FSMLang Documentation

version 1.46

2025, The Maintainers

June 11, 2025

Contents

FSMLang: Better State Machine Design

FSMLang is was created to allow design work in the problem space of finite state machines without the encumbrances of any particular implementation language. Thus, FSMLang is implemented as a “pre-processor,” generating code in any desired general programming language to implement the described finite state machine. FSMLang allows effort to be focused on the definition of events, states, and transitions. Indeed, though the action to be taken in any particular event/state intersection is declarable, the actual definition of that action is treated as a detail which falls outside the scope of FSMLang. Moreover, the mechanisms for collecting or determining events are also outside the language scope. FSMLang creates an object or objects in a target programming language which, when inserted into the larger program structure will invoke the correct actions and make the correct transitions for the events handed it.

“Any general programming language”

Presently, FSMLang only supports the C programming language. It is left as an exercise to the reader to develop support for other languages. The FSMLang maintainers will supply what help is needed.

The created state machine contains a single state variable, which should not be manipulated by any user-written function. This variable is maintained on the heap, not on the machine’s function call stack. This means that the machine must not be called recursively; neither from within any action function, nor from separate threads of execution. The keyword `reentrant` can be used to designate machines which are called from different execution threads. Macros will be inserted at the beginning and end of the state machine function which are to be defined by the user to properly protect the machine from re-entrance.

Supporting the desire to keep FSMLang focused on machine design, provision is made to insert `native` blocks into the machine specification files. These blocks are not interpreted by FSMLang, but are copied into the generated source at different points. The blocks also serve to reduce or eliminate the need to post-process the generated code.

Data can be specified for both machines and events. States may have entry and exit functions. Transitions may be done to a specified state, or through the auspices of a transition function. A machine may have a transition function called when any state transition is made. And, machines may have sub-machines (to any (keep it reasonable!) depth).

FSMLang supports designing with action functions that return events (the default), states, or nothing. Returning events from action functions is a powerful technique for the machine to immediately feed events to itself, simplifying, for example, the management of internal errors. Imagine an action function which must allocate some memory to complete its task; should that memory not be available, the function can simply exit, returning a “memory not available” event; the state machine processing for this situation can be placed in another action function, and the machine can be transitioned to an error state, should the situation be unrecoverable. The core FSM function loops, calling appropriate actions, then making designated transitions, until an action function returns `noEvent`.

Having said all of that, useful state machines can be designed with actions that return states or nothing.

The language syntax is intended to make the action the state machine will take in response to any input immediately clear. Nevertheless, a UML state chart or a simple HTML event-state action matrix are useful tools for state machine visualization. FSMLang can output both HTML and PlantUML to assist in visualizing and documenting the state machine. For somewhat more formal documentation, ReStructuredText output is available, as well.

Both the language and this documentation are intended to assist in the design of useful state machines. Feedback is welcomed, both for language and for documentation improvements. Use the *issue* mechanisms in either GitHub repo.

The repos are at Language and Documents.

Revision history is maintained for the curious, and for those who support medical device development tool validation.

This documentation presents language concepts primarily through two examples, The Simple Communicator, and The HSM Communicator. Data for Machines and Events shows how FSMLang handles data for both machines and events, and States and Transitions rounds out the discussion of states.

The Simple Communicator

The Simple Communicator is used to illustrate basic FSMLang features.

The Simple Communicator

As an example, consider a simple communications protocol which specifies that an acknowledgement must be received for each message before another may be sent. The sender of messages in such a protocol must have at least two states: In the first state, which will be called IDLE, the sender has sent no message. In the second, which will be called AWAITING_ACK, the sender has sent a message and is awaiting an acknowledgement. The events which this automaton will see are SEND_MESSAGE, the request by a client entity for a message to be sent, and ACK, the receipt of the acknowledgment from the peer automaton.

The valid actions, then, are to send a message if one is requested, and the automaton is in the IDLE state,and to simply return to the IDLE state if an ACK is received while in the AWAITING_ACK state. Sending a message requires a transition to the AWAITING_ACK state. The receipt of an acknowledgement while in the IDLE state represents a protocol error which may be safely ignored. A request to send a message while in the AWAITING_ACK state, however, must result in the message being queued for later transmission.

FSM Source

Using FSMLang, this machine can be described this way:

```
/**
   This machine manages communications using a "stop and wait" protocol.
   Only one message is allowed to be outstanding.
*/
machine simpleCommunicator
{

   state IDLE
         , AWAITING_ACK
         ;

   event SEND_MESSAGE
         , ACK
         ;

   /**
      Since we're idle, we can simply send the message.  Transitioning
      to the AWAITING_ACK state ensures that any other messages
      we're asked to send will be queued.
   */
   action sendMessage[SEND_MESSAGE,IDLE] transition AWAITING_ACK;

   /**
     Since we're busy, we must queue the message for later
```

```
      sending.  The queue will be checked when the ACK
      is received.
    */
    action queueMessage[SEND_MESSAGE,AWAITING_ACK];

    /**
       We've received our ACK for the previous message.  It is
       time to check for any others.
    */
    action checkQueue[ACK,AWAITING_ACK] transition IDLE;

    /* these lines are informational; they affect the html output,
       but do not affect any code generated.
    */

    /** queueMessage adds a message to the queue */
    queueMessage returns noEvent;

    /** sendMessage sends a message from the queue.  The
        message is expected to be there, since
        checkQueue will have been previously called.
    */
    sendMessage  returns noEvent;

    /** checkQueue only checks; it does not dequeue; that
        is done by sendMessage.

        Return SEND_MESSAGE when the queue is not empty.
    */
    checkQueue   returns SEND_MESSAGE, noEvent;

}
```

The layout of the FSM source is expected to feel familar to those familiar with C. Moreover, it is intended to mimic the textual description of the machine.

FSMLang can create three different layouts for the event, state, action, and transition data. The default (or `-tc` option) is to simply build an action array dimensioned by event and state having member structures to indicate the action and transitions to take when that combination of event and state are encountered. The second approach (`-ts`) creates an array of state functions, each containing a switch statement for the event being handled. The third approach (`-te`) reverses the second, creating an array of event functions, each containing a switch for the current state. As usual, the trade-offs between these three approaches involve size and speed.

We'll look at the code generated by each of the three, starting with the parts that are common.

Common Output

All variants produce the same files, namely:

Headers:

- simpleCommunicator.h
- simpleCommunicator_events.h
- simpleCommunicator_priv.h

Source:

- simpleCommunicator.c

simpleCommunicator.h is intended to be used by client code to invoke the state machine. Thus, it contains the single point-of-entry function.

```
void run_simpleCommunicator(SIMPLE_COMMUNICATOR_EVENT);
```

It also supplies convenience macros by which client code can refer to machine events by their short names:

```
#undef THIS
#define THIS(A) simpleCommunicator_##A
#endif
```

It also includes *simpleCommunicator_events.h*, which contains the event enumeration:

```
typedef enum SIMPLE_COMMUNICATOR_EVENT {
    simpleCommunicator_SEND_MESSAGE
   , simpleCommunicator_ACK
   , simpleCommunicator_noEvent
   , simpleCommunicator_numEvents
   , simpleCommunicator_numAllEvents = simpleCommunicator_numEvents
}  SIMPLE_COMMUNICATOR_EVENT;
```

Taken together, cliend code intending to have the *simpleCommunicator* send a message need only write:

```
run_simpleCommunicator(THIS(SEND_MESSAGE));
```

And, the code which knows how to receive the ACK from the peer, can simply write:

```
run_simpleCommunicator(THIS(ACK));
```

simpleCommunicator_priv.h is intended to be included by files implementing the machine's action and other user defined functions. It is also included by *simpleCommunicator.c* Here is the common material:

```
#include "simpleCommunicator.h"

#ifdef SIMPLE_COMMUNICATOR_DEBUG
#include <stdio.h>
#include <stdlib.h>
#endif


#ifdef SIMPLE_COMMUNICATOR_DEBUG
extern char *SIMPLE_COMMUNICATOR_EVENT_NAMES[];
extern char *SIMPLE_COMMUNICATOR_STATE_NAMES[];
#endif

typedef enum {
   simpleCommunicator_IDLE
   , simpleCommunicator_AWAITING_ACK
   , simpleCommunicator_numStates
}  SIMPLE_COMMUNICATOR_STATE;

typedef struct _simpleCommunicator_struct_ SIMPLE_COMMUNICATOR;
#undef FSM_TYPE_PTR
#define FSM_TYPE_PTR pSIMPLE_COMMUNICATOR
```

```
extern SIMPLE_COMMUNICATOR simpleCommunicator;

typedef SIMPLE_COMMUNICATOR_EVENT (*SIMPLE_COMMUNICATOR_ACTION_FN)(FSM_TYPE_PTR);

typedef void (*SIMPLE_COMMUNICATOR_FSM)(FSM_TYPE_PTR,SIMPLE_COMMUNICATOR_EVENT);

void simpleCommunicatorFSM(FSM_TYPE_PTR,SIMPLE_COMMUNICATOR_EVENT);

ACTION_RETURN_TYPE simpleCommunicator_sendMessage(FSM_TYPE_PTR);
ACTION_RETURN_TYPE simpleCommunicator_queueMessage(FSM_TYPE_PTR);
ACTION_RETURN_TYPE simpleCommunicator_checkQueue(FSM_TYPE_PTR);
ACTION_RETURN_TYPE simpleCommunicator_noAction(FSM_TYPE_PTR);
```

FSMLang provides the ability to debug the operation of the state machine through the use of several macros and name lists, made available by defining (in this case, for example), SIMPLE_COMMUNICATOR_DEBUG during compilation.

The source file diverges quickly between the three variants. Debugging support, including the generation of weak versions of user functions comprise most of the common material.

```
#ifndef DBG_PRINTF
#define DBG_PRINTF(...)
#endif

SIMPLE_COMMUNICATOR_EVENT __attribute__((weak)) UFMN(sendMessage)(FSM_TYPE_PTR pfsm)
{
   DBG_PRINTF("weak: %s", __func__);
   (void) pfsm;
   return THIS(noEvent);
}

SIMPLE_COMMUNICATOR_EVENT __attribute__((weak)) UFMN(queueMessage)(FSM_TYPE_PTR pfsm)
{
   DBG_PRINTF("weak: %s", __func__);
   (void) pfsm;
   return THIS(noEvent);
}

SIMPLE_COMMUNICATOR_EVENT __attribute__((weak)) UFMN(checkQueue)(FSM_TYPE_PTR pfsm)
{
   DBG_PRINTF("weak: %s", __func__);
   (void) pfsm;
   return THIS(noEvent);
}

SIMPLE_COMMUNICATOR_EVENT __attribute__((weak)) UFMN(noAction)(FSM_TYPE_PTR pfsm)
{
   DBG_PRINTF("weak: %s", __func__);
   (void) pfsm;
   return THIS(noEvent);
}


#ifdef SIMPLE_COMMUNICATOR_DEBUG
```

```
char *SIMPLE_COMMUNICATOR_EVENT_NAMES[] = {
    "simpleCommunicator_SEND_MESSAGE"
   ,"simpleCommunicator_ACK"
   , "simpleCommunicator_noEvent"
   , "simpleCommunicator_numEvents"
};

char *SIMPLE_COMMUNICATOR_STATE_NAMES[] = {
    "simpleCommunicator_IDLE"
   ,"simpleCommunicator_AWAITING_ACK"
};

#endif
```

Note

For action array output, it may be necessary to inhibit the generation of the weak functions (`--generate-weak-fns=false`). Some linkers get confused by the presence of these weak definitions in the same file as the action array is defined.

The final bit of commonality is the looping nature of the main FSM function, seen because this machine has actions which return events:

```
void simpleCommunicatorFSM(pSIMPLE_COMMUNICATOR pfsm, SIMPLE_COMMUNICATOR_EVENT event)
{
   SIMPLE_COMMUNICATOR_EVENT new_e;
   SIMPLE_COMMUNICATOR_EVENT e = event;

   while (e != THIS(noEvent)) {

       /* e = (somehow call an action function to get a new event) */;

   }
}
```

With that, let's look at how the three variants differ.

Action Array Output

The action array first appears in the private header as a type definition for the structure to hold the action function pointer and the desired state transition. This is then used in the FSM structure type definition to declare the action array, dimensioned by the number of events and states.

```
typedef enum {
   simpleCommunicator_IDLE
   , simpleCommunicator_AWAITING_ACK
   , simpleCommunicator_numStates
}  SIMPLE_COMMUNICATOR_STATE;

typedef void (*SIMPLE_COMMUNICATOR_FSM)(FSM_TYPE_PTR,SIMPLE_COMMUNICATOR_EVENT);

typedef struct _simpleCommunicator_action_trans_struct_ {
```

```
  SIMPLE_COMMUNICATOR_ACTION_FN      action;
  SIMPLE_COMMUNICATOR_STATE          transition;
} SIMPLE_COMMUNICATOR_ACTION_TRANS, *pSIMPLE_COMMUNICATOR_ACTION_TRANS;

struct _simpleCommunicator_struct_ {
   SIMPLE_COMMUNICATOR_STATE                  state;
   SIMPLE_COMMUNICATOR_EVENT                  event;
   SIMPLE_COMMUNICATOR_ACTION_TRANS   const   (*actionArray)[THIS(numEvents)][simpleCommunicator_numStates];
   SIMPLE_COMMUNICATOR_FSM                    fsm;
};
```

The source file, *simpleCommunicator.c*, declares the array:

```
static const SIMPLE_COMMUNICATOR_ACTION_TRANS simpleCommunicator_action_array[THIS(numEvents)][simpleCommunicator_numStates] =
{
   {
      /* -- SEND_MESSAGE -- */

      /* -- IDLE -- */             {UFMN(sendMessage), simpleCommunicator_AWAITING_ACK}
      /* -- AWAITING_ACK -- */   , {UFMN(queueMessage),simpleCommunicator_AWAITING_ACK}
   },
   {
      /* -- ACK -- */

      /* -- IDLE -- */             {UFMN(noAction),  simpleCommunicator_IDLE}
      /* -- AWAITING_ACK -- */   , {UFMN(checkQueue),simpleCommunicator_IDLE}
   },
};
```

In practice, this array can become quite large; the benefit, of course, being speed of action function lookup and execution, as seen in the main FSM function:

```
void simpleCommunicatorFSM(pSIMPLE_COMMUNICATOR pfsm, SIMPLE_COMMUNICATOR_EVENT event)
{
   SIMPLE_COMMUNICATOR_EVENT new_e;
   SIMPLE_COMMUNICATOR_EVENT e = event;

   while (e != THIS(noEvent)) {

#ifdef SIMPLE_COMMUNICATOR_DEBUG
      if (EVENT_IS_NOT_EXCLUDED_FROM_LOG(e))
      {
         DBG_PRINTF("event: %s; state: %s"
                    ,SIMPLE_COMMUNICATOR_EVENT_NAMES[e]
                    ,SIMPLE_COMMUNICATOR_STATE_NAMES[pfsm->state]
                   );
      }
#endif

      /* This is read-only data to facilitate error reporting in action functions */
      pfsm->event = e;

      new_e = ((* (*pfsm->actionArray)[e][pfsm->state].action)(pfsm));

      pfsm->state = (*pfsm->actionArray)[e][pfsm->state].transition;

      e = new_e;

   }

}
```

State Function Array

The state function array also first appears in the private header:

```
typedef ACTION_RETURN_TYPE (*SIMPLE_COMMUNICATOR_STATE_FN)(pSIMPLE_COMMUNICATOR,SIMPLE_COMMUNICATOR_EVENT);

static const SIMPLE_COMMUNICATOR_STATE_FN simpleCommunicator_state_fn_array[simpleCommunicator_numStates];

struct _simpleCommunicator_struct_ {
   SIMPLE_COMMUNICATOR_STATE              state;
   SIMPLE_COMMUNICATOR_EVENT              event;
   SIMPLE_COMMUNICATOR_STATE_FN   const (*statesArray)[simpleCommunicator_numStates];
   SIMPLE_COMMUNICATOR_FSM                fsm;
};
```

The state functions need the event the machine is currently handling in addition to a pointer to the machine structure.

The array and main FSM function are again defined in the source file:

```
static SIMPLE_COMMUNICATOR_EVENT IDLE_stateFn(pSIMPLE_COMMUNICATOR,SIMPLE_COMMUNICATOR_EVENT);
static SIMPLE_COMMUNICATOR_EVENT AWAITING_ACK_stateFn(pSIMPLE_COMMUNICATOR,SIMPLE_COMMUNICATOR_EVENT);
static const SIMPLE_COMMUNICATOR_STATE_FN simpleCommunicator_state_fn_array[simpleCommunicator_numStates] =
{
     IDLE_stateFn
   , AWAITING_ACK_stateFn
};

void simpleCommunicatorFSM(pSIMPLE_COMMUNICATOR pfsm, SIMPLE_COMMUNICATOR_EVENT event)
{
   SIMPLE_COMMUNICATOR_EVENT e = event;

   while (e != simpleCommunicator_noEvent) {

#ifdef SIMPLE_COMMUNICATOR_DEBUG
      if (EVENT_IS_NOT_EXCLUDED_FROM_LOG(e))
      {
         DBG_PRINTF("event: %s; state: %s"
                    ,SIMPLE_COMMUNICATOR_EVENT_NAMES[e]
                    ,SIMPLE_COMMUNICATOR_STATE_NAMES[pfsm->state]
                   );
      }
#endif

      /* This is read-only data to facilitate error reporting in action functions */
      pfsm->event = e;

      e = ((* (*pfsm->statesArray)[pfsm->state])(pfsm,e));
   }

}
```

While the lookup into the state table is quick, the state function retrieved must first be called before the action function can finally be located and called. Some efficiency is gained, however, in the assignment of any new state, since this can be done within the state functions.

```
static SIMPLE_COMMUNICATOR_EVENT IDLE_stateFn(pSIMPLE_COMMUNICATOR pfsm,SIMPLE_COMMUNICATOR_EVENT e)
{
   SIMPLE_COMMUNICATOR_EVENT retVal = THIS(noEvent);

   switch(e)
   {
   case THIS(SEND_MESSAGE):
      retVal = UFMN(sendMessage)(pfsm);
      pfsm->state = simpleCommunicator_AWAITING_ACK;
      break;
```

```
   default:
      DBG_PRINTF("simpleCommunicator_noAction");
      break;
   }

   return retVal;
}

static SIMPLE_COMMUNICATOR_EVENT AWAITING_ACK_stateFn(pSIMPLE_COMMUNICATOR pfsm,SIMPLE_COMMUNICATOR_EVENT e)
{
   SIMPLE_COMMUNICATOR_EVENT retVal = THIS(noEvent);

   switch(e)
   {
   case THIS(SEND_MESSAGE):
      retVal = UFMN(queueMessage)(pfsm);
      break;
   case THIS(ACK):
      retVal = UFMN(checkQueue)(pfsm);
      pfsm->state = simpleCommunicator_IDLE;
      break;
   default:
      DBG_PRINTF("simpleCommunicator_noAction");
      break;
   }

   return retVal;
}
```

Event Function Array

The final variant offers the chance of a bit of efficiency when an event is handled identically in all states, and has no associated transition. In this case, the action function itself can be placed in the array of event handlers. Otherwise, of course, an “event function” must be created, which will then result in a double lookup, as is the case with the state function implementation.

