure 3).

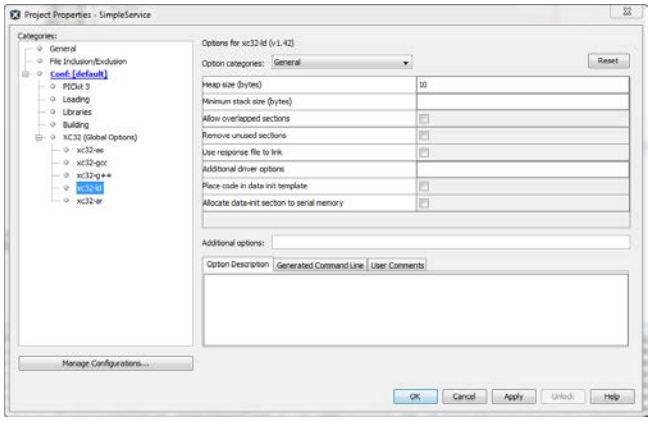

Figure 4: Adding a heap to the project

The test harness to run the Simple Service requires a heap for the printf() function. This is again done through the customization menu, “Customize …” -> XC32 (Global Options) -> xc32-ld, select on the general tab and add a heap size of 10 into the box (see *Figure 4*)

In order to run the simple service in test mode, various changes need to be made to the files in order to correctly configure the ES_Framework. The first modification is to TemplateES_Main.c, which is the file that contains main() and runs the framework. If you are working on a roach, you will need to insert the line (line 18):

```
Roach_Init();
```

Immediately after the comment `// Your hardware initialization function calls go here`. If you are working on an Uno32 stack, then you do not need to add this line. Save the file and open ES_Configure.h. Several changes will be made to this file in order to set up the ES_Framework to run the simple service using Keyboard Input as the initial test (later we will change this to run directly from a timeout event and test again). In ES_Configure.h change:

Uncomment line 21 and 23 (USE_KEYBOARD_INPUT and POSTFUNCTION_FOR_KEYBOARD_INPUT). At the end of line 23 add in PostTemplateService. The block should not be grayed out in MPLABX:

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

This will direct the Keyboard Input to send events to the TemplateService Post function (causing the simple service to run). Remember that no service will run until there is an event in its queue. On line 73, the header file needs to be (any) valid header file in your project, change this line to:

```
#define EVENT_CHECK_HEADER "ES_Configure.h"
```

Also delete everything that comes after EVENT_CHECK_LIST on line 77, so that the line looks like:

```
#define EVENT_CHECK_LIST
```

Note that these are both degenerate uses, but the simple service function test does not include conventional event checkers and this is needed for compilation. Your own projects will of course have their own event checkers (in their own project!).

On line 120, you will need to set the number of services to 2, and set the lines 140 to 146 as follows (this is where you tell the ES_Framework what to run when events get posted, and which queues to use to post events):

```
#define SERV_1_HEADER "TemplateService.h"
// the name of the Init function
#define SERV_1_INIT InitTemplateService
// the name of the run function
#define SERV_1_RUN RunTemplateService
// How big should this services Queue be?
#define SERV_1_QUEUE_SIZE 3
```

Save the ES_Configure.h file, and at this point you should be able to compile and run your test harness. What will happen now is that the Simple Service will run using the keyboard input (which you set the post function to the PostTemplateService function—that is all events from keyboard input will be directed to the simple service).