To see the insertion of an action function into the event handler array, we add an additional event and specify an action for it:

```
event NEVER_SEEN;

action neverExecuted[NEVER_SEEN, (IDLE, AWAITING_ACK)];
```

Notice the use of the state vector (event vectors are also allowed) to indicate that the action should be taken when the event occurs in any of the states in the vector. When both event and state vectors are used, the action is placed in the action matrix cells representing the cross-product of the two vectors.

The private header:

```
#define simpleCommunicator_numMachineEvents 2
struct _simpleCommunicator_struct_ {
   SIMPLE_COMMUNICATOR_STATE         state;
   SIMPLE_COMMUNICATOR_EVENT         event;
   SIMPLE_COMMUNICATOR_ACTION_FN const (*eventsArray)[simpleCommunicator_numMachineEvents];
   SIMPLE_COMMUNICATOR_FSM           fsm;
};
```

No new function typedef is needed for the event handlers, as the event handler must take the same input as an action function. The macro giving the number of machine events is neccessary because the event enumeration counts the *noEvent* entry. That “event” is never handled.

The source:

```
static ACTION_RETURN_TYPE simpleCommunicator_handle_SEND_MESSAGE(FSM_TYPE_PTR);
static ACTION_RETURN_TYPE simpleCommunicator_handle_ACK(FSM_TYPE_PTR);

static const SIMPLE_COMMUNICATOR_ACTION_FN simpleCommunicator_event_fn_array[simpleCommunicator_numMachineEvents] =
{
   simpleCommunicator_handle_SEND_MESSAGE
   , UFMN(neverExecuted)
   , simpleCommunicator_handle_ACK
};

void simpleCommunicatorFSM(pSIMPLE_COMMUNICATOR pfsm, SIMPLE_COMMUNICATOR_EVENT event)
{
   SIMPLE_COMMUNICATOR_EVENT e = event;

   while (e != simpleCommunicator_noEvent) {

#ifdef SIMPLE_COMMUNICATOR_DEBUG
      if (EVENT_IS_NOT_EXCLUDED_FROM_LOG(e))
      {
         DBG_PRINTF("event: %s; state: %s"
                    ,SIMPLE_COMMUNICATOR_EVENT_NAMES[e]
                    ,SIMPLE_COMMUNICATOR_STATE_NAMES[pfsm->state]
                   );
      }
#endif

      /* This is read-only data to facilitate error reporting in action functions */
      pfsm->event = e;

      e = ((* (*pfsm->eventsArray)[e])(pfsm));

   }

}

static ACTION_RETURN_TYPE simpleCommunicator_handle_SEND_MESSAGE(FSM_TYPE_PTR pfsm)
{
   ACTION_RETURN_TYPE event = THIS(noEvent);

   switch (pfsm->state)
   {
      case STATE(IDLE):
         event = UFMN(sendMessage)(pfsm);
         pfsm->state = STATE(AWAITING_ACK);
         break;
      case STATE(AWAITING_ACK):
         event = UFMN(queueMessage)(pfsm);
         break;
   }

   return event;
}

static ACTION_RETURN_TYPE simpleCommunicator_handle_ACK(FSM_TYPE_PTR pfsm)
{
   ACTION_RETURN_TYPE event = THIS(noEvent);

   switch (pfsm->state)
   {
      case STATE(AWAITING_ACK):
         event = UFMN(checkQueue)(pfsm);
         pfsm->state = STATE(IDLE);
         break;
      default:
         DBG_PRINTF("simpleCommunicator_noAction");
         break;
   }

   return event;
}
```

As with the state functions, the generated event functions handle any needed transition.

Summary

Though implemented differently, the three variants produce code which exhibits the same behavior, given the same event input. The variants are made available to accomodate different requirements in the size/speed tradeoff. As observed earlier, the Action Array is the quickest of the three, but when the machine is “sparse,” having a low percentage of the event-state cells filled, the array is mostly empty, so size considerations would point to either a state or event based array of handlers. On the other hand, when the machine is not sparse, the size differences are not as great, so the speed of the array may be attractive.

Native Language Considerations

FSMLang provides for including user code in the generated files through the use of `native` blocks. These are available to be exploited by any output language generator. This section explores the use of these blocks, and, since the current “native language” output is C, some comments are made on the convenience and debugging macros available in the generated code.

Native Blocks

Native blocks come in two flavors: one which will be placed in the machine’s private header file; and another which will be placed in the source file.

The following block is placed at the beginning of the generated headers:

```
native
{
   #ifdef SILLY_MACHINE_DEBUG
   #define DBG_PRINTF(...) printf(__VA_ARGS__); printf("\n");
   #else
   #define DBG_PRINTF
   #endif

}
machine sillyMachine
{
   event e1;
   state s1;

   action a1[e1,s1];
}
```

Here it is in the private header:

```
/**
   sillyMachine_priv.h

   This file automatically generated by FSMLang
*/

#ifndef _SILLYMACHINE_PRIV_H_
#define _SILLYMACHINE_PRIV_H_

#include "sillyMachine.h"
#ifndef SILLY_MACHINE_NATIVE_PROLOG
#define SILLY_MACHINE_NATIVE_PROLOG
```

```
#ifdef SILLY_MACHINE_DEBUG
#define DBG_PRINTF(...) printf(__VA_ARGS__); printf("\n");
#else
#define DBG_PRINTF
#endif

#endif
```

Native material such as this, of course, is not needed in every header, and, indeed, must be protected against multiple inclusions. However, there are situations (such as the use of pragmas to silence undesired warnings (better would be to log a bug with the FSMLang maintainers, though!)), which require the block to be placed in all headers.

Alert readers noticed the multiple-inclusion protection macro incorporated the word, *PROLOG*. Indeed, the keyword `prologue` can be used with `native` (though, it is the default), as can the keyword `epilogue`. `native epilogue` blocks are placed at the end of the headers. Both blocks can be specified; both come *before* the `machine` declaration.

```
native
{
#ifdef SILLY_MACHINE_DEBUG
#define DBG_PRINTF(...) printf(__VA_ARGS__); printf("\n");
#else
#define DBG_PRINTF
#endif

#pragma stop warnings!
}

native epilogue
{
#pragma resume warnings!
}

machine sillyMachine
{
   event e1;
   state s1;

   action a1[e1,s1];
}

/**
        sillyMachine.h

        This file automatically generated by FSMLang
*/

#ifndef _SILLYMACHINE_H_
#define _SILLYMACHINE_H_

#include "sillyMachine_events.h"
#ifndef SILLY_MACHINE_NATIVE_PROLOG
#define SILLY_MACHINE_NATIVE_PROLOG
```

```
#ifdef SILLY_MACHINE_DEBUG
#define DBG_PRINTF(...) printf(__VA_ARGS__); printf("\n");
#else
#define DBG_PRINTF
#endif

#pragma stop warnings!

#endif
#define FSM_VERSION "1.45.1"

#ifndef NO_CONVENIENCE_MACROS
#undef UFMN
#define UFMN(A) sillyMachine_##A
#undef THIS
#define THIS(A) sillyMachine_##A
#endif
#undef STATE
#define STATE(A) sillyMachine_##A

#undef ACTION_RETURN_TYPE
#define ACTION_RETURN_TYPE SILLY_MACHINE_EVENT

void run_sillyMachine(SILLY_MACHINE_EVENT);

typedef struct _sillyMachine_struct_ *pSILLY_MACHINE;
extern pSILLY_MACHINE psillyMachine;

#ifndef SILLY_MACHINE_NATIVE_EPILOG
#define SILLY_MACHINE_NATIVE_EPILOG

#pragma resume warnings!

#endif

#endif
```

Native implementation (both prologue and epilogue) blocks are placed in the machine's generated source file. As with the native header blocks, any legal language construct may be placed here. The default `prologue` may be omitted, and `implementation` may be shortened to `impl`.

For example, this fsmlang:

```
native
{
#ifdef SILLY_MACHINE_DEBUG
#define DBG_PRINTF(...) printf(__VA_ARGS__); printf("\n");
#else
#define DBG_PRINTF
#endif

#pragma stop warnings!
}
```

```
native epilogue
{
#pragma resume warnings!
}

machine sillyMachine
native implementation
{
#pragma stop warnings!
}
native impl epilogue
{
#pragma resume warnings!
}
{
   event e1;
   state s1;

   action a1[e1,s1];
}
```

Yields (the middle has been cut out):

```
/**
        sillyMachine.c

        This file automatically generated by FSMLang
*/

#include "sillyMachine_priv.h"
#include <stddef.h>

/* Begin Native Implementation Prolog */


#pragma stop warnings!

/* End Native Implementation Prolog */

/*
 Lines omitted for brevity.
*/

/* Begin Native Implementation Epilog */


#pragma resume warnings!

/* End Native Implementation Epilog */
```

Note that native implementation blocks come after the machine name, but before the machine's opening brace.

Macros: User Functions and Debugging

Some of the macros are most valuable in a hierarchical machine setting, and others are most valuable in settings where events have data. Neither of these topics has yet been addressed, but placing the table here, in the context of things done to help C programmers seemed good.

Macros

Name	Definition

ACTION_RETURN_TYPE	Gives the type returned by the machine's action functions. When actions return events, this macro is set to the event enumeration, even when events have data. This is because within the state machine only the enumeration is returned.
	Using this macro in the definition of action functions obviates the need to adjust function return type should the design of events (having/not having data) change.
	`#define ACTION_RETURN_TYPE HSM_COMMUNICATOR_EVENT`
	`ACTION_RETURN_TYPE hsmCommunicator_startSessionEstablishment(FSM_TYPE_PTR);`

FSM_TYPE_PTR	Gives the type of the pointer to the current machine's structure.
	Using this macro, along with UFMN, ACTION_RETURN_TYPE, and THIS facilitates machine redesign. Adding or removing sub-machines, with the concommitant re-shuffling of action functions is much easier, since the macro definitions will change appropriately in the function's new setting.
	`#define FSM_TYPE_PTR pHSM_COMMUNICATOR`
	`ACTION_RETURN_TYPE UFMN(startSessionEstablishment)(FSM_TYPE_PTR);`

THIS(A)	THIS prepends the machine name prefix to the given argument. `return THIS(SEND_MESSAGE);` will always return the correct event enumeration.
PARENT(A)	Similar to THIS, PARENT prepends the name of the machine's parent to the given argument. `return PARENT(SESSION_ESTABLISHED)` will always return the correct event enumeration.
UFMN(A)	UFMN is to user function names as THIS is to event names. That is, The name of the this function: `ACTION_RETURN_TYPE UFMN(startSessionEstablishment)(FSM_TYPE_PTR);` will always expand correctly, even when the function is moved from one machine to another, and when the use of the CL argument `--short-user-fn-names` is changed.

State machine design, as all code design, is iterative. The macros above facilitate this activity. The two debugging macros below help identify the pain points which may prompt the redesign.

`DBG_PRINTF` is used to output useful debug information. It must be defined to accept variadic arguments. If it is not defined by the time compilation gets to the source file, it will be defined to be vacuous.

`<UPPER_CASE_MACHINE_NAME>_DEBUG` must be defined during compilation in order to see the debug output.

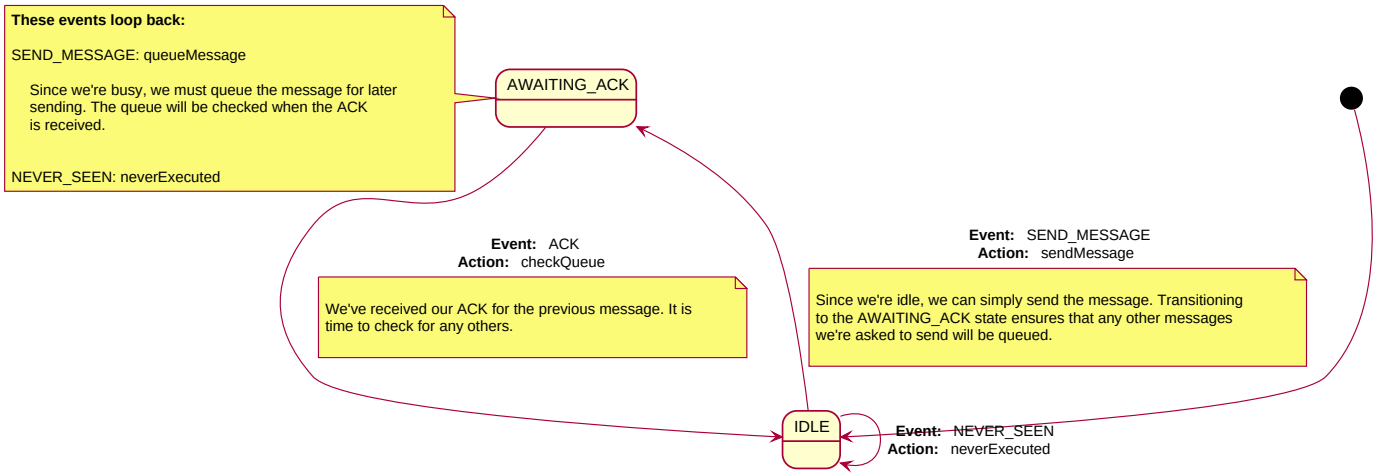
Visualizing the FSM

HTML Output

An HTML page can be generated with the `-th` command line switch.

PlantUML

Or, the PlantUML code for creating the following diagram can be generated with the `-tp` command line switch.



simpleCommunicator

This machine manages communications using a “stop and wait” protocol. Only one message is allowed to be outstanding.

Machine Statistics

Number of events:	3
Events not handled:	0
Events handled in one state:	1
At least one event handled the same in all states?	yes
Number of states:	2
Number of states with entry functions:	0
Number of states with exit functions:	0
States handling no events:	0

Number of events:	3
States handling exactly one event:	0
States with no way in:	0
States with no way out:	0

State Chart

	SEND_MESSAGE	NEVER_SEEN	ACK
IDLE	sendMessage transition: AWAITING_ACK	neverExecuted	
AWAITING_ACK	queueMessage	neverExecuted	checkQueue transition: IDLE

Events

SEND_MESSAGE

These states handle this event:

- IDLE
- AWAITING_ACK

This yields a state density of 100%.

These actions are taken in response to this event:

- sendMessage
- queueMessage

NEVER_SEEN

This event is handled identically in all states.

These states handle this event:

- IDLE
- AWAITING_ACK

This yields a state density of 100%.

These actions are taken in response to this event:

- neverExecuted

ACK

This event is handled identically in 1 states.

These states handle this event:

- AWAITING_ACK

This yields a state density of 50%.

These actions are taken in response to this event:

- checkQueue

States

IDLE

These events are handled in this state:

- NEVER_SEEN
- SEND_MESSAGE

This yields an event density of 66%.

These actions are taken in this state:

- neverExecuted
- sendMessage

These states transition into this state:

- AWAITING_ACK

This state transitions into these states:

- AWAITING_ACK

AWAITING_ACK

These events are handled in this state:

- NEVER_SEEN
- SEND_MESSAGE
- ACK

This yields an event density of 100%.

These actions are taken in this state:

- neverExecuted
- queueMessage
- checkQueue

These states transition into this state:

- IDLE

This state transitions into these states:

- IDLE

Actions

neverExecuted

sendMessage

sendMessage sends a message from the queue. The message is expected to be there, since checkQueue will have been previously called.

This action returns:

- noEvent

queueMessage

queueMessage adds a message to the queue

This action returns:

- noEvent

checkQueue

checkQueue only checks; it does not dequeue; that is done by sendMessage.

Return SEND_MESSAGE when the queue is not empty.

This action returns:

- SEND_MESSAGE
- noEvent

Gnu Make

Supporting FSMLang

The FSMLang development is done in C. And, since a large part of the development tree is devoted to testing, the maintainers have put some effort into having FSMLang support GNU Make in a natural way.

First, the file, fsmrules.mk, found in the root of the development tree, provides all of the target rules needed for creating the output files supported by FSMLang. The file also contains rules specifically to support the testing tree. These will not be discussed at present.

Starting at the top, we have the additional suffixes supported and a definition for our FSM macro, should one not have been provided.

```
##########################################################
#
#  .fsm rules
#

.SUFFIXES: .fsm .html .plantuml

FSM ?= fsm
```

Cleanliness is important; this next material identifies all of the output generated by FSMLang so that the cleanfsm target can remove it.

```
GENERATED_SRC = $(shell $(FSM) -M $(FSM_FLAGS) $(FSM_SRC))
GENERATED_HDR = $(shell $(FSM) -Mh $(FSM_FLAGS) $(FSM_SRC))

cleanfsm:
        @-rm -f $(GENERATED_SRC) 2> /dev/null
        @-rm -f $(GENERATED_HDR) 2> /dev/null
        @-rm -f *.fsmd           2> /dev/null
```

Next, Make was created for project managment, which has, as a key point, dependency tracking. So, fsmrules.mk provides a way to make dependency rules for files created by FSMLang.

```
%.fsmd: %.fsm
        @set -e; $(FSM) -Md $(FSM_FLAGS) $< > $@

ifneq ($(MAKECMDGOALS),clean)
-include $(FSM_SRC:.fsm=.fsmd)
endif
```

The dependencies (.fsmd) created here, and the generated file lists above, are C-output specific, but a similar rule and variables could be set up for html and plantuml output.

The suffix rules are pretty standard; though perhaps they should be adapted to more modern usage.

```
.fsm.o:
        @$(FSM) $(FSM_FLAGS) $< > fsmout
        @$(CC) -c $(CFLAGS) $*.c
        @rm -f $*.c

.fsm.c:
        @$(FSM) $(FSM_FLAGS) $< > fsmout

.fsm.h:
        @$(FSM) $(FSM_FLAGS) $< > fsmout

.fsm.html:
        @$(FSM) $(FSM_HTML_FLAGS) -th $< > fsmout

.fsm.plantuml:
        @$(FSM) $(FSM_PLANTUML_FLAGS) -tp $< > fsmout

.fsm.rst:
        @$(FSM) $(FSM_RST_FLAGS) -tr $< > fsmout
```

These rules assume the including Makefile has defined appropriate make variables giving any desired command line options.

Separate flags are expected for each type of FSMLang output:

- FSM_FLAGS for C output (FLAGS *must* contain -tc, -ts, -te or no -t option)
- FSM_HTML_FLAGS for HTML output (the FLAGS do *not* need to contain -th)
- FSM_PLANTUML_FLAGS for PlantUML output (the FLAGS do *not* need to contain -tp)
- FSM_RST_FLAGS for ReStructuredText output (the FLAGS do *not* need to contain -tr)

Of course, supporting files also need to be included in the build. By adopting a naming convention for action files (in the FSMLang development tree, all such file names contain '-actions'), the including Makefile can easily provide the full list of object files needed.

The simpleCommunicator Makefile

To put this all together, we'll build and test the simple communicator.

The Makefile begins by supplying the pieces needed by the material above:

```
##############################################################
#
#  Makefile for the simpleCommunicator FSMLang example
#

SRC = $(wildcard *actions.c)
FSM_SRC = $(wildcard *.fsm)

FSM_FLAGS=-tc --generate-weak-fns=false

CFLAGS=-DSIMPLE_COMMUNICATOR_DEBUG
```

Then, after including the C-language relevant portions of the above material, we have the stuff specific to our test project:

```
OBJS=$(SRC:.c=.o) $(GENERATED_SRC:.c=.o)

all: simpleCommunicator

test.out: simpleCommunicator
        ./$< >$@

simpleCommunicator: $(OBJS) $(FSM) Makefile
        $(CC) -o $@ $(LDFLAGS) $(OBJS)

clean: cleanfsm
        @-rm -f $(OBJS)            2> /dev/null
        @-rm -f simpleCommunicator 2> /dev/null
        @-rm -f test.out           2> /dev/null

$(SRC): simpleCommunicator_priv.h
```

The action functions we will use are "strong" versions of the weak functions which FSMLang would have created:

```
SIMPLE_COMMUNICATOR_EVENT UFMN(sendMessage)(FSM_TYPE_PTR pfsm)
{
   DBG_PRINTF("%s", __func__);
   (void) pfsm;
   return THIS(noEvent);
}

SIMPLE_COMMUNICATOR_EVENT UFMN(queueMessage)(FSM_TYPE_PTR pfsm)
{
   DBG_PRINTF("%s", __func__);
   (void) pfsm;
   return THIS(noEvent);
```

```
}

SIMPLE_COMMUNICATOR_EVENT UFMN(checkQueue)(FSM_TYPE_PTR pfsm)
{
   DBG_PRINTF("%s", __func__);
   (void) pfsm;
   return THIS(noEvent);
}

SIMPLE_COMMUNICATOR_EVENT UFMN(noAction)(FSM_TYPE_PTR pfsm)
{
   DBG_PRINTF("%s", __func__);
   (void) pfsm;
   return THIS(noEvent);
}
```

And, our “test” routine simply calls our machine with its two events:

```
int main(void)
{

   run_simpleCommunicator(THIS(SEND_MESSAGE));
   run_simpleCommunicator(THIS(ACK));

   return 0;
}
```

Executing

```
make test.out
```

creates:

```
event: simpleCommunicator_SEND_MESSAGE; state: simpleCommunicator_IDLE
simpleCommunicator_sendMessage
event: simpleCommunicator_ACK; state: simpleCommunicator_AWAITING_ACK
simpleCommunicator_checkQueue
```

The first and third lines of the output are generated from the built-in debug messaging; the second and fourth are the output of the action functions. Inspecting this output, we find that it is what is expected from the event sequence executed.

Sources

simpleCommunicator.fsm

Download

```
native
{
#include <stdio.h>
#define DBG_PRINTF(...) printf(__VA_ARGS__); printf("\n")
}

/**
   This machine manages communications using a "stop and wait" protocol.
```

```
       Only one message is allowed to be outstanding.
 */
 machine simpleCommunicator {

    state   IDLE,
            AWAITING_ACK;

    event   SEND_MESSAGE,
            NEVER_SEEN,
            ACK;

    action neverExecuted[NEVER_SEEN, (IDLE, AWAITING_ACK)];

   /**
     Since we're idle, we can simply send the message.  Transitioning
     to the AWAITING_ACK state ensures that any other messages
     we're asked to send will be queued.
   */
    action   sendMessage[SEND_MESSAGE,IDLE] transition AWAITING_ACK;

   /**
     Since we're busy, we must queue the message for later
     sending.  The queue will be checked when the ACK
     is received.
   */
    action   queueMessage[SEND_MESSAGE,AWAITING_ACK];

   /**
     We've received our ACK for the previous message.  It is
     time to check for any others.
   */
    action   checkQueue[ACK,AWAITING_ACK] transition IDLE;

   /* these lines are informational; they affect the html output,
       but do not affect any code generated.
   */

   /** queueMessage adds a message to the queue */
   queueMessage returns noEvent;

   /** sendMessage sends a message from the queue.  The
       message is expected to be there, since
       checkQueue will have been previously called.
   */
   sendMessage  returns noEvent;

   /** checkQueue only checks; it does not dequeue; that
       is done by sendMessage.

       Return SEND_MESSAGE when the queue is not empty.
   */
   checkQueue   returns SEND_MESSAGE, noEvent;
```

```
}
```

Makefile

Download

```
#################################################################
#
#  Makefile for the simpleCommunicator FSMLang example
#

SRC = $(wildcard *actions.c)
FSM_SRC = $(wildcard *.fsm)

FSM_FLAGS=-tc --generate-weak-fns=false

CFLAGS=-DSIMPLE_COMMUNICATOR_DEBUG

## FSMLang start

.SUFFIXES: .fsm .html .plantuml

FSM ?= fsm

GENERATED_SRC = $(shell $(FSM) -M $(FSM_FLAGS) $(FSM_SRC))
GENERATED_HDR = $(shell $(FSM) -Mh $(FSM_FLAGS) $(FSM_SRC))

cleanfsm:
        @-rm -f $(GENERATED_SRC) 2> /dev/null
        @-rm -f $(GENERATED_HDR) 2> /dev/null
        @-rm -f *.fsmd           2> /dev/null

%.fsmd: %.fsm
        @set -e; $(FSM) -Md $(FSM_FLAGS) $< > $@

.fsm.o:
        @$(FSM) $(FSM_FLAGS) $< > fsmout
        @$(CC) -c $(CFLAGS) $*.c
        @rm -f $*.c

.fsm.c:
        @$(FSM) $(FSM_FLAGS) $< > fsmout

.fsm.h:
        @$(FSM) $(FSM_FLAGS) $< > fsmout

ifneq ($(MAKECMDGOALS),clean)
-include $(FSM_SRC:.fsm=.fsmd)
endif

## FSMLang end
```

```
OBJS=$(SRC:.c=.o) $(GENERATED_SRC:.c=.o)

all: simpleCommunicator

test.out: simpleCommunicator
        ./$< >$@

simpleCommunicator: $(OBJS) $(FSM) Makefile
        $(CC) -o $@ $(LDFLAGS) $(OBJS)

clean: cleanfsm
        @-rm -f $(OBJS)            2> /dev/null
        @-rm -f simpleCommunicator 2> /dev/null
        @-rm -f test.out           2> /dev/null

$(SRC): simpleCommunicator_priv.h
```

sc-actions.c

Download

```
#include "simpleCommunicator_priv.h"

SIMPLE_COMMUNICATOR_EVENT UFMN(sendMessage)(FSM_TYPE_PTR pfsm)
{
        DBG_PRINTF("%s", __func__);
        (void) pfsm;
        return THIS(noEvent);
}

SIMPLE_COMMUNICATOR_EVENT UFMN(queueMessage)(FSM_TYPE_PTR pfsm)
{
        DBG_PRINTF("%s", __func__);
        (void) pfsm;
        return THIS(noEvent);
}

SIMPLE_COMMUNICATOR_EVENT UFMN(checkQueue)(FSM_TYPE_PTR pfsm)
{
        DBG_PRINTF("%s", __func__);
        (void) pfsm;
        return THIS(noEvent);
}

SIMPLE_COMMUNICATOR_EVENT UFMN(noAction)(FSM_TYPE_PTR pfsm)
{
        DBG_PRINTF("%s", __func__);
        (void) pfsm;
        return THIS(noEvent);
}


int main(void)
```

```
{

        run_simpleCommunicator(THIS(SEND_MESSAGE));
        run_simpleCommunicator(THIS(ACK));

        return 0;
}
```

Action Array Sources

These sources are generated using `fsm -tc simpleCommunicator.fsm` or `fsm simpleCommunicator`, since `-tc` is the default.

simpleCommunicator.c

Generated using `fsm simpleCommunicator.fsm`

```
/**
        simpleCommunicator.c

        This file automatically generated by FSMLang
*/

#include "simpleCommunicator_priv.h"
#include <stddef.h>


#ifndef DBG_PRINTF
#define DBG_PRINTF(...)
#endif

static const SIMPLE_COMMUNICATOR_ACTION_TRANS simpleCommunicator_action_array[THIS(numEvents)][simpleCommunicator_numStates] =
{
    {
        /* -- SEND_MESSAGE -- */

        /* -- IDLE -- */        {UFMN(sendMessage),simpleCommunicator_AWAITING_ACK}
        /* -- AWAITING_ACK -- */        , {UFMN(queueMessage),simpleCommunicator_AWAITING_ACK}
    },
    {
        /* -- ACK -- */

        /* -- IDLE -- */        {UFMN(noAction), simpleCommunicator_IDLE}
        /* -- AWAITING_ACK -- */        , {UFMN(checkQueue),simpleCommunicator_IDLE}
    },
};
SIMPLE_COMMUNICATOR simpleCommunicator = {
    simpleCommunicator_IDLE,
    THIS(SEND_MESSAGE),
    &simpleCommunicator_action_array,
    simpleCommunicatorFSM
};

pSIMPLE_COMMUNICATOR psimpleCommunicator = &simpleCommunicator;


void run_simpleCommunicator(SIMPLE_COMMUNICATOR_EVENT e)
{
     if (psimpleCommunicator)
     {
          psimpleCommunicator->fsm(psimpleCommunicator,e);
     }
}

#ifndef EVENT_IS_NOT_EXCLUDED_FROM_LOG
```

```
#define EVENT_IS_NOT_EXCLUDED_FROM_LOG(e) ((e) == (e))
#endif
void simpleCommunicatorFSM(pSIMPLE_COMMUNICATOR pfsm, SIMPLE_COMMUNICATOR_EVENT event)
{
    SIMPLE_COMMUNICATOR_EVENT new_e;
    SIMPLE_COMMUNICATOR_EVENT e = event;
    while (e != THIS(noEvent)) {

#ifdef SIMPLE_COMMUNICATOR_DEBUG
        if (EVENT_IS_NOT_EXCLUDED_FROM_LOG(e))
        {
            DBG_PRINTF("event: %s; state: %s"
                       ,SIMPLE_COMMUNICATOR_EVENT_NAMES[e]
                       ,SIMPLE_COMMUNICATOR_STATE_NAMES[pfsm->state]
                      );
        }
#endif

        /* This is read-only data to facilitate error reporting in action functions */
        pfsm->event = e;

        new_e = ((* (*pfsm->actionArray)[e][pfsm->state].action)(pfsm));

        pfsm->state = (*pfsm->actionArray)[e][pfsm->state].transition;

        e = new_e;

    }

}

SIMPLE_COMMUNICATOR_EVENT __attribute__((weak)) UFMN(sendMessage)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("weak: %s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}

SIMPLE_COMMUNICATOR_EVENT __attribute__((weak)) UFMN(queueMessage)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("weak: %s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}

SIMPLE_COMMUNICATOR_EVENT __attribute__((weak)) UFMN(checkQueue)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("weak: %s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}

SIMPLE_COMMUNICATOR_EVENT __attribute__((weak)) UFMN(noAction)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("weak: %s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}


#ifdef SIMPLE_COMMUNICATOR_DEBUG
char *SIMPLE_COMMUNICATOR_EVENT_NAMES[] = {
    "simpleCommunicator_SEND_MESSAGE"
    ,"simpleCommunicator_ACK"
    , "simpleCommunicator_noEvent"
    , "simpleCommunicator_numEvents"
};

char *SIMPLE_COMMUNICATOR_STATE_NAMES[] = {
    "simpleCommunicator_IDLE"
    ,"simpleCommunicator_AWAITING_ACK"
};

#endif
```

simpleCommunicator.h

Generated using `fsm simpleCommunicator.fsm`

```
/**
        simpleCommunicator.h

        This file automatically generated by FSMLang
*/

#ifndef _SIMPLECOMMUNICATOR_H_
#define _SIMPLECOMMUNICATOR_H_

#include "simpleCommunicator_events.h"
#define FSM_VERSION "1.45.1"

#ifndef NO_CONVENIENCE_MACROS
#undef UFMN
#define UFMN(A) simpleCommunicator_##A
#undef THIS
#define THIS(A) simpleCommunicator_##A
#endif
#undef STATE
#define STATE(A) simpleCommunicator_##A

#undef ACTION_RETURN_TYPE
#define ACTION_RETURN_TYPE SIMPLE_COMMUNICATOR_EVENT

void run_simpleCommunicator(SIMPLE_COMMUNICATOR_EVENT);

typedef struct _simpleCommunicator_struct_ *pSIMPLE_COMMUNICATOR;
extern pSIMPLE_COMMUNICATOR psimpleCommunicator;


#endif
```

simpleCommunicator_events.h

Generated using `fsm simpleCommunicator.fsm`

```
/**
        simpleCommunicator_events.h

        This file automatically generated by FSMLang
*/

#ifndef _SIMPLECOMMUNICATOR_EVENTS_H_
#define _SIMPLECOMMUNICATOR_EVENTS_H_

typedef enum SIMPLE_COMMUNICATOR_EVENT {
    simpleCommunicator_SEND_MESSAGE
    , simpleCommunicator_ACK
    , simpleCommunicator_noEvent
    , simpleCommunicator_numEvents
    , simpleCommunicator_numAllEvents = simpleCommunicator_numEvents
```

```
}  SIMPLE_COMMUNICATOR_EVENT;


#endif
```

simpleCommunicator_priv.h

Generated using `fsm simpleCommunicator.fsm`

```
/**
        simpleCommunicator_priv.h

        This file automatically generated by FSMLang
*/

#ifndef _SIMPLECOMMUNICATOR_PRIV_H_
#define _SIMPLECOMMUNICATOR_PRIV_H_

#include "simpleCommunicator.h"

#ifdef SIMPLE_COMMUNICATOR_DEBUG
#include <stdio.h>
#include <stdlib.h>
#endif


#ifdef SIMPLE_COMMUNICATOR_DEBUG
extern char *SIMPLE_COMMUNICATOR_EVENT_NAMES[];
extern char *SIMPLE_COMMUNICATOR_STATE_NAMES[];
#endif

typedef enum {
    simpleCommunicator_IDLE
    , simpleCommunicator_AWAITING_ACK
    , simpleCommunicator_numStates
}  SIMPLE_COMMUNICATOR_STATE;

typedef struct _simpleCommunicator_struct_ SIMPLE_COMMUNICATOR;
#undef FSM_TYPE_PTR
#define FSM_TYPE_PTR pSIMPLE_COMMUNICATOR
extern SIMPLE_COMMUNICATOR simpleCommunicator;

typedef SIMPLE_COMMUNICATOR_EVENT (*SIMPLE_COMMUNICATOR_ACTION_FN)(FSM_TYPE_PTR);

typedef void (*SIMPLE_COMMUNICATOR_FSM)(FSM_TYPE_PTR,SIMPLE_COMMUNICATOR_EVENT);

void simpleCommunicatorFSM(FSM_TYPE_PTR,SIMPLE_COMMUNICATOR_EVENT);

typedef struct _simpleCommunicator_action_trans_struct_ {
    SIMPLE_COMMUNICATOR_ACTION_FN       action;
    SIMPLE_COMMUNICATOR_STATE   transition;
} SIMPLE_COMMUNICATOR_ACTION_TRANS, *pSIMPLE_COMMUNICATOR_ACTION_TRANS;

struct _simpleCommunicator_struct_ {
```

```
    SIMPLE_COMMUNICATOR_STATE          state;
    SIMPLE_COMMUNICATOR_EVENT          event;
    SIMPLE_COMMUNICATOR_ACTION_TRANS   const    (*actionArray)[THIS(numEvents)][simpleCommunicator_numStates];
    SIMPLE_COMMUNICATOR_FSM           fsm;
};

ACTION_RETURN_TYPE simpleCommunicator_sendMessage(FSM_TYPE_PTR);
ACTION_RETURN_TYPE simpleCommunicator_queueMessage(FSM_TYPE_PTR);
ACTION_RETURN_TYPE simpleCommunicator_checkQueue(FSM_TYPE_PTR);
ACTION_RETURN_TYPE simpleCommunicator_noAction(FSM_TYPE_PTR);



#endif
```