Load the code onto your roach/stack using the ds30Loader, again with the switch off. At this point, you will see the output of the Keyboard Input:

```
Starting ES Framework Template
using the 2nd Generation Events & Services Framework

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: BATTERY_CONNECTED      11: BATTERY_DISCONNECTED


Keyboard input is active, no other events except timer activations will be processed. You can
redisplay the event list by sending a 6 event.
Send an event using the form [event#]; or [event#]->[EventParam];
```

The Framework in now running using the keyboard input, and you can send an event to your simple service. Since most likely it is going to be a service running on a timer, you will send it a timeout event (7->0;) in the text box on top of the ds30Loader. Every time you do so, you will see a line in the serial window:

```
ES_TIMEOUT with parameter 0 was passed to PostTemplateService
```

This indicated that the simple service ran responding to an ES_TIMEOUT event. In order to see an actual event happen, you will need to switch the power switch on and off and send ES_TIMEOUT to the service using the keyboard. If you do this you will see:

```
ES_TIMEOUT with parameter 0 was passed to PostTemplateService
Event: BATTERY_CONNECTED     Param: 0x12E
ES_TIMEOUT with parameter 0 was passed to PostTemplateService
Event: BATTERY_DISCONNECTED  Param: 0x1
```

Congratulations, you have a fully running simple service that is tested. The next thing you are going to do is to hook it up to a real timer and trigger the simple service every 500msec (2Hz) and make sure it still responds. This will require a few changes to the ES_Configure.h file, as well as the TemplateService.c file in order to make sure that events are going to the right service and that they are being processed correctly.

Once again, most of the changes are within ES_Configure.h. The first thing to do is to comment out the USE_KEYBOARD_INPUT macros on lines 22 and 22. At this point if you compile it and run, nothing will happen as no event will go to the service. To fix that, set the TIMER0_RESP_FUNC to PostTemplateService on line 83:

```
#define TIMER0_RESP_FUNC PostTemplateService
```

It is also good form to give your timers meaningful names in the code, so that you can see by reading your code which timer is being used for what. Do this with a #define inserted just after line 107:

```
#define SIMPLE_SERVICE_TIMER 0
```

Now the service is set to run when the timer expires, but you will still need to initialize the timer (and reset it each time the service runs), in order to get the periodic response. This happens inside the init function and the

service run function. In the TemplateService.c file, add in a line #defining the timer interval (500) so that it can be changed easily. Insert this just after line 32:

```
#define TIMER_0_TICKS 500
```

In the init function, you will need to initialize the timer and set the countdown to 500msec. This is done by inserting the ES_Timer_InitTimer() function call on line 74:

```
ES_Timer_InitTimer(SIMPLE_SERVICE_TIMER,TIMER_0_TICKS);
```

Lastly, the run function needs to also reset the timer each time it gets an ES_TIMEOUT event, so that the next event will be generated. Insert the same line as above just below the `case ES_TIMEOUT:` on line 129. There is one more thing that needs to be done (just for cleanliness), that is to trap the response to ES_TIMERACTIVE so that it does not echo to the serial port. Every time a timer is started it generates an ES_TIMERACTIVE event, and every time it is stopped it generates an ES_TIMERSTOPPED event. These should be handled directly by the run function and ignored. To do this, add in a case to the switch statement, in between the ES_INIT and the ES_TIMEOUT:

```
case ES_TIMERACTIVE:
case ES_TIMERSTOPPED:
    break;
```

This will trap these events and ensure that they don't trigger the post to self function used for the test harness. Recompile the code and reload it onto your platform using the ds30Loader. You should see events only from the switch being turned on or off, and nothing else. Note that the response here will be at best ½ second between the switch and the echo, because the simple service is being run at 2Hz. Change that by changing the TIMER_0_TICKS to 100 (10Hz) and see if you notice it or not.

Now that the code is working and fully tested, go back and change the names of all of the functions and files to things that make sense for your lab/project. Go through the steps above and make sure it all works using both keyboard input and timers before you go and write your specific code. Inspect our code closely, and use it as a starting point for your own simple service functions.

While this might seem simple, you have just instantiated a fully functioning ES_Framework project with all of the complexity that this entails. Every project you write is going to be a variant of this kind of thing, and this is an excellent place to start each time to make sure things run correctly.