State Function Array Sources

These sources are generated using `fsm -ts simpleCommunicator.fsm`.

simpleCommunicator.c

Generated using `fsm -ts simpleCommunicator.fsm`

```
/**
        simpleCommunicator.c

        This file automatically generated by FSMLang
*/

#include "simpleCommunicator_priv.h"
#include <stddef.h>


#ifndef DBG_PRINTF
#define DBG_PRINTF(...)
#endif

static SIMPLE_COMMUNICATOR_EVENT IDLE_stateFn(pSIMPLE_COMMUNICATOR,SIMPLE_COMMUNICATOR_EVENT);
static SIMPLE_COMMUNICATOR_EVENT AWAITING_ACK_stateFn(pSIMPLE_COMMUNICATOR,SIMPLE_COMMUNICATOR_EVENT);
static const SIMPLE_COMMUNICATOR_STATE_FN simpleCommunicator_state_fn_array[simpleCommunicator_numStates] =
{
    IDLE_stateFn
    , AWAITING_ACK_stateFn
};

SIMPLE_COMMUNICATOR simpleCommunicator = {
    simpleCommunicator_IDLE,
    THIS(SEND_MESSAGE),
    &simpleCommunicator_state_fn_array,
    simpleCommunicatorFSM
};

pSIMPLE_COMMUNICATOR psimpleCommunicator = &simpleCommunicator;

void run_simpleCommunicator(SIMPLE_COMMUNICATOR_EVENT e)
{
    if (psimpleCommunicator)


     {
          psimpleCommunicator->fsm(psimpleCommunicator,e);
     }
}
```

```
#ifndef EVENT_IS_NOT_EXCLUDED_FROM_LOG
#define EVENT_IS_NOT_EXCLUDED_FROM_LOG(e) ((e) == (e))
#endif
void simpleCommunicatorFSM(pSIMPLE_COMMUNICATOR pfsm, SIMPLE_COMMUNICATOR_EVENT event)
{
    SIMPLE_COMMUNICATOR_EVENT e = event;

    while (e != simpleCommunicator_noEvent) {

#ifdef SIMPLE_COMMUNICATOR_DEBUG
        if (EVENT_IS_NOT_EXCLUDED_FROM_LOG(e))
        {
            DBG_PRINTF("event: %s; state: %s"
                       ,SIMPLE_COMMUNICATOR_EVENT_NAMES[e]
                       ,SIMPLE_COMMUNICATOR_STATE_NAMES[pfsm->state]
                      );
        }
#endif

        /* This is read-only data to facilitate error reporting in action functions */
        pfsm->event = e;

        e = ((* (*pfsm->statesArray)[pfsm->state])(pfsm,e));
    }

}

static SIMPLE_COMMUNICATOR_EVENT IDLE_stateFn(pSIMPLE_COMMUNICATOR pfsm,SIMPLE_COMMUNICATOR_EVENT e)
{
    SIMPLE_COMMUNICATOR_EVENT retVal = THIS(noEvent);

    switch(e)
    {
    case THIS(SEND_MESSAGE):
        retVal = UFMN(sendMessage)(pfsm);
        pfsm->state = simpleCommunicator_AWAITING_ACK;
        break;
    default:
        DBG_PRINTF("simpleCommunicator_noAction");
        break;
    }

    return retVal;
}

static SIMPLE_COMMUNICATOR_EVENT AWAITING_ACK_stateFn(pSIMPLE_COMMUNICATOR pfsm,SIMPLE_COMMUNICATOR_EVENT e)
{
    SIMPLE_COMMUNICATOR_EVENT retVal = THIS(noEvent);

    switch(e)
    {
    case THIS(SEND_MESSAGE):
        retVal = UFMN(queueMessage)(pfsm);

        break;
     case THIS(ACK):
        retVal = UFMN(checkQueue)(pfsm);
        pfsm->state = simpleCommunicator_IDLE;
        break;
     default:
        DBG_PRINTF("simpleCommunicator_noAction");
        break;
     }
```

```
    return retVal;
}


SIMPLE_COMMUNICATOR_EVENT __attribute__((weak)) UFMN(sendMessage)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("weak: %s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}

SIMPLE_COMMUNICATOR_EVENT __attribute__((weak)) UFMN(queueMessage)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("weak: %s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}

SIMPLE_COMMUNICATOR_EVENT __attribute__((weak)) UFMN(checkQueue)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("weak: %s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}


#ifdef SIMPLE_COMMUNICATOR_DEBUG
char *SIMPLE_COMMUNICATOR_EVENT_NAMES[] = {
    "simpleCommunicator_SEND_MESSAGE"
    ,"simpleCommunicator_ACK"
    , "simpleCommunicator_noEvent"
    , "simpleCommunicator_numEvents"
};

char *SIMPLE_COMMUNICATOR_STATE_NAMES[] = {
    "simpleCommunicator_IDLE"
    ,"simpleCommunicator_AWAITING_ACK"
};

#endif
```

simpleCommunicator.h

Generated using `fsm -ts simpleCommunicator.fsm`

```
/**
        simpleCommunicator.h

        This file automatically generated by FSMLang
*/

#ifndef _SIMPLECOMMUNICATOR_H_
#define _SIMPLECOMMUNICATOR_H_
```

```
#include "simpleCommunicator_events.h"
#define FSM_VERSION "1.45.1"

#ifndef NO_CONVENIENCE_MACROS
#undef UFMN
#define UFMN(A) simpleCommunicator_##A
#undef THIS
#define THIS(A) simpleCommunicator_##A
#endif
#undef STATE
#define STATE(A) simpleCommunicator_##A

#undef ACTION_RETURN_TYPE
#define ACTION_RETURN_TYPE SIMPLE_COMMUNICATOR_EVENT

void run_simpleCommunicator(SIMPLE_COMMUNICATOR_EVENT);

typedef struct _simpleCommunicator_struct_ *pSIMPLE_COMMUNICATOR;
extern pSIMPLE_COMMUNICATOR psimpleCommunicator;


#endif
```

simpleCommunicator_events.h

Generated using `fsm -ts simpleCommunicator.fsm`

```
/**
        simpleCommunicator_events.h

        This file automatically generated by FSMLang
*/

#ifndef _SIMPLECOMMUNICATOR_EVENTS_H_
#define _SIMPLECOMMUNICATOR_EVENTS_H_

typedef enum SIMPLE_COMMUNICATOR_EVENT {
    simpleCommunicator_SEND_MESSAGE
    , simpleCommunicator_ACK
    , simpleCommunicator_noEvent
    , simpleCommunicator_numEvents
    , simpleCommunicator_numAllEvents = simpleCommunicator_numEvents
}  SIMPLE_COMMUNICATOR_EVENT;


#endif
```

simpleCommunicator_priv.h

Generated using `fsm -ts simpleCommunicator.fsm`

```
/**
        simpleCommunicator_priv.h
```

The Simple Communicator

```
        This file automatically generated by FSMLang
*/

#ifndef _SIMPLECOMMUNICATOR_PRIV_H_
#define _SIMPLECOMMUNICATOR_PRIV_H_

#include "simpleCommunicator.h"

#ifdef SIMPLE_COMMUNICATOR_DEBUG
#include <stdio.h>
#include <stdlib.h>
#endif


#ifdef SIMPLE_COMMUNICATOR_DEBUG
extern char *SIMPLE_COMMUNICATOR_EVENT_NAMES[];
extern char *SIMPLE_COMMUNICATOR_STATE_NAMES[];
#endif

typedef enum {
    simpleCommunicator_IDLE
    , simpleCommunicator_AWAITING_ACK
    , simpleCommunicator_numStates
}  SIMPLE_COMMUNICATOR_STATE;

typedef struct _simpleCommunicator_struct_ SIMPLE_COMMUNICATOR;
#undef FSM_TYPE_PTR
#define FSM_TYPE_PTR pSIMPLE_COMMUNICATOR
extern SIMPLE_COMMUNICATOR simpleCommunicator;

typedef SIMPLE_COMMUNICATOR_EVENT (*SIMPLE_COMMUNICATOR_ACTION_FN)(FSM_TYPE_PTR);

typedef void (*SIMPLE_COMMUNICATOR_FSM)(FSM_TYPE_PTR,SIMPLE_COMMUNICATOR_EVENT);

void simpleCommunicatorFSM(FSM_TYPE_PTR,SIMPLE_COMMUNICATOR_EVENT);

typedef ACTION_RETURN_TYPE (*SIMPLE_COMMUNICATOR_STATE_FN)(pSIMPLE_COMMUNICATOR,SIMPLE_COMMUNICATOR_EVENT);

static const SIMPLE_COMMUNICATOR_STATE_FN simpleCommunicator_state_fn_array[simpleCommunicator_numStates];

struct _simpleCommunicator_struct_ {
    SIMPLE_COMMUNICATOR_STATE            state;
    SIMPLE_COMMUNICATOR_EVENT            event;
    SIMPLE_COMMUNICATOR_STATE_FN   const (*statesArray)[simpleCommunicator_numStates];
    SIMPLE_COMMUNICATOR_FSM              fsm;
};


ACTION_RETURN_TYPE simpleCommunicator_sendMessage(FSM_TYPE_PTR);
ACTION_RETURN_TYPE simpleCommunicator_queueMessage(FSM_TYPE_PTR);
ACTION_RETURN_TYPE simpleCommunicator_checkQueue(FSM_TYPE_PTR);



#endif
```

Event Function Array Sources

These sources are generated using `fsm -te simpleCommunicator.fsm`.

simpleCommunicator.c

Generated using `fsm -te simpleCommunicator.fsm`

```
/**
        simpleCommunicator.c

        This file automatically generated by FSMLang
*/

#include "simpleCommunicator_priv.h"
#include <stddef.h>


#ifndef DBG_PRINTF
#define DBG_PRINTF(...)
#endif

static ACTION_RETURN_TYPE simpleCommunicator_handle_SEND_MESSAGE(FSM_TYPE_PTR);
static ACTION_RETURN_TYPE simpleCommunicator_handle_ACK(FSM_TYPE_PTR);

static const SIMPLE_COMMUNICATOR_ACTION_FN simpleCommunicator_event_fn_array[simpleCommunicator_numMachineEvents] =
{
    simpleCommunicator_handle_SEND_MESSAGE
    , simpleCommunicator_handle_ACK
};

SIMPLE_COMMUNICATOR simpleCommunicator = {
    simpleCommunicator_IDLE,
    THIS(SEND_MESSAGE),
    &simpleCommunicator_event_fn_array,
    simpleCommunicatorFSM
};

pSIMPLE_COMMUNICATOR psimpleCommunicator = &simpleCommunicator;

void run_simpleCommunicator(SIMPLE_COMMUNICATOR_EVENT e)
{
    if (psimpleCommunicator)
    {
        psimpleCommunicator->fsm(psimpleCommunicator,e);
    }
}


#ifndef EVENT_IS_NOT_EXCLUDED_FROM_LOG
#define EVENT_IS_NOT_EXCLUDED_FROM_LOG(e) ((e) == (e))
#endif
void simpleCommunicatorFSM(pSIMPLE_COMMUNICATOR pfsm, SIMPLE_COMMUNICATOR_EVENT event)
{
    SIMPLE_COMMUNICATOR_EVENT e = event;

    while (e != simpleCommunicator_noEvent) {

#ifdef SIMPLE_COMMUNICATOR_DEBUG
        if (EVENT_IS_NOT_EXCLUDED_FROM_LOG(e))
        {

            DBG_PRINTF("event: %s; state: %s"
                       ,SIMPLE_COMMUNICATOR_EVENT_NAMES[e]
                       ,SIMPLE_COMMUNICATOR_STATE_NAMES[pfsm->state]
                      );
        }
#endif

        /* This is read-only data to facilitate error reporting in action functions */
        pfsm->event = e;

        e = ((* (*pfsm->eventsArray)[e])(pfsm));
    }


}
```

```
static ACTION_RETURN_TYPE simpleCommunicator_handle_SEND_MESSAGE(FSM_TYPE_PTR pfsm)
{
    ACTION_RETURN_TYPE event = THIS(noEvent);

    switch (pfsm->state)
    {
    case STATE(IDLE):
        event = UFMN(sendMessage)(pfsm);
        pfsm->state = STATE(AWAITING_ACK);
        break;
    case STATE(AWAITING_ACK):
        event = UFMN(queueMessage)(pfsm);
        break;
    }

    return event;
}

static ACTION_RETURN_TYPE simpleCommunicator_handle_ACK(FSM_TYPE_PTR pfsm)
{
    ACTION_RETURN_TYPE event = THIS(noEvent);

    switch (pfsm->state)
    {
    case STATE(AWAITING_ACK):
        event = UFMN(checkQueue)(pfsm);
        pfsm->state = STATE(IDLE);
        break;
    default:
        DBG_PRINTF("simpleCommunicator_noAction");
        break;
    }

    return event;
}

SIMPLE_COMMUNICATOR_EVENT __attribute__((weak)) UFMN(sendMessage)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("weak: %s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}

SIMPLE_COMMUNICATOR_EVENT __attribute__((weak)) UFMN(queueMessage)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("weak: %s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}

SIMPLE_COMMUNICATOR_EVENT __attribute__((weak)) UFMN(checkQueue)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("weak: %s", __func__);
```

```
    (void) pfsm;
    return THIS(noEvent);
}


#ifdef SIMPLE_COMMUNICATOR_DEBUG
char *SIMPLE_COMMUNICATOR_EVENT_NAMES[] = {
    "simpleCommunicator_SEND_MESSAGE"
    ,"simpleCommunicator_ACK"
    , "simpleCommunicator_noEvent"
    , "simpleCommunicator_numEvents"
};

char *SIMPLE_COMMUNICATOR_STATE_NAMES[] = {
    "simpleCommunicator_IDLE"
    ,"simpleCommunicator_AWAITING_ACK"
};

#endif
```

simpleCommunicator.h

Generated using `fsm -te simpleCommunicator.fsm`

```
/**
        simpleCommunicator.h

        This file automatically generated by FSMLang
*/

#ifndef _SIMPLECOMMUNICATOR_H_
#define _SIMPLECOMMUNICATOR_H_

#include "simpleCommunicator_events.h"
#define FSM_VERSION "1.45.1"

#ifndef NO_CONVENIENCE_MACROS
#undef UFMN
#define UFMN(A) simpleCommunicator_##A
#undef THIS
#define THIS(A) simpleCommunicator_##A
#endif
#undef STATE
#define STATE(A) simpleCommunicator_##A

#undef ACTION_RETURN_TYPE
#define ACTION_RETURN_TYPE SIMPLE_COMMUNICATOR_EVENT

void run_simpleCommunicator(SIMPLE_COMMUNICATOR_EVENT);

typedef struct _simpleCommunicator_struct_ *pSIMPLE_COMMUNICATOR;
extern pSIMPLE_COMMUNICATOR psimpleCommunicator;
```

```
#endif
```

simpleCommunicator_events.h

Generated using `fsm -te simpleCommunicator.fsm`

```
/**
        simpleCommunicator_events.h

        This file automatically generated by FSMLang
*/

#ifndef _SIMPLECOMMUNICATOR_EVENTS_H_
#define _SIMPLECOMMUNICATOR_EVENTS_H_

typedef enum SIMPLE_COMMUNICATOR_EVENT {
    simpleCommunicator_SEND_MESSAGE
    , simpleCommunicator_ACK
    , simpleCommunicator_noEvent
    , simpleCommunicator_numEvents
    , simpleCommunicator_numAllEvents = simpleCommunicator_numEvents
}  SIMPLE_COMMUNICATOR_EVENT;


#endif
```

simpleCommunicator_priv.h

Generated using `fsm -te simpleCommunicator.fsm`

```
/**
        simpleCommunicator_priv.h

        This file automatically generated by FSMLang
*/

#ifndef _SIMPLECOMMUNICATOR_PRIV_H_
#define _SIMPLECOMMUNICATOR_PRIV_H_

#include "simpleCommunicator.h"

#ifdef SIMPLE_COMMUNICATOR_DEBUG
#include <stdio.h>
#include <stdlib.h>
#endif


#ifdef SIMPLE_COMMUNICATOR_DEBUG
extern char *SIMPLE_COMMUNICATOR_EVENT_NAMES[];
extern char *SIMPLE_COMMUNICATOR_STATE_NAMES[];
#endif

typedef enum {
```

```
    simpleCommunicator_IDLE
    , simpleCommunicator_AWAITING_ACK
}  SIMPLE_COMMUNICATOR_STATE;

typedef struct _simpleCommunicator_struct_ SIMPLE_COMMUNICATOR;
#undef FSM_TYPE_PTR
#define FSM_TYPE_PTR pSIMPLE_COMMUNICATOR
extern SIMPLE_COMMUNICATOR simpleCommunicator;

typedef SIMPLE_COMMUNICATOR_EVENT (*SIMPLE_COMMUNICATOR_ACTION_FN)(FSM_TYPE_PTR);

typedef void (*SIMPLE_COMMUNICATOR_FSM)(FSM_TYPE_PTR,SIMPLE_COMMUNICATOR_EVENT);

void simpleCommunicatorFSM(FSM_TYPE_PTR,SIMPLE_COMMUNICATOR_EVENT);

#define simpleCommunicator_numMachineEvents 2
struct _simpleCommunicator_struct_ {
    SIMPLE_COMMUNICATOR_STATE         state;
    SIMPLE_COMMUNICATOR_EVENT         event;
    SIMPLE_COMMUNICATOR_ACTION_FN const (*eventsArray)[simpleCommunicator_numMachineEvents];
    SIMPLE_COMMUNICATOR_FSM           fsm;
};

ACTION_RETURN_TYPE simpleCommunicator_sendMessage(FSM_TYPE_PTR);
ACTION_RETURN_TYPE simpleCommunicator_queueMessage(FSM_TYPE_PTR);
ACTION_RETURN_TYPE simpleCommunicator_checkQueue(FSM_TYPE_PTR);



#endif
```

The HSM Communicator

The HSM Communicator illustrates the use of hierarchical state machines (HSMs).

Hierarchical State Machines are indicated whenever the process to be managed involves loosely connected, highly coherent, sub-processes. Our communications example here is one such situation. Another might be the management of an external FLASH part through a QSPI interface.

Real-life Example: Design Re-use

A driver for an internal FLASH controller was developed using FSMLang. The driver allowed writes to the flash from different parts of the code to be queued for sequential operations. When it was desired to write to external FLASH as well, a sub-machine to manage the QSPI interaction was added to the existing state machine, with minimal adaption of the existing logic.

To be most useful, sub-machines should be thought of as sub-routines. Just as sub-routines might be created for “readability,” a sub-machine might clarify an FSM design by encapsulating events which always go together. And, just as with sub-routines, a sub-machine might encapsulate a process that must be repeated in response to an event, regardless of what else might have been going on at the time.

Here, one thinks again, of a communications example: establishing a Bluetooth Low Energy (BLE) connection in a noisy environment, when that connection involves some “higer-level” sequence of steps. A sub-machine that knows how to recover from a signal loss at any time can help the higher-level machine continue through the “higher-level” sequence essentially without regard to the signal loss.

The HSM Communicator

As stated in the hsmCommunicator.fsm document, the design goal is a machine that is not only able to send messages to a peer using a stop and wait protocol, in which only one message may be outstanding, but which also implements the additional constraint that no message may be sent until a session has been established. Session establishment requires two messages to be sent and two to be received. Sessions expire after some amount of inactivity.

The events such a machine would experience might be conceived in this way:

- A request to send a message. This is an event from a local client program.
- The sending of the first session establishment message to the peer communicator.
- The receipt of the first session establishment message from the peer communicator.
- The receipt of the second session establishment message from the peer communicator.
- The receipt of the message acknowledgement from the peer communicator.
- The receipt of a timer message from the local OS indicating that the session timeout has elapsed.

Of course, being a simple example, a flat state machine may be conceived which would handle these events. This machine would need only three states:

1. Nothing is happening.
2. Session establishment messages are outstanding.
3. A client message is outstanding.

The flat machine might start to look like this:

```
machine complexCommunicator
{

   event SEND_MESSAGE
         , MESSAGE_RECEIVED
         , STEP0_RESPONSE
         , STEP1_RESPONSE
         , ACK
         , TIMER_EXPIRED
         ;

   state IDLE
         , ESTABLISHING_SESSION
         , IN_SESSION
         , AWAITING_ACK
         ;

}
```

Note the addition of the MESSAGE_RECIEVED event. In our scenario, the machine is responsible for parsing the peer messages.

Since our goal is to illustrate a hierarchical machine, however, we’ll leave the flat machine behind.

One conception would be to have a top-level machine with two sub-machines. One sub-machine will handle the session establishment chore, while the other will send the client messages.

> **Note**
>
> FSMLang interchanges the use of terms parent/child with parent/sub-machine.
>
> The term *top-level* always refers to the machine of which all others are children (or grandchildren).

Skeleton

The skeleton looks like this:

```
machine hsmCommunicator
{

   machine establishSession
   {

   }

   machine sendMessage
   {

   }

}
```

Events

The next step is to allocate the events:

```
machine hsmCommunicator
{

   event SEND_MESSAGE
         , MESSAGE_RECEIVED
         , SESSION_ESTABLISHED
         , TIMER_EXPIRED
         ;

   machine establishSession
   {
      event ESTABLISH_SESSION
            , STEP0_RESPONSE
            , STEP1_RESPONSE
            , parent::MESSAGE_RECEIVED
            ;

   }

   machine sendMessage
   {
```

```
        event parent::SEND_MESSAGE
              , parent::MESSAGE_RECEIVED
              , ACK
              ;
    }

 }
```

The alert reader notes the addition of several events. Namely, `event ESTABLISH_SESSION` is added in the sub-machine establishSession; and `event SESSION_ESTABLISHED` is added in the top-level machine. In addition, the keyword and namespace indicator `parent::` are used to add this event to the sub-machines. These will be used to show the two ways that HSMs can communicate within the hierarchy. The top-level machine will send ESTABLISH_SESSION to the sub-machine establishSession only when a new session is needed. The sub-machine, establishSession will return SESSION_ESTABLISHED only upon session establishment. But, `event MESSAGE_RECEIVED`, on the other hand, will be shared down to the sub-machines whenever it occurs.

States

To finish our design, we add the states needed to track progress through the event stream.

All of the machines will start in the IDLE state.

When the top-level machine receives the SEND_MESSAGE event in the IDLE state, it must first establish a session before it can send the message. It begins the session establishment sequence by sending the ESTABLISH_SESSION event to the establishSession machine. The top-level machine also transitions to the ESTABLISHING_SESSION state in order to not interrupt the session establishment sequence, should a new SEND_MESSAGE event be received before the session is fully established. Then, when the establishSession machine has completed its work and has returned `event SESSION_ESTABLISHED`, the top-level machine will transition to the IN_SESSION state. This transition is again necessary to take the correct action on the next SEND_MESSAGE event received. It is in this state, too, that the TIMER_EXPIRED event might occur, which would cause the machine to return to the IDLE state.

Our machine now looks like this:

```
machine hsmCommunicator
{

    event SEND_MESSAGE
          , MESSAGE_RECEIVED
          , TIMER_EXPIRED
          ;

    state IDLE
          , ESTABLISHING_SESSION
          , IN_SESSION
          ;

    machine establishSession
    {
                event ESTABLISH_SESSION
             , STEP0_RESPONSE
             , STEP1_RESPONSE
             , parent::MESSAGE_RECEIVED
             ;
```

```
        state IDLE
              , AWAITING_RESPONSE
              ;

    }

    machine sendMessage
    {

        event parent::SEND_MESSAGE
              , parent::MESSAGE_RECEIVED
              , ACK
              ;

        state IDLE;

    }

}
```

Next, we take up the states for the establishSession machine.

The ESTABLISH_SESSION message is first received in the IDLE state. The machine will begin its work and move to the AWAITING_RESPONSE state. While in this state, responses from the peer are received. Since the “outside world” knows only about the hsmCommunicator, the MESSAGE_RECEIVED events are first seen there, then shared down to the children; each child machine parses the message, creating the approprate machine events from the message received. In the case of the establishSession machine, it is looking for the STEP0_RESPONSE and STEP1_RESPONSE messages. Assuming the peer will not send these responses out of order, establishSession may remain in the single AWAITING_RESPONSE state until both messages are received. Upon this, establishSession returns SESSION_ESTABLISHED to the parent and transitions to the IDLE state.

The final machine, sendMessage, manages the sending of the client messages. Since the protocol is simple, the machine needs only a second state, AWAITING_ACK, to ensure that no more than one message is outstanding.

Here, then, is the final skeleton:

```
machine hsmCommunicator
{

    event SEND_MESSAGE
          , MESSAGE_RECEIVED
          , TIMER_EXPIRED
          ;

    state IDLE
          , ESTABLISHING_SESSION
          , IN_SESSION
          ;

    machine establishSession
    {
        event ESTABLISH_SESSION
              , STEP0_RESPONSE
              , STEP1_RESPONSE
```

```
            , parent::MESSAGE_RECEIVED
            ;

      state IDLE
            , AWAITING_RESPONSE
            ;

   }

   machine sendMessage
   {

      event parent::SEND_MESSAGE
            , parent::MESSAGE_RECEIVED
            , ACK
            ;

      state IDLE
            , AWAITING_ACK
            ;

   }

}
```

The next pages flesh out each machine with the actions and transitions which make them perform. Each machine is treated on its own page, even though all exist together in a single .fsm source file.

Top-Level: hsmCommunicator

In this section, we look at adding actions and transitions for the top-level, hsmCommunicator. The FSMLang representation given here is not syntactically correct, since the sub-machines are not included. Their omission, however, makes the presentation easier to read.

The Design

The top-level machine with events and states looks like this:

```
machine hsmCommunicator
{

   event SEND_MESSAGE
         , MESSAGE_RECEIVED
         , TIMER_EXPIRED
         ;

   state IDLE
         , ESTABLISHING_SESSION
         , IN_SESSION
         ;

}
```

Normal operation would see the SEND_MESSAGE event appear first to the machine. The machine must somehow save the message, since the rule is that a session must first be established with the peer before a message may be sent. The top-level machine must also ask the establishSession machine to begin work. So, we'll add the following:

```
event SEND_MESSAGE, ESTABLISH_SESSION_REQUEST;
state IDLE;

machine establishSession;

/** This action also adds the message to the queue. */
action startSessionEstablishment[SEND_MESSAGE, IDLE] transition ESTABLISHING_SESSION;

startSessionEstablishment returns establishSession::ESTABLISH_SESSION_REQUEST;
```

Should other requests to send messages arrive before the session is established, the messages are simply queued.

```
event SEND_MESSAGE;
state ESTABLISHING_SESSION;

/** This action also adds the message to the queue. */
action queueMessage[SEND_MESSAGE, ESTABLISHING_SESSION];

queueMessage returns noEvent;
```

When the session is established, the establishSession sub-machine will return our top-level SESSION_ESTABLISHED event. At this point, we complete the starting of the session by passing the SEND_MESSAGE event to the sendMessage sub-machine.

```
event SESSION_ESTABLISHED, SEND_MESSAGE;
state ESTABLISHING_SESSION;

action completeSessionStart[SESSION_ESTABLISHED, ESTABLISHING_SESSION] transition IN_SESSION;

completeSessionStart returns noEvent;
```

While IN_SESSION, the machine queues incomming messages and passes the ESTABLISHED_SESSION event to the sendMessage machine.

```
event SEND_MESSAGE;
state IN_SESSION;

action requestMessageTransmission[SEND_MESSAGE, IN_SESSION];

requestMessageTransmission returns noEvent;
```

Also while IN_SESSION, it may be possible for the machine to receive the TIMER_EXPIRED event. This event requires no action, but simply a transition to the IDLE state.

```
event TIMER_EXPIRED;
state IN_SESSION, IDLE;

transition [TIMER_EXPIRED, IN_SESSION] IDLE;
```

The full top-level machine looks like this:

```
machine hsmCommunicator
{
```

The HSM Communicator

```
    event SEND_MESSAGE
          , MESSAGE_RECEIVED
          , TIMER_EXPIRED
          ;

    state IDLE
          , ESTABLISHING_SESSION
          , IN_SESSION
          ;

    machine establishSession
    {
       event START_SESSION_ESTABLISHMENT;
    }

    /** This action also adds the message to the queue. */
    action startSessionEstablishment[SEND_MESSAGE, IDLE] transition ESTABLISHING_SESSION;

    action queueMessage[SEND_MESSAGE, ESTABLISHING_SESSION];

    action completeSessionStart[SESSION_ESTABLISHED, ESTABLISHING_SESSION] transition IN_SESSION;

    action requestMessageTransmission[SEND_MESSAGE, IN_SESSION];

    transition [TIMER_EXPIRED, IN_SESSION] IDLE;

    startSessionEstablishment  returns establishSession::START_SESSION_ESTABLISHMENT;
    queueMessage               returns noEvent;
    completeSessionStart       returns noEvent;
    requestMessageTransmission returns noEvent;

}
```

The Generated Code

The command line, `fsm -tc --generate-weak-fns=false hsmCommunicator.fsm`, produces the following files:

Source files:

- **hsmCommunicator.c**
- establishSession.c
- sendMessage.c

Header files:

- **hsmCommunicator_priv.h**
- **hsmCommunicator.h**
- **hsmCommunicator_submach.h**
- **hsmCommunicator_events.h**
- establishSession_priv.h
- sendMessage_priv.h

In this section, we look only at the top-level files, *i.e.* the ones beginning with *hsmCommunicator*.

The top-level header is hsmCommunicator.h. It is the only file that should be included by code which uses the machine.

The header, hsmCommunicator_priv.h, is for the action function files. It contains all the definitions they need, and itself includes the top-level header.

The source code for the top-level machine is in hsmCommunicator.c.

As with a flat machine, the top-level header file provides convenience macros and a function through which the state machine may be run. Though macros are provided to directly inject sub-machine events, they should only be used if unavoidable, since this exposes the internals of the state machine, complicating any machine re-design.

```
#ifndef NO_CONVENIENCE_MACROS
#undef UFMN
#define UFMN(A) hsmCommunicator_##A
#undef THIS
#define THIS(A) hsmCommunicator_##A
#endif
#undef STATE
#define STATE(A) hsmCommunicator_##A
#undef HSM_COMMUNICATOR
#define HSM_COMMUNICATOR(A) hsmCommunicator_##A
#undef ESTABLISH_SESSION
#define ESTABLISH_SESSION(A) hsmCommunicator_establishSession_##A
#undef SEND_MESSAGE
#define SEND_MESSAGE(A) hsmCommunicator_sendMessage_##A

#undef ACTION_RETURN_TYPE
#define ACTION_RETURN_TYPE HSM_COMMUNICATOR_EVENT

void run_hsmCommunicator(HSM_COMMUNICATOR_EVENT);

typedef struct _hsmCommunicator_struct_ *pHSM_COMMUNICATOR;
extern pHSM_COMMUNICATOR phsmCommunicator;
```

This file, as with flat machines, includes the header containing the events enumeration. This enumeration is our first indication that we are dealing with a hierarchical machine.

```
typedef enum HSM_COMMUNICATOR_EVENT {
    hsmCommunicator_SEND_MESSAGE
    , hsmCommunicator_SESSION_ESTABLISHED
    , hsmCommunicator_SESSION_TIMEOUT
    , hsmCommunicator_MESSAGE_RECEIVED
    , hsmCommunicator_noEvent
    , hsmCommunicator_numEvents
    , hsmCommunicator_establishSession_firstEvent
    , hsmCommunicator_establishSession_ESTABLISH_SESSION_REQUEST = hsmCommunicator_establishSession_firstEvent
    , hsmCommunicator_establishSession_STEP0_RESPONSE
    , hsmCommunicator_establishSession_STEP1_RESPONSE
    , hsmCommunicator_establishSession_MESSAGE_RECEIVED
    , hsmCommunicator_establishSession_noEvent
    , hsmCommunicator_sendMessage_firstEvent
    , hsmCommunicator_sendMessage_SEND_MESSAGE = hsmCommunicator_sendMessage_firstEvent
    , hsmCommunicator_sendMessage_MESSAGE_RECEIVED
    , hsmCommunicator_sendMessage_ACK
    , hsmCommunicator_sendMessage_noEvent
    , hsmCommunicator_numAllEvents
}  HSM_COMMUNICATOR_EVENT;
```

All events for all machines appear in this one enumeration. The enumeration has some structure, with the special "…_firstEvent" entries, to allow easy descrimination, when used with "…noEvent" of which machine should handle each event.

Another important difference between flat and hierarchical FSMs is seen in hsmCommunicator_priv.h.

```
#include "hsmCommunicator_submach.h"

struct _hsmCommunicator_struct_ {
        HSM_COMMUNICATOR_STATE              state;
        HSM_COMMUNICATOR_EVENT              event;
        HSM_COMMUNICATOR_STATE_FN      const (*statesArray)[hsmCommunicator_numStates];
        pHSM_COMMUNICATOR_SUB_FSM_IF   const (*subMachineArray)[hsmCommunicator_numSubMachines];
        HSM_COMMUNICATOR_FSM                fsm;
};
```

The file inludes the hsmCommunicator_submach.h header which contains the material necessary for a parent machine to interact with its sub-machines. One of these items is the HSM_COMMUNICATOR_SUB_FSM_IF block which is examined below.

The other difference from a flat machine is the presence of a pointer to an array of these blocks; the array having one entry for each sub machine.

Looking into hsmCommunicator_submach.h, we find an enumeration of the sub-machines immediately before the sub-machine interface block:

```
typedef enum {
         establishSession_e
        , hsmCommunicator_firstSubMachine = establishSession_e
        ,  sendMessage_e
        , hsmCommunicator_numSubMachines
} HSM_COMMUNICATOR_SUB_MACHINES;

typedef HSM_COMMUNICATOR_EVENT (*HSM_COMMUNICATOR_SUB_MACHINE_FN)(HSM_COMMUNICATOR_EVENT);
typedef struct _hsmCommunicator_sub_fsm_if_ HSM_COMMUNICATOR_SUB_FSM_IF, *pHSM_COMMUNICATOR_SUB_FSM_IF;
struct _hsmCommunicator_sub_fsm_if_
{
        HSM_COMMUNICATOR_EVENT           first_event;
        HSM_COMMUNICATOR_EVENT           last_event;
        HSM_COMMUNICATOR_SUB_MACHINE_FN  subFSM;
};
```

The typedef for the sub-machine function shows that sub-machines, unlike top-level and flat machines, return events for machines in which actions return events (as the present example). This is how sub-machines are able to act as "sub-routines."

Sub-machines as Sub-routines

It is expected that a sub-machine must process a sequence of events in order to accomplish its task. As each intermediate event is handled, the sub-machine returns `parent::noEvent` to its parent, to indicate that it is still working. However, when the sub-machine is done, either through some happy path, or otherwise, the sub-machine will return some other event, to indicate the completion status.

Moving to the source file, we see how this structure is used by the top-level FSM function to select and execute appropriate sub-machines.

```
void hsmCommunicatorFSM(pHSM_COMMUNICATOR pfsm, HSM_COMMUNICATOR_EVENT event)
{
   HSM_COMMUNICATOR_EVENT e = event;

   while (e != hsmCommunicator_noEvent) {

#ifdef HSM_COMMUNICATOR_DEBUG
      if (EVENT_IS_NOT_EXCLUDED_FROM_LOG(e))
      {
         DBG_PRINTF("event: %s; state: %s"
                    ,HSM_COMMUNICATOR_EVENT_NAMES[e]
                    ,HSM_COMMUNICATOR_STATE_NAMES[pfsm->state]
                   );
      }
#endif

      /* This is read-only data to facilitate error reporting in action functions */
      pfsm->event = e;

      if (e < hsmCommunicator_noEvent)
      {
         e = ((* (*pfsm->statesArray)[pfsm->state])(pfsm,e));
      }
      else
      {
         e = findAndRunSubMachine(pfsm, e);
      }

   }
}
```

When the event being handled is less than the the machine's own *noEvent*, the machine goes to its own state function array, as would a flat machine. However, for events above that the machine finds, then runs, the approprate sub machine:

```
static HSM_COMMUNICATOR_EVENT findAndRunSubMachine(pHSM_COMMUNICATOR pfsm, HSM_COMMUNICATOR_EVENT e)
{
   for (HSM_COMMUNICATOR_SUB_MACHINES machineIterator = THIS(firstSubMachine);
        machineIterator < THIS(numSubMachines);
        machineIterator++
       )
   {
      if (
            ((*pfsm->subMachineArray)[machineIterator]->first_event <= e)
            && ((*pfsm->subMachineArray)[machineIterator]->last_event > e)
         )
      {
         return ((*(*pfsm->subMachineArray)[machineIterator]->subFSM)(e));
      }
   }

   return THIS(noEvent);

}
```

This function loops through the array of sub-machine blocks, looking for the one whose event range encompasses the event being handled. Upon finding the right block, the function pointer is used to invoke the sub-machine's FSM function, passing the event. The event returned by that function is returned from *findAndRunSubMachine*.