FSM's and HSM's:

Setting up and testing both FSM's and HSM's are in many ways more straightforward than either the EventCheckers or the SimpleService. The testing for both state machine templates will simply use the Keyboard Input to generate events, and the Tattle Tale to trace the event/state pairs. In this way, the template testing for the FSM's and HSM's are very much the same as you would use to debug your own project code.

Finite State Machine (FSM):

Begin with the FSM project. Here the steps are almost identical to the SimpleService project, with only minor modifications. Build a new project, add in the usual files (ES_*.c/h, AD.c/h, BOARD.c/h, and Serial.c/h). Once again, create local copies of ES_Configure.h, and TemplateES_Main.c and alter TemplateES_Main.c as above if using a roach. Create local copies of TemplateFSM.c/h, and move them to the appropriate places in your project.

Set the "Enum_to_String.py" script to run before building as above, and for the FSM project there are no project macros that need be defined. You again need to define a heap size (10 bytes is fine, see Figure 4). Once again, the project setup begins by modifying ES_Configure.h in a manner similar to the simple service project. Once again, uncomment USE_KEYBOARD_INPUT, and change the post function to PostTemplateFSM:

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

Once again, since there are no event checkers in this project, modify the EVENT_CHECK_HEADER to a valid header file, and blank out the EVENT_CHECK_LIST (on lines 73 and 77):

```
#define EVENT_CHECK_HEADER "ES_Configure.h"
#define EVENT_CHECK_LIST
```

Again, set the number of services to 2 (line 120), and change the service functions to the appropriate ones for the FSM in the block of code from lines 140 to 146:

```
#define SERV_1_HEADER "TemplateFSM.h"
// the name of the Init function
#define SERV_1_INIT InitTemplateFSM
// the name of the run function
#define SERV_1_RUN RunTemplateFSM
// How big should this services Queue be?
#define SERV_1_QUEUE_SIZE 3
```

Save your ES_Configure.h, and compile and build your code. Load it onto your target using the ds30Loader, and you should see output that looks like:

```
Starting ES Framework Template
using the 2nd Generation Events & Services Framework

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: BATTERY_CONNECTED       11: BATTERY_DISCONNECTED


Keyboard input is active, no other events except timer activations will be processed. You can
redisplay the event list by sending a 6 event.
Send an event using the form [event#]; or [event#]->[EventParam];

RunTemplateFSM(InitPState[ES_INIT,0])->RunTemplateFSM(InitPState[ES_EXIT,0])-
>RunTemplateFSM(FirstState[ES_ENTRY,0]);
```

The top part of the output is the familiar keyboard input screen. The highlighted text is output from the TattleTale that gives you the event/state trace that lets you know how the framework is processing events and moving through your state machine. In the case of the template FSM project, it shows that three steps were taken:

1. RunTemplateFSM was called with an ES_INIT event and was in state InitPState.
2. RunTemplateFSM was called with an ES_EXIT event and was in state InitPState.
3. RunTemplateFSM was called with and ES_ENTRY event and was in state FirstState.

This means that the ES_Framework initialized the state machine by sending an ES_INIT event to the state machine which starts in InitPState, and then it transitions to FirstState by calling ES_EXIT in InitPState, transitioning to FirstState, and calling ES_ENTRY on FirstState. This is the correct transition method between states, where you exit the first state, and then enter the next state. If your state machine does not use entry and exit functions, then you will simply ignore the entry and exit events (though they become quite useful especially in HSM's).

Just as with the simple service you added a real ES_TIMEOUT event in order to see the framework operate under normal settings, we will add in our events from our template event checker to ensure that you can see the entire framework run normally using the TattleTale (note that this exact same procedure would work with the HSM project as well).

The first step is to add the TemplateEventChecker.c/h to your project (this should be from the original EventChecker project you built, and you will want to use relative paths). The next is to make a few changes to

your ES_Configure.h file. Since the framework will now be running with real events, the first change is to comment out USE_KEYBOARD_INPUT and POSTFUNCTION_FOR_KEYBOARD_INPUT, lines 21 and 23. Add in the event checker header on line 73:

```
#define EVENT_CHECK_HEADER "TemplateEventChecker.h"
```

And add in the (one and only) event checker function on line 77:

```
#define EVENT_CHECK_LIST TemplateCheckBattery
```

The last modification necessary is in the TemplateEventChecker.c, which is that you need to set the post function to the correct one (note that since this file is being used in a different project, the project macro is NOT defined). The way to do this is to alter line 103:

```
PostTemplateFSM(thisEvent);
```

If you attempt to compile at this point, it will fail, as even though you have added the files to MPLAB-X project, you still have to put the paths into the compiler (see NewMPLABXProject Document). In order to do this you will need to do "Customize …" -> XC32 (Global Options) -> xc32-gcc select "Preprocessing and Messages" and add the path to your EventCheckers project in the "Include directories." You should now be able to rebuild and compile the FSM project, load it onto your target using the ds30Loader, and see the following output:

```
Starting ES Framework Template
using the 2nd Generation Events & Services Framework
RunTemplateFSM(InitPState[ES_INIT,0])->RunTemplateFSM(InitPState[ES_EXIT,0])-
>RunTemplateFSM(FirstState[ES_ENTRY,0]);
```

Note that this is the same init and entry function as before, but there is no keyboard input menu (as expected). If you now connect or disconnect the battery, that event will be sent to your FSM, and the output will look like:

```
RunTemplateFSM(FirstState[BATTERY_CONNECTED,B6]);
RunTemplateFSM(FirstState[BATTERY_DISCONNECTED,AF]);
```

Again, at this point you should rename the functions and files to something appropriate for your project, and make sure you can run the framework before modifying anything else.

HIERARCHICAL STATE MACHINE (HSM):

The Hierarchical State Machines (HSMs) are very similar to the FSMs, with the main difference that any given state can have a state machine under that state (sub-state machine). This turns out to be an extraordinarily efficient method for managing the complexity of reactive systems. The key insight is that general actions or responses to events are handled at the super-state level, and more specific actions are handled at the sub-state level. In order to implement this correctly, an incoming event is first passed down to the lower level, which checks if it handles the event (consumes it). If it does, then it performs its actions and returns an ES_NO_EVENT.

If the event is not handled at the lower level, then the event is returned to the level above it for handling it. That is, an event hits the HSM, and travels down to the lowest level. From there it is either consumed or returned to the level above it. It is very important to note that the event is not altered. It is either the original event, or ES_NO_EVENT. In the case where reaching some sub-state is itself an event, then the sub-state machine will post an event to the top level service.

The project setup for the HSM project is almost identical to the FSM project. Build a new project, add in the usual files (ES_*.c/h, AD.c/h, BOARD.c/h, and Serial.c/h). Once again, create local copies of ES_Configure.h, and TemplateES_Main.c and alter TemplateES_Main.c as above if using a roach. Create local copies of TemplateHSM.c/h and TemplateSubHSM.c/h. Move them to the appropriate places in your project.

Set the “Enum_to_String.py” script to run before building as above, and for the FSM project there are no project macros that need be defined. You again need to define a heap size (10 bytes is fine, see Figure 4). Once again, the project setup begins by modifying ES_Configure.h in a manner similar to the simple service project. Once again, uncomment USE_KEYBOARD_INPUT, and change the post function to PostTemplateHSM:

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

Once again, since there are no event checkers in this project, modify the EVENT_CHECK_HEADER to a valid header file, and blank out the EVENT_CHECK_LIST (on lines 73 and 77):

```
#define EVENT_CHECK_HEADER "ES_Configure.h"
#define EVENT_CHECK_LIST
```

Again, set the number of services to 2 (line 120), and change the service functions to the appropriate ones for the FSM in the block of code from lines 140 to 146:

```
#define SERV_1_HEADER "TemplateHSM.h"
// the name of the Init function
#define SERV_1_INIT InitTemplateHSM
// the name of the run function
#define SERV_1_RUN RunTemplateHSM
// How big should this services Queue be?
#define SERV_1_QUEUE_SIZE 3
```