The calling FSM function then loops, looking at the event returned from the sub-machine. When that event is the top-level's own *noEvent*, the FSM function exits. Otherwise, it looks again for an action or machine to handle the new event.

As can be seen, a sub-machine can return an event that will be handled only by another sub-machine. However, doing so can quickly result in the kind of inter-weaving that FSMs are intended to prevent. Best is to have sub-machines only return events belonging to their parent. It would then be up to the parent to decide when that event should spark the running of another sub-machine.

This is illustrated by the interaction between the top-level machine and the *establishSession* sub-machine. When the top-level machine calls *startSessionEstablishment* to handle the *SEND_MESSAGE* event from the *IDLE* state, the function adds the message to the queue and returns *establishSession::BEGIN_SESSION_ESTABLISHMENT*. The top-level FSM function loops and quickly finds the *establishSession* as the machine which should handle the event. The receipt of two messages is required for the *establishSession* machine to complete its work; after processing the first message, it returns *parent::noEvent*, causing the top level machine to also simply exit. Upon the receipt of the second message, however, *establishSession* returns *parent::SESSION_ESTABLISHED*. The top-level machine processes this event by asking the *sendMessage* machine to begin sending messages from the queue.

As stated, *startSessionEstablishment* returns a sub-machine event in order to get that machine to do some work:

```
HSM_COMMUNICATOR_EVENT  UFMN(startSessionEstablishment)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;

    queue_count++;

    return ESTABLISH_SESSION(ESTABLISH_SESSION_REQUEST);
}
```

This is a good technique to use to start an idle machine.

For shared events, however, a different technique is used, as seen in *passMessageReceived*:

```
HSM_COMMUNICATOR_EVENT  UFMN(passMessageReceived)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return hsmCommunicator_pass_shared_event(sharing_hsmCommunicator_MESSAGE_RECEIVED);
}
```

The *pass_shared_event* function is defined in each parent machine's source file:

```
HSM_COMMUNICATOR_EVENT hsmCommunicator_pass_shared_event(pHSM_COMMUNICATOR_SHARED_EVENT_STR sharer_list[])
{
    HSM_COMMUNICATOR_EVENT return_event = THIS(noEvent);
    for (pHSM_COMMUNICATOR_SHARED_EVENT_STR *pcurrent_sharer = sharer_list;
            *pcurrent_sharer && return_event == THIS(noEvent);
            pcurrent_sharer++)
    {
        return_event = (*(*pcurrent_sharer)->psub_fsm_if->subFSM)((*pcurrent_sharer)->event);
    }

    return return_event;
}
```

Sharer lists are constructed for each parent event shared down to any sub-machines.

```
pHSM_COMMUNICATOR_SHARED_EVENT_STR sharing_hsmCommunicator_SEND_MESSAGE[] =
{
    &sendMessage_share_hsmCommunicator_SEND_MESSAGE_str
    , NULL
};

pHSM_COMMUNICATOR_SHARED_EVENT_STR sharing_hsmCommunicator_MESSAGE_RECEIVED[] =
{
    &establishSession_share_hsmCommunicator_MESSAGE_RECEIVED_str
    , &sendMessage_share_hsmCommunicator_MESSAGE_RECEIVED_str
    , NULL
};
```

As seen, the *MESSAGE_RECEIVED* event is always shared to both of the sub-machines. It is up to those sub-machines whether or not they act on that event in their current state. Though this can (and does) result in making a call to a sub-machine with an even that it will simply ignore, to do otherwise would bring the sub-machine's state chart into the parent, reducing the value of the hierarchical concept.

Sub-machine: establishSession

In this section, we look at adding actions and transitions for the sub-machine, establishSession.

The Design

establishSession, with events and states looks like this:

```
event MESSAGE_RECEIVED;

machine establishSession
{
   event ESTABLISH_SESSION_REQUEST
         , STEP0_RESPONSE
         , STEP1_RESPONSE
         , parent::MESSAGE_RECEIVED
         ;

   state IDLE
         , AWAITING_RESPONSE
         ;

}
```

The *ESTABLISH_SESSION_REQUEST* event is expected only in the *IDLE* state. When it is received, the machine sends the STEP0 message and moves to the *AWAITING_RESPONSE* state to receive the STEP0_RESPONSE.

```
event ESTABLISH_SESSION_REQUEST;
state IDLE, AWAITING_RESPONSE;

/** Start the session establishment process. */
action sendStep0Message[ESTABLISH_SESSION_REQUEST, IDLE]  transition AWAITING_RESPONSE;

sendStep0Message returns noEvent;
```

Obligingly, while in the *AWAITING_RESPONSE* state the *MESSAGE_RECEIVED* event is shared from the parent machine. *establishSession* must parse that message to see whether it is a step0 or step1 response.

```
event MESSAGE_RECEIVED, STEP0_RESPONSE, STEP1_RESPONSE;
state AWAITING_RESPONSE;

/** Parse the incoming message */
action parseMessage[MESSAGE_RECEIVED, AWAITING_RESPONSE];

parseMessage returns STEP0_RESPONSE, STEP1_RESPONSE, noEvent;
```

When the *STEP0_RESPONSE* is found, the step1 message is sent, and the machine remains in the *AWAITING_RESPONSE* state to await the next *MESSAGE_RECEIVED* event. When received, that message will again be parsed.

States are used to disambiguate actions which must be different upon subsequent receptions of a single event. It might be thought that since the *MESSAGE_RECEIVED* event is received twice, that the machine should move to a different state to await the event. This is not necessary, though, since the action is the same in each case: the message is parsed. Because the parsing yields a unique event for each message type, there is no ambiguity as to the action which must be taken.

Any temptation to design with the expectation that the second *MESSAGE_RECEIVED* event will be the expected *STEP1_RESPONSE* event should be effectively resisted. This example is the “happy path” to session establishment; it could very well be in the real world that an error indication may be received from the peer. Remaining in the *AWAITING_RESPONSE* state until the received message is parsed makes for the easist approache to designing for the “unhappy path.”

```
event STEP0_RESPONSE;
state AWAITING_RESPONSE;

/** Continue session establisment */
action sendStep1Message[STEP0_RESPONSE, AWAITING_RESPONSE];

sendStep1Message returns noEvent;
```

When the *STEP1_RESPONSE* is found, the machine notifies the parent that the session is established and returns to the *IDLE* state.

```
event SESSION_ESTABLISHED;

/** Notify parent that session is established. */
action notifyParent[STEP1_RESPONSE, AWAITING_RESPONSE] transition IDLE;

notifyParent     returns parent::SESSION_ESTABLISHED;
```

Remember that the parent machine begins to have messages send by the *sendMessage* sub-machine (the existance of which this machine is ignorant) once the session is established. Because of that, the parent will receive its *MESSAGE_RECEIVED* event when ACKs are sent from the peer. These events will be shared with this sub-machine, but will not be acted upon in the *IDLE* state. This highlights an important design principle, namely that sub-machines must return to a neutral state when their task is complete. For sub-machines such as this, which expect to be called more than once, that state is usually the initial state. For machines which expect to act only once, though, it may be required that they enter a *done* state, in which they react to no events.

A use-case for this last example might be a sub-machine which is acting as a co-routine to accomplish a long calculation. Once the calculation is finished, the machine must quiesce, even though the periodic event driving the machine may continue to be shared to it.

The full *establishSession* machine looks like this:

```
event MESSAGE_RECEIVED, SESSION_ESTABLISHED;

machine establishSession
{
   event ESTABLISH_SESSION_REQUEST
         , STEP0_RESPONSE
         , STEP1_RESPONSE
         , parent::MESSAGE_RECEIVED
         ;

   state IDLE
         , AWAITING_RESPONSE
         ;

   /** Start the session establishment process. */
   action sendStep0Message[ESTABLISH_SESSION_REQUEST, IDLE]  transition AWAITING_RESPONSE;

   /** Parse the incoming message */
   action parseMessage[MESSAGE_RECEIVED, AWAITING_RESPONSE];

   /** Continue session establisment */
   action sendStep1Message[STEP0_RESPONSE, AWAITING_RESPONSE];

   /** Notify parent that session is established. */
   action notifyParent[STEP1_RESPONSE, AWAITING_RESPONSE] transition IDLE;

   sendStep0Message returns noEvent;
   sendStep1Message returns noEvent;
   parseMessage     returns STEP0_RESPONSE, STEP1_RESPONSE, noEvent;
   notifyParent     returns parent::SESSION_ESTABLISHED;

}
```

The Generated Code

As mentioned in a previous section, the command line, `fsm -tc --generate-weak-fns=false hsmCommunicator.fsm`, produces the following files:

Source files:

- hsmCommunicator.c
- **establishSession.c**
- sendMessage.c

Header files:

- hsmCommunicator_priv.h
- hsmCommunicator.h
- hsmCommunicator_submach.h
- hsmCommunicator_events.h
- **establishSession_priv.h**
- sendMessage_priv.h

In this section, we look at only the files related to this sub-machine, *i.e.* the ones beginning with *establishSession*.

Being a sub-machine, *establishSession* has no function to call it directly from the outside world, nor does it publish its own events. Thus, neither *establishSession.h* nor *establishSession_events.h* are needed.

Note

Should *establishSession* have also been a parent machine, having at least one sub-machine, it would have needed the *establishSession_submach.h* file.

Also because it is a sub-machine of a machine having actions which return events, the FSM function for *establishSession* must return an event. But, because it has no sub-machines of its own, its FSM structure does not have a sub-machine interface block array. So, we find the following in *establishSession_priv.h*:

```
typedef HSM_COMMUNICATOR_EVENT (*ESTABLISH_SESSION_FSM)(FSM_TYPE_PTR,HSM_COMMUNICATOR_EVENT);

struct _establishSession_struct_ {
    ESTABLISH_SESSION_STATE            state;
    HSM_COMMUNICATOR_EVENT             event;
    ESTABLISH_SESSION_STATE_FN   const (*statesArray)[establishSession_numStates];
    ESTABLISH_SESSION_FSM              fsm;
};
```

As with the top-level, this header contains everything needed by the action functions file(s). Convenience macros are re-defined as necessary to fit the needs of this machine.

In the source file, *establishSession* must provide the sub-fsm interface block needed by its parent.

```
HSM_COMMUNICATOR_EVENT THIS(sub_machine_fn)(HSM_COMMUNICATOR_EVENT e)
{
    return establishSessionFSM(pestablishSession,e);
}

HSM_COMMUNICATOR_SUB_FSM_IF hsmCommunicator_establishSession_sub_fsm_if =
{
    .subFSM = THIS(sub_machine_fn)
    , .first_event = THIS(firstEvent)
    , .last_event = THIS(noEvent)
};
```

The structure needed for each event shared from the parent must also be provided.

```
HSM_COMMUNICATOR_SHARED_EVENT_STR establishSession_share_hsmCommunicator_MESSAGE_RECEIVED_str = {
    .event               = THIS(MESSAGE_RECEIVED)
    , .psub_fsm_if         = &hsmCommunicator_establishSession_sub_fsm_if
};
```

The FSM function implementation is closer to that of a flat FSM, needing only to pass on any event returned from an action function, when that event does not belong to this machine.

```
HSM_COMMUNICATOR_EVENT establishSessionFSM(pESTABLISH_SESSION pfsm, HSM_COMMUNICATOR_EVENT event)
{
    HSM_COMMUNICATOR_EVENT e = event;

    while ((e != THIS(noEvent))
           && (e >= THIS(firstEvent))
          )
    {
```

```
#ifdef HSM_COMMUNICATOR_ESTABLISH_SESSION_DEBUG
        if ((EVENT_IS_NOT_EXCLUDED_FROM_LOG(e))
                && (e >= THIS(firstEvent))
                && (e < THIS(noEvent))
           )
        {
            DBG_PRINTF("event: %s; state: %s"
                       ,ESTABLISH_SESSION_EVENT_NAMES[e - THIS(firstEvent)]
                       ,ESTABLISH_SESSION_STATE_NAMES[pfsm->state]
                      );
        }
#endif

        /* This is read-only data to facilitate error reporting in action functions */
        pfsm->event = e;

        if ((e >= THIS(firstEvent))
                && (e < THIS(noEvent))
           )
        {
            e = ((* (*pfsm->statesArray)[pfsm->state])(pfsm,e));
        }

    }

    return e == THIS(noEvent) ? PARENT(noEvent) : e;
}
```

Note that a local *noEvent* will stop the loop, but must be transformed to the parent's *noEvent* in order to be returned (otherwise, the parent would hand it back!).

Also note that should *establishSession* itself had had sub-machines, the conditional would have had an `else` block and a `findAndRunSubmachine` function.

The check on the event range for both the `while` and `if` constructs serves to capture local events. In our example, recall that the *parseMessage* action will return the local *STEP0_RESPONSE* and *STEP1_RESPONSE* events; these will fall in the range allowed in the loop and the conditional.

```
HSM_COMMUNICATOR_EVENT UFMN(parseMessage)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    static bool first = true;

    return first ? (first = false, THIS(STEP0_RESPONSE)) : THIS(STEP1_RESPONSE);
}
```

(This simplistic action function serves only to illustrate our point, of course.)

Our *notifyParent* action illustrates meaningfull communication back to the parent.

```
HSM_COMMUNICATOR_EVENT UFMN(notifyParent)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return PARENT(SESSION_ESTABLISHED);
}
```

The parent *SESSION_ESTABLISED* event falls outside of our range check (being below THIS(firstEvent) - see *hsmCommunicator_events.h*), and will thus end the loop. Since it is not equal to our local *noEvent*, it will not be transformed, but will be returned unchanged to the parent FSM function. That function will see it as one of its own events and will act on it as directed by its own state chart.

Sub-machine: sendMessage

In this section, we look at adding actions and transitions for the sub-machine, *sendMessage*.

The Design

sendMessage, with events and states looks like this:

```
event SEND_MESSAGE, MESSAGE_RECEIVED;

machine sendMessage
{

   event parent::SEND_MESSAGE
         , parent::MESSAGE_RECEIVED
         , ACK
         ;

   state IDLE
         , AWAITING_ACK
         ;

}
```

This machine will be in its *IDLE* state when it receives the first *SEND_MESSAGE* event from the parent. Thereafter, the machine may be in the *AWAITING_ACK* state. When recieved in the former state, the machine will pull the top message from the queue, send it, and transition to the *AWAITING_ACK* state. In that state, any *SEND_MESSAGE* event received will simply be ignored.

```
event SEND_MESSAGE;
state IDLE, AWAITING_ACK;

/** Dequeue and transmit message to the peer. */
action sendMessage[SEND_MESSAGE,IDLE] transition AWAITING_ACK;

sendMessage returns noEvent;
```

When the *MESSAGE_RECEIVED* event is shared to this machine in the *AWAITING_ACK* state, it must parse it. Since the event is not expected while *IDLE*, it will be ignored in that state.

```
event MESSAGE_RECEIVED;
state AWAITING_ACK;

action parseMessage[MESSAGE_RECEIVED, AWAITING_ACK];

parseMessage returns ACK, noEvent;
```

When the *ACK* is received in the *AWAITING_ACK* state, the queue is checked for more messages, and the machine transitions to the *IDLE* state. This ensures that the machine will handle the *SEND_MESSAGE* event returned by the checkQueue action when the queue is not empty. The transition cannot wait, since if *checkQueue* returns *noEvent*, the FSM function loop will exit, thus giving no futher opportunity for the transition to be made.

```
/** Check queue for messages; if found return SEND_MESSAGE; otherwise, return noEvent. */
action checkQueue[ACK,AWAITING_ACK] transition IDLE;

checkQueue returns SEND_MESSAGE, noEvent;
```

The full sendMessage machine looks like this:

```
machine sendMessage
{
   event parent::SEND_MESSAGE
         , parent::MESSAGE_RECEIVED
         , ACK
         ;

   state IDLE
         , AWAITING_ACK
         ;

   /** Dequeue and transmit message to the peer. */
   action sendMessage[SEND_MESSAGE,IDLE] transition AWAITING_ACK;

   /** Check queue for messages; if found return SEND_MESSAGE; otherwise, return noEvent. */
   action checkQueue[ACK,AWAITING_ACK] transition IDLE;

   action parseMessage[MESSAGE_RECEIVED, AWAITING_ACK];

   sendMessage  returns noEvent;
   checkQueue   returns SEND_MESSAGE, noEvent;
   parseMessage returns ACK, noEvent;

}
```

The Generated Code

As mentioned in a previous section, the command line, `fsm -tc --generate-weak-fns=false hsmCommunicator.fsm`, produces the following files:

Source files:

- hsmCommunicator.c
- establishSession.c
- **sendMessage.c**

Header files:

- hsmCommunicator_priv.h
- hsmCommunicator.h
- hsmCommunicator_submach.h
- hsmCommunicator_events.h
- establishSession_priv.h
- **sendMessage_priv.h**

It would be tedious to examine the *sendMessage* files, since they mimic those generated for the *establishSession* machine.