Save your ES_Configure.h, and compile and build your code. Load it onto your target using the ds30Loader, and you should see output that looks like:

```
Starting ES Framework Template
using the 2nd Generation Events & Services Framework

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: BATTERY_CONNECTED      11: BATTERY_DISCONNECTED


Keyboard input is active, no other events except timer activations will be processed. You can
redisplay the event list by sending a 6 event.
Send an event using the form [event#]; or [event#]->[EventParam];

RunTemplateHSM(InitPState[ES_INIT,0])->RunTemplateSubHSM(InitPSubState[ES_INIT,0])-
>RunTemplateSubHSM(InitPSubState[ES_EXIT,0])->RunTemplateSubHSM(SubFirstState[ES_ENTRY,0])-
>RunTemplateHSM(InitPState[ES_EXIT,0])->RunTemplateHSM(FirstState[ES_ENTRY,0])-
>RunTemplateSubHSM(SubFirstState[ES_ENTRY,0]);
```

The top part of the output is the familiar keyboard input screen. The highlighted text is output from the TattleTale that gives you the event/state trace that lets you know how the framework is processing events and moving through your state machine. In the case of the template HSM project, it shows that the events move down through the levels of the HSM:

1. RunTemplateHSM was called with an ES_INIT event and was in state InitPState.
2. The ES_INIT event is passed down to the RunTemplateSubHSM which is in InitPSubState.
3. RunTemplateSubHSM exits InitPSubState with an ES_EXIT event.
4. RunTemplateSubHSM enters SubFirstState with an ES_ENTRY event.
5. RunTemplateFSM exits InitPState with an ES_EXIT event.
6. RunTemplateFSM enters FirstState with an ES_ENTRY event.
7. The ES_ENTRY event is passed down to the RunTemplateSubHSM which is in SubFirstState.

This means that the ES_Framework initialized the state machine by sending an ES_INIT event to the state machine which starts in InitPState, and then that event is passed down to the sub-state machine below it. The sub-state machine transitions to its first state (using an Exit/Entry) and then the upper level state machine transitions to its first state (using an Exit/Entry). Note that in a normal HSM, it is unlikely that the sub-state machine would be the same from multiple super-states (in this case both InitPState and FirstState call the same sub-state machine).

If you use the keyboard input to send another event to the HSM, for example an ES_TIMEOUT event, you will get the following output:

```
ES_TIMEOUT with parameter 0 was passed to PostTemplateHSM
RunTemplateHSM(FirstState[ES_TIMEOUT,0])->RunTemplateSubHSM(SubFirstState[ES_TIMEOUT,0]);
```

This shows the keyboard input posting the event to the appropriate queue, and then the HSM responding by receiving the event at the top level (RunTemplateHSM) and then passing that down to the sub-level (RunTemplateSubHSM). This is exactly correct.

You can add in the real event checker identically to what you did for the FSM above, with the only change being that the post function in the TemplateEventChecker.c file on line 103 should be `PostTemplateHSM(thisEvent);`

Again, at this point you should rename the functions and files to something appropriate for your project, and make sure you can run the framework before modifying anything else.

CONCLUSIONS:

Each and every time you make a new ES_Framework project, follow the steps above to ensure you have the entire machinery functioning (and everything renamed to something appropriate). You will find that the use of the test harnesses provided (Keyboard Input and TattleTale) will help you to debug your projects and ensure that they are working correctly.

Modularity is key in this. Keep your event checkers in their own project so that you can test them and ensure they are working, but link them into your main project. This will help you manage the complexity of the project and ensure that your various modules can be tested independently. Note that you will most likely want to keep a single or a small number of ES_Configure.h files so that you don't have to keep them synchronized. Make sure you have the right one associated with your project.