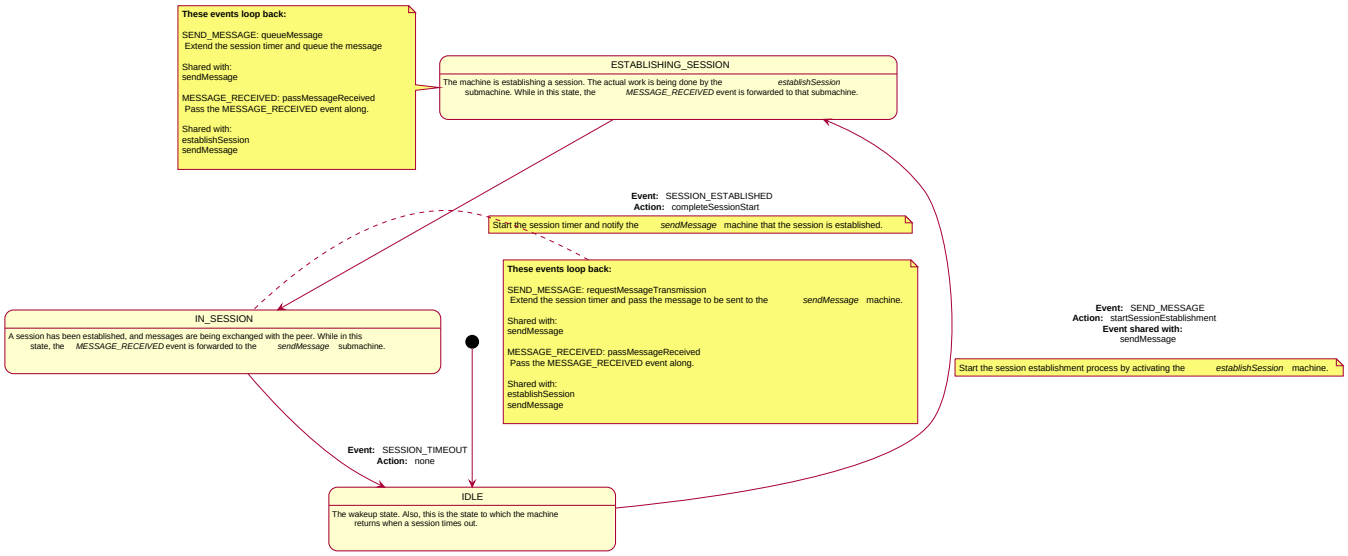
Visualizing the FSM

HTML Output

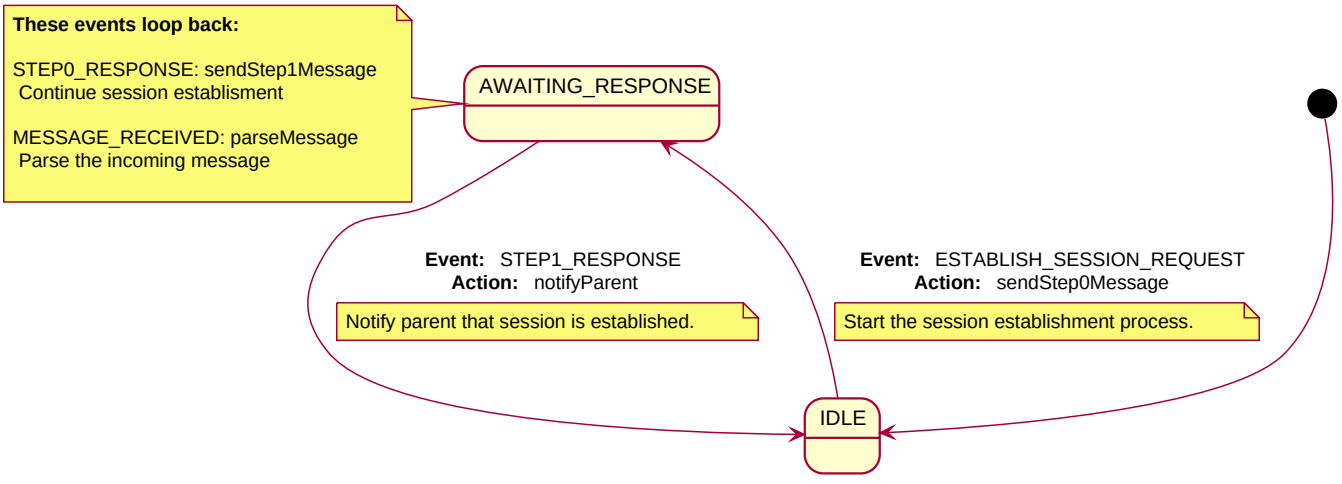
HTML pages can be generated with the `-th` command line switch.

PlantUML

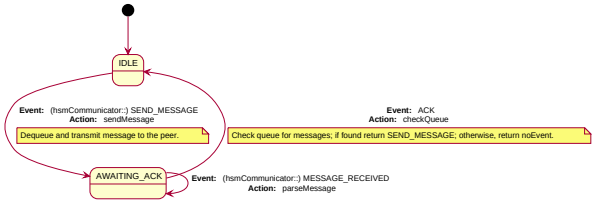
Or, the PlantUML code for creating the following diagrams can be generated with the `-tp` command line switch.



hsmCommunicator



establishSession



sendMessage

hsmCommunicator

This machine manages communications using a “stop and wait” protocol. Only one message is allowed to be outstanding.

Before any message can be exchanged, however, a session must be established with the peer. Establishing a connection requires several exchanges to authenticate. The session will remain active as long as messages continue to be exchanged with a minimum frequency.

The user of this machine calls run_hsmCommunicator, passing the SEND_MESSAGE event. For the first message, the machine will be IDLE, and thus needs to queue the message, start the establishSession machine, and transition to the ESTABLISHING_SESSION state. Requests to send messages received in this state will simply be queued. While the top level machine is in the ESTABLISHING_SESSION state, the establishSession machine does the establishment work.

When the establishSession machine receives the STEP1_RESPONSE event, it reports to the top level machine that the session is established by returning the parent’s SESSION_ESTABLISHED event. This will move the top level machine to its IN_SESSION state and cause it to send the message(s) which are enqueued.

establishSession

Establish a connection with the peer. Two messages must be exchanged with the peer to successfully establish the session. The machine needs only two states, IDLE and AWAITING_RESPONSE since the top level machine tracks whether or not it is in a session. The AWAITING_RESPONSE state serves for both required messages, since the receipt of each message produces a unique event. When the STEP1_RESPONSE event is received, the session is considered established. This machine will then return the parent’s SESSION_ESTABLISHED message and move to its IDLE state.

Machine Statistics

Number of events:	4
Events not handled:	0
Events handled in one state:	4
At least one event handled the same in all states?	no
Number of states:	2
Number of states with entry functions:	0
Number of states with exit functions:	0
States handling no events:	0
States handling exactly one event:	1
States with no way in:	0
States with no way out:	0

State Chart

The HSM Communicator

	ESTABLISH_SESSION_REQUEST	STEP0_RESPONSE	STEP1_RESPONSE	MESSAGE_RECEIVED
IDLE	sendStep0Message transition: AWAITING_RESPONSE			
AWAITING_RESPONSE		sendStep1Message	notifyParent transition: IDLE	parseMessage

Events

ESTABLISH_SESSION_REQUEST

This event is handled identically in 1 states.

These states handle this event:

- IDLE

This yields a state density of 50%.

These actions are taken in response to this event:

- sendStep0Message

STEP0_RESPONSE

This event is handled identically in 1 states.

These states handle this event:

- AWAITING_RESPONSE

This yields a state density of 50%.

These actions are taken in response to this event:

- sendStep1Message

STEP1_RESPONSE

This event is handled identically in 1 states.

These states handle this event:

- AWAITING_RESPONSE

This yields a state density of 50%.

These actions are taken in response to this event:

- notifyParent

MESSAGE_RECEIVED

This event is shared from the parent machine.

This event is handled identically in 1 states.

These states handle this event:

- AWAITING_RESPONSE

This yields a state density of 50%.

These actions are taken in response to this event:

- parseMessage

States

IDLE

These events are handled in this state:

- ESTABLISH_SESSION_REQUEST

This yields an event density of 25%.

These actions are taken in this state:

- sendStep0Message

These states transition into this state:

- AWAITING_RESPONSE

This state transitions into these states:

- AWAITING_RESPONSE

AWAITING_RESPONSE

These events are handled in this state:

- STEP0_RESPONSE
- STEP1_RESPONSE
- MESSAGE_RECEIVED

This yields an event density of 75%.

These actions are taken in this state:

- sendStep1Message
- notifyParent
- parseMessage

These states transition into this state:

- IDLE

This state transitions into these states:

- IDLE

Actions

sendStep0Message

This action returns:

- noEvent

The HSM Communicator

sendStep1Message

This action returns:

- noEvent

notifyParent

This action returns:

- SESSION_ESTABLISHED

parseMessage

This action returns:

- STEP0_RESPONSE
- STEP1_RESPONSE
- noEvent

sendMessage

Send a message to the peer. Since the protocol allows only one message to be outsanding, the machine dequeues and transmits a message only from the IDLE state, transitioning to the AWAITING_ACK state immediately thereafter. In the AWAITNG_ACK state, incomming messages are parsed and, when an ACK is found, the machine checks the queue and transitions to the IDLE state. Checking the queue can return the SEND_MESSAGE event, which will be handled from the IDLE state, thus resulting in a transmission and return to the AWAITING_ACK state.

Machine Statistics

Number of events:	3
Events not handled:	0
Events handled in one state:	3
At least one event handled the same in all states?	no
Number of states:	2
Number of states with entry functions:	0
Number of states with exit functions:	0
States handling no events:	0
States handling exactly one event:	1
States with no way in:	0
States with no way out:	0

State Chart

The HSM Communicator

	SEND_MESSAGE	MESSAGE_RECEIVED	ACK
IDLE	sendMessage transition: AWAITING_ACK		
AWAITING_ACK		parseMessage	checkQueue transition: IDLE

Events

SEND_MESSAGE

This event is shared from the parent machine.

This event is handled identically in 1 states.

These states handle this event:

- IDLE

This yields a state density of 50%.

These actions are taken in response to this event:

- sendMessage

MESSAGE_RECEIVED

This event is shared from the parent machine.

This event is handled identically in 1 states.

These states handle this event:

- AWAITING_ACK

This yields a state density of 50%.

These actions are taken in response to this event:

- parseMessage

ACK

This event is handled identically in 1 states.

These states handle this event:

- AWAITING_ACK

This yields a state density of 50%.

These actions are taken in response to this event:

- checkQueue

States

IDLE

These events are handled in this state:

- SEND_MESSAGE

This yields an event density of 33%.

These actions are taken in this state:

- sendMessage

These states transition into this state:

- AWAITING_ACK

This state transitions into these states:

- AWAITING_ACK

AWAITING_ACK

These events are handled in this state:

- ACK
- MESSAGE_RECEIVED

This yields an event density of 66%.

These actions are taken in this state:

- checkQueue
- parseMessage

These states transition into this state:

- IDLE

This state transitions into these states:

- IDLE

Actions

sendMessage

Send a message to the peer. Since the protocol allows only one message to be outsanding, the machine dequeues and transmits a message only from the IDLE state, transitioning to the AWAITING_ACK state immediately thereafter. In the AWAITNG_ACK state, incomming messages are parsed and, when an ACK is found, the machine checks the queue and transitions to the IDLE state. Checking the queue can return the SEND_MESSAGE event, which will be handled from the IDLE state, thus resulting in a transmission and return to the AWAITING_ACK state.

This action returns:

- noEvent

checkQueue

This action returns:

- SEND_MESSAGE
- noEvent

parseMessage

This action returns:

- ACK
- noEvent

Machine Statistics

Number of events:	4
Events not handled:	0
Events handled in one state:	2
At least one event handled the same in all states?	no
Number of states:	3
Number of states with entry functions:	0
Number of states with exit functions:	0
States handling no events:	0
States handling exactly one event:	1
States with no way in:	0
States with no way out:	0

State Chart

	SEND_MESSAGE	SESSION_ESTABLISHED	SESSION_TIMEOUT	MESSAGE_RECEIVED
IDLE	startSessionEstablishment transition: ESTABLISHING_SESSION			
ESTABLISHING_SESSION	queueMessage	completeSessionStart transition: IN_SESSION		passMessageReceived
IN_SESSION	requestMessageTransmission		transition: IDLE	passMessageReceived

Events

SEND_MESSAGE

This event comes from our client code, asking us to send a message.

These sub-machines share this event:

- sendMessage

These states handle this event:

- IDLE
- ESTABLISHING_SESSION
- IN_SESSION

This yields a state density of 100%.

These actions are taken in response to this event:

- startSessionEstablishment
- queueMessage
- requestMessageTransmission

SESSION_ESTABLISHED

This event comes from our establishSession submachine, indicating that it has successfully completed its work. We then forward it to our sendMessage submachine to indicate that it may now begin to send messages.

This event is handled identically in 1 states.

These states handle this event:

- ESTABLISHING_SESSION

This yields a state density of 33%.

These actions are taken in response to this event:

- completeSessionStart

SESSION_TIMEOUT

This event comes from our external timer, indicating that we've not tickled it in a while, and thus should close down our session.

This event is handled identically in 1 states.

These states handle this event:

- IN_SESSION

This yields a state density of 33%.

No actions are taken in response to this event.

MESSAGE_RECEIVED

This event comes from our lower comm layers, indicating that a peer message has arrived. While we're in the ESTABLISHING_SESSION state, we forward this event to the establishSession submachine; while in the IN_SESSION state, we forward it to the sendMessage submachine.

This event is handled identically in 2 states.

These sub-machines share this event:

- establishSession
- sendMessage

These states handle this event:

- ESTABLISHING_SESSION
- IN_SESSION

This yields a state density of 66%.

These actions are taken in response to this event:

- passMessageReceived

States

IDLE

The wakeup state. Also, this is the state to which the machine returns when a session times out.

These events are handled in this state:

- SEND_MESSAGE

This yields an event density of 25%.

These actions are taken in this state:

- startSessionEstablishment

These states transition into this state:

- IN_SESSION

This state transitions into these states:

- ESTABLISHING_SESSION

ESTABLISHING_SESSION

The machine is establishing a session. The actual work is being done by the establishSession submachine. While in this state, the MESSAGE_RECEIVED event is forwarded to that submachine.

These events are handled in this state:

- SESSION_ESTABLISHED
- MESSAGE_RECEIVED
- SEND_MESSAGE

This yields an event density of 75%.

These actions are taken in this state:

- completeSessionStart
- passMessageReceived
- queueMessage

These states transition into this state:

- IDLE

This state transitions into these states:

- IN_SESSION

The HSM Communicator

IN_SESSION

A session has been established, and messages are being exchanged with the peer. While in this state, the MESSAGE_RECEIVED event is forwarded to the sendMessage submachine.

These events are handled in this state:

- MESSAGE_RECEIVED
- SEND_MESSAGE
- SESSION_TIMEOUT

This yields an event density of 75%.

These actions are taken in this state:

- passMessageReceived
- requestMessageTransmission

These states transition into this state:

- ESTABLISHING_SESSION

This state transitions into these states:

- IDLE

Actions

startSessionEstablishment

This action returns:

- ESTABLISH_SESSION_REQUEST

completeSessionStart

This action returns:

- noEvent

passMessageReceived

queueMessage

This action returns:

- noEvent

requestMessageTransmission

This action returns:

- noEvent

Sources

These pages show the full source of the fsm and generated c and h files.

hsmCommunicator.fsm

Download

```
native
{
#ifndef DBG_PRINTF
#include <stdio.h>
#define DBG_PRINTF(A, ...) printf((A), __VA_ARGS__); printf("\n");
#endif

extern unsigned queue_count;
}

/**
This machine manages communications using a "stop and wait" protocol. Only one message is allowed to be outstanding.

Before any message can be exchanged, however, a session must be established with the peer. Establishing a connection
requires several exchanges to authenticate. The session will remain active as long as messages continue to be
exchanged with a minimum frequency.

The user of this machine calls run_hsmCommunicator, passing the SEND_MESSAGE event.  For the first message,
the machine will be IDLE, and thus needs to queue the message, start the establishSession machine, and transition
to the ESTABLISHING_SESSION state.  Requests to send messages received in this state will simply be queued.

While the top level machine is in the ESTABLISHING_SESSION state, the establishSession machine does the establishment work.

When the establishSession machine receives the STEP1_RESPONSE event, it reports to the top level machine that
the session is established by returning the parent's SESSION_ESTABLISHED event.  This will move the top level
machine to its IN_SESSION state and cause it to send the message(s) which are enqueued.

*/
machine hsmCommunicator
native impl
{
/*
   The barest skeleton of a queue has only a count.  We aren't really
   sending messages, after all.
*/
unsigned queue_count = 0;
}
{
    /** This event comes from our client code, asking us to send a message.
    */
    event SEND_MESSAGE;

    /** This event comes from our <i>establishSession</i> submachine, indicating that it has successfully
        completed its work.  We then forward it to our <i>sendMessage</i> submachine to indicate that
        it may now begin to send messages.
    */
    event SESSION_ESTABLISHED;

    /** This event comes from our external timer, indicating that we've not tickled it in a while, and
        thus should close down our session.


    */
    event SESSION_TIMEOUT;

    /** This event comes from our lower comm layers, indicating that a peer message has arrived.
        While we're in the ESTABLISHING_SESSION state, we forward this event to the <i>establishSession</i>
        submachine; while in the IN_SESSION state, we forward it to the <i>sendMessage</i> submachine.
    */
    event MESSAGE_RECEIVED;

    /** The wakeup state.  Also, this is the state to which the machine
        returns when a session times out.
    */
    state IDLE;

    /** The machine is establishing a session.  The actual work is being done by the <i>establishSession</i>


        submachine.  While in this state, the <i>MESSAGE_RECEIVED</i> event is forwarded to that submachine.
    */
    state ESTABLISHING_SESSION;
```

```
/** A session has been established, and messages are being exchanged with the peer.  While in this
    state, the <i>MESSAGE_RECEIVED</i> event is forwarded to the <i>sendMessage</i> submachine.
*/
state IN_SESSION;

/**

Establish a connection with the peer.


Two messages must be exchanged with the peer to successfully establish the session.  The machine needs
only two states, IDLE and AWAITING_RESPONSE since the top level machine tracks whether or not it is in a
session.  The AWAITING_RESPONSE state serves for both required messages, since the receipt of each message produces
a unique event.


When the STEP1_RESPONSE event is received, the session is considered established.  This machine will then
return the parent's SESSION_ESTABLISHED message and move to its IDLE state.

*/
machine establishSession
{
event ESTABLISH_SESSION_REQUEST, STEP0_RESPONSE, STEP1_RESPONSE;
event parent::MESSAGE_RECEIVED;

state IDLE, AWAITING_RESPONSE;

/** Start the session establishment process. */
action sendStep0Message[ESTABLISH_SESSION_REQUEST, IDLE]  transition AWAITING_RESPONSE;

/** Continue session establisment */
action sendStep1Message[STEP0_RESPONSE, AWAITING_RESPONSE];

/** Notify parent that session is established. */
action notifyParent[STEP1_RESPONSE, AWAITING_RESPONSE] transition IDLE;

/** Parse the incoming message */
action parseMessage[MESSAGE_RECEIVED, AWAITING_RESPONSE];

/* these lines are informational; they affect the html output, but do not affect any C code generated. */
sendStep0Message returns noEvent;

sendStep1Message returns noEvent;

notifyParent     returns parent::SESSION_ESTABLISHED;

parseMessage returns STEP0_RESPONSE, STEP1_RESPONSE, noEvent;
}

/**

Send a message to the peer.



Since the protocol allows only one message to be outsanding, the machine dequeues and transmits a message only
from the IDLE state, transitioning to the AWAITING_ACK state immediately thereafter.


In the AWAITNG_ACK state, incomming messages are parsed and, when an ACK is found, the machine checks the queue
and transitions to the IDLE state.  Checking the queue can return the SEND_MESSAGE event, which will be handled
from the IDLE state, thus resulting in a transmission and return to the AWAITING_ACK state.


*/
machine sendMessage
{
event       parent::SEND_MESSAGE
               , parent::MESSAGE_RECEIVED
               , ACK;

 state        IDLE, AWAITING_ACK;
```

```
    /** Dequeue and transmit message to the peer. */
    action      sendMessage[SEND_MESSAGE,IDLE] transition AWAITING_ACK;

    /** Check queue for messages; if found return SEND_MESSAGE; otherwise, return noEvent. */
    action      checkQueue[ACK,AWAITING_ACK]          transition IDLE;

    action      parseMessage[MESSAGE_RECEIVED, AWAITING_ACK];

    /* these lines are informational; they affect the html output, but do not affect any C code generated. */
    sendMessage  returns noEvent;

    checkQueue   returns SEND_MESSAGE, noEvent;

    parseMessage returns ACK, noEvent;

    }

    /* these are actions of the top level machine */

    /** Start the session establishment process by activating the <i>establishSession</i> machine. */
    action startSessionEstablishment[SEND_MESSAGE, IDLE] transition  ESTABLISHING_SESSION;

    /** Start the session timer and notify the <i>sendMessage</i> machine that the session is established. */
    action completeSessionStart[SESSION_ESTABLISHED, ESTABLISHING_SESSION] transition IN_SESSION;

    /** Pass the MESSAGE_RECEIVED event along. */
    action passMessageReceived[MESSAGE_RECEIVED, (ESTABLISHING_SESSION, IN_SESSION)];

    /** Extend the session timer and queue the message */
    action queueMessage[SEND_MESSAGE, ESTABLISHING_SESSION];

    /** Extend the session timer and pass the message to be sent to the <i>sendMessage</i> machine. */
    action requestMessageTransmission[SEND_MESSAGE, IN_SESSION];

    transition [SESSION_TIMEOUT, IN_SESSION] IDLE;


    /* these lines are informational; they affect the html output, but do not affect any C code generated. */
    startSessionEstablishment   returns establishSession::ESTABLISH_SESSION_REQUEST;

    completeSessionStart        returns noEvent;

    requestMessageTransmission  returns noEvent;

    queueMessage                returns noEvent;
}
```

hsmCommunicator.c

```
/**
        hsmCommunicator.c

        This file automatically generated by FSMLang
*/

#include "hsmCommunicator_priv.h"
#include <stddef.h>

/* Begin Native Implementation Prolog */


/*
   The barest skeleton of a queue has only a count.  We aren't really
   sending messages, after all.
*/
unsigned queue_count = 0;

/* End Native Implementation Prolog */


#ifndef DBG_PRINTF
#define DBG_PRINTF(...)
#endif

extern HSM_COMMUNICATOR_SUB_FSM_IF hsmCommunicator_establishSession_sub_fsm_if;
extern HSM_COMMUNICATOR_SUB_FSM_IF hsmCommunicator_sendMessage_sub_fsm_if;

static HSM_COMMUNICATOR_EVENT IDLE_stateFn(pHSM_COMMUNICATOR,HSM_COMMUNICATOR_EVENT);
static HSM_COMMUNICATOR_EVENT ESTABLISHING_SESSION_stateFn(pHSM_COMMUNICATOR,HSM_COMMUNICATOR_EVENT);
static HSM_COMMUNICATOR_EVENT IN_SESSION_stateFn(pHSM_COMMUNICATOR,HSM_COMMUNICATOR_EVENT);
static const HSM_COMMUNICATOR_STATE_FN hsmCommunicator_state_fn_array[hsmCommunicator_numStates] =
{
    IDLE_stateFn
    , ESTABLISHING_SESSION_stateFn
    , IN_SESSION_stateFn
};


const pHSM_COMMUNICATOR_SUB_FSM_IF hsmCommunicator_sub_fsm_if_array[THIS(numSubMachines)] =
{
    &hsmCommunicator_establishSession_sub_fsm_if
    , &hsmCommunicator_sendMessage_sub_fsm_if
};

pHSM_COMMUNICATOR_SHARED_EVENT_STR sharing_hsmCommunicator_SEND_MESSAGE[] =
{
    &sendMessage_share_hsmCommunicator_SEND_MESSAGE_str
    , NULL
};


pHSM_COMMUNICATOR_SHARED_EVENT_STR sharing_hsmCommunicator_MESSAGE_RECEIVED[] =
{
    &establishSession_share_hsmCommunicator_MESSAGE_RECEIVED_str
    , &sendMessage_share_hsmCommunicator_MESSAGE_RECEIVED_str
    , NULL
};

HSM_COMMUNICATOR_EVENT hsmCommunicator_pass_shared_event(pHSM_COMMUNICATOR_SHARED_EVENT_STR sharer_list[])
{
    HSM_COMMUNICATOR_EVENT return_event = THIS(noEvent);
```

```
    for (pHSM_COMMUNICATOR_SHARED_EVENT_STR *pcurrent_sharer = sharer_list;
            *pcurrent_sharer && return_event == THIS(noEvent);
            pcurrent_sharer++)
    {
        return_event = (*(*pcurrent_sharer)->psub_fsm_if->subFSM)((*pcurrent_sharer)->event);
    }

    return return_event;
}

HSM_COMMUNICATOR hsmCommunicator = {
    hsmCommunicator_IDLE,
    THIS(SEND_MESSAGE),
    &hsmCommunicator_state_fn_array,
    &hsmCommunicator_sub_fsm_if_array,
    hsmCommunicatorFSM
};

pHSM_COMMUNICATOR phsmCommunicator = &hsmCommunicator;

void run_hsmCommunicator(HSM_COMMUNICATOR_EVENT e)
{
    if (phsmCommunicator)
    {
        phsmCommunicator->fsm(phsmCommunicator,e);
    }
}

static HSM_COMMUNICATOR_EVENT findAndRunSubMachine(pHSM_COMMUNICATOR,HSM_COMMUNICATOR_EVENT);


#ifndef EVENT_IS_NOT_EXCLUDED_FROM_LOG
#define EVENT_IS_NOT_EXCLUDED_FROM_LOG(e) ((e) == (e))
#endif
void hsmCommunicatorFSM(pHSM_COMMUNICATOR pfsm, HSM_COMMUNICATOR_EVENT event)
{
    HSM_COMMUNICATOR_EVENT e = event;

    while (e != hsmCommunicator_noEvent) {

#ifdef HSM_COMMUNICATOR_DEBUG
        if (EVENT_IS_NOT_EXCLUDED_FROM_LOG(e))
        {
            DBG_PRINTF("event: %s; state: %s"
                       ,HSM_COMMUNICATOR_EVENT_NAMES[e]
                       ,HSM_COMMUNICATOR_STATE_NAMES[pfsm->state]
                      );
        }
#endif

        /* This is read-only data to facilitate error reporting in action functions */
        pfsm->event = e;


          if (e < hsmCommunicator_noEvent)
          {
              e = ((* (*pfsm->statesArray)[pfsm->state])(pfsm,e));
          }
          else
          {
              e = findAndRunSubMachine(pfsm, e);
          }

     }
```

```
}


static HSM_COMMUNICATOR_EVENT findAndRunSubMachine(pHSM_COMMUNICATOR pfsm, HSM_COMMUNICATOR_EVENT e)
{
    for (HSM_COMMUNICATOR_SUB_MACHINES machineIterator = THIS(firstSubMachine);
            machineIterator < THIS(numSubMachines);
            machineIterator++
        )
    {
        if (
            ((*pfsm->subMachineArray)[machineIterator]->first_event <= e)
            && ((*pfsm->subMachineArray)[machineIterator]->last_event > e)
        )
        {
            return ((*(*pfsm->subMachineArray)[machineIterator]->subFSM)(e));
        }
    }

    return THIS(noEvent);

}

static HSM_COMMUNICATOR_EVENT IDLE_stateFn(pHSM_COMMUNICATOR pfsm,HSM_COMMUNICATOR_EVENT e)
{
    HSM_COMMUNICATOR_EVENT retVal = THIS(noEvent);

    switch(e)
    {
    case THIS(SEND_MESSAGE):
        retVal = UFMN(startSessionEstablishment)(pfsm);
        pfsm->state = hsmCommunicator_ESTABLISHING_SESSION;
        break;
    default:
        DBG_PRINTF("hsmCommunicator_noAction");
        break;
    }

    return retVal;
}

static HSM_COMMUNICATOR_EVENT ESTABLISHING_SESSION_stateFn(pHSM_COMMUNICATOR pfsm,HSM_COMMUNICATOR_EVENT e)
{
    HSM_COMMUNICATOR_EVENT retVal = THIS(noEvent);

    switch(e)
    {
    case THIS(SESSION_ESTABLISHED):
        retVal = UFMN(completeSessionStart)(pfsm);
        pfsm->state = hsmCommunicator_IN_SESSION;
        break;
    case THIS(MESSAGE_RECEIVED):
        retVal = UFMN(passMessageReceived)(pfsm);

          break;
     case THIS(SEND_MESSAGE):
          retVal = UFMN(queueMessage)(pfsm);
          break;
     default:
          DBG_PRINTF("hsmCommunicator_noAction");
          break;
     }

     return retVal;
```

```
}

static HSM_COMMUNICATOR_EVENT IN_SESSION_stateFn(pHSM_COMMUNICATOR pfsm,HSM_COMMUNICATOR_EVENT e)
{
    HSM_COMMUNICATOR_EVENT retVal = THIS(noEvent);

    switch(e)
    {
    case THIS(MESSAGE_RECEIVED):
        retVal = UFMN(passMessageReceived)(pfsm);
        break;
    case THIS(SEND_MESSAGE):
        retVal = UFMN(requestMessageTransmission)(pfsm);
        break;
    case THIS(SESSION_TIMEOUT):
        DBG_PRINTF("hsmCommunicator_noAction");
        pfsm->state = hsmCommunicator_IDLE;
        break;
    default:
        DBG_PRINTF("hsmCommunicator_noAction");
        break;
    }

    return retVal;
}



#ifdef HSM_COMMUNICATOR_DEBUG
char *HSM_COMMUNICATOR_EVENT_NAMES[] = {
    "hsmCommunicator_SEND_MESSAGE"
    ,"hsmCommunicator_SESSION_ESTABLISHED"
    ,"hsmCommunicator_SESSION_TIMEOUT"
    ,"hsmCommunicator_MESSAGE_RECEIVED"
    , "hsmCommunicator_noEvent"
    , "hsmCommunicator_numEvents"
    , "hsmCommunicator_establishSession_ESTABLISH_SESSION_REQUEST"
    , "hsmCommunicator_establishSession_STEP0_RESPONSE"
    , "hsmCommunicator_establishSession_STEP1_RESPONSE"
    , "hsmCommunicator_establishSession_MESSAGE_RECEIVED"
    , "hsmCommunicator_establishSession_noEvent"
    , "hsmCommunicator_sendMessage_SEND_MESSAGE"
    , "hsmCommunicator_sendMessage_MESSAGE_RECEIVED"
    , "hsmCommunicator_sendMessage_ACK"
    , "hsmCommunicator_sendMessage_noEvent"
};

char *HSM_COMMUNICATOR_STATE_NAMES[] = {
    "hsmCommunicator_IDLE"
    ,"hsmCommunicator_ESTABLISHING_SESSION"
    ,"hsmCommunicator_IN_SESSION"
};

#endif
```

hsmCommunicator.h

```
/**
        hsmCommunicator.h

        This file automatically generated by FSMLang
*/

#ifndef _HSMCOMMUNICATOR_H_
#define _HSMCOMMUNICATOR_H_

#include "hsmCommunicator_events.h"
#ifndef HSM_COMMUNICATOR_NATIVE_PROLOG
#define HSM_COMMUNICATOR_NATIVE_PROLOG

#ifndef DBG_PRINTF
#include <stdio.h>
#define DBG_PRINTF(...) printf(__VA_ARGS__); printf("\n")
#endif

extern unsigned queue_count;

#endif
#define FSM_VERSION "1.45.1"

#ifndef NO_CONVENIENCE_MACROS
#undef UFMN
#define UFMN(A) hsmCommunicator_##A
#undef THIS
#define THIS(A) hsmCommunicator_##A
#endif
#undef STATE
#define STATE(A) hsmCommunicator_##A
#undef HSM_COMMUNICATOR
#define HSM_COMMUNICATOR(A) hsmCommunicator_##A
#undef ESTABLISH_SESSION
#define ESTABLISH_SESSION(A) hsmCommunicator_establishSession_##A
#undef SEND_MESSAGE
#define SEND_MESSAGE(A) hsmCommunicator_sendMessage_##A

#undef ACTION_RETURN_TYPE
#define ACTION_RETURN_TYPE HSM_COMMUNICATOR_EVENT

void run_hsmCommunicator(HSM_COMMUNICATOR_EVENT);

typedef struct _hsmCommunicator_struct_ *pHSM_COMMUNICATOR;
extern pHSM_COMMUNICATOR phsmCommunicator;


#endif
```

sendMessage_priv.h

```
/**
        sendMessage_priv.h

        This file automatically generated by FSMLang
*/

#ifndef _SENDMESSAGE_PRIV_H_
#define _SENDMESSAGE_PRIV_H_

#include "hsmCommunicator_submach.h"
#include "hsmCommunicator.h"

#ifdef SEND_MESSAGE_DEBUG
#include <stdio.h>
#include <stdlib.h>
#endif

/*
        sub-machine events are included in the top-level machine event enumeration.
        These macros set the appropriate names for events from THIS machine
        and those from the PARENT machine.

        They may be turned off as needed.
*/
#ifndef NO_CONVENIENCE_MACROS
#undef UFMN
#define UFMN(A) hsmCommunicator_sendMessage_##A
#undef THIS
#define THIS(A) hsmCommunicator_sendMessage_##A
#undef PARENT
#define PARENT(A) hsmCommunicator_##A
#endif
#undef STATE
#define STATE(A) sendMessage_##A

#ifdef HSM_COMMUNICATOR_SEND_MESSAGE_DEBUG
extern char *SEND_MESSAGE_EVENT_NAMES[];
extern char *SEND_MESSAGE_STATE_NAMES[];
#endif

typedef enum {
    sendMessage_IDLE
    , sendMessage_AWAITING_ACK
    , sendMessage_numStates
}  SEND_MESSAGE_STATE;

typedef struct _sendMessage_struct_ SEND_MESSAGE, *pSEND_MESSAGE;
#undef FSM_TYPE_PTR
#define FSM_TYPE_PTR pSEND_MESSAGE
extern SEND_MESSAGE sendMessage;
```

```
extern pSEND_MESSAGE psendMessage;

typedef HSM_COMMUNICATOR_EVENT (*SEND_MESSAGE_ACTION_FN)(FSM_TYPE_PTR);

typedef HSM_COMMUNICATOR_EVENT (*SEND_MESSAGE_FSM)(FSM_TYPE_PTR,HSM_COMMUNICATOR_EVENT);

typedef ACTION_RETURN_TYPE (*SEND_MESSAGE_STATE_FN)(pSEND_MESSAGE,HSM_COMMUNICATOR_EVENT);

static const SEND_MESSAGE_STATE_FN sendMessage_state_fn_array[sendMessage_numStates];

struct _sendMessage_struct_ {
    SEND_MESSAGE_STATE            state;
    HSM_COMMUNICATOR_EVENT        event;
    SEND_MESSAGE_STATE_FN   const (*statesArray)[sendMessage_numStates];
    SEND_MESSAGE_FSM              fsm;
};


ACTION_RETURN_TYPE hsmCommunicator_sendMessage_sendMessage(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_sendMessage_checkQueue(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_sendMessage_parseMessage(FSM_TYPE_PTR);


#endif
```

hsmCommunicator_priv.h

```
/**
        hsmCommunicator_priv.h

        This file automatically generated by FSMLang
*/

#ifndef _HSMCOMMUNICATOR_PRIV_H_
#define _HSMCOMMUNICATOR_PRIV_H_

#include "hsmCommunicator.h"
#ifndef HSM_COMMUNICATOR_NATIVE_PROLOG
#define HSM_COMMUNICATOR_NATIVE_PROLOG

#ifndef DBG_PRINTF
#include <stdio.h>
#define DBG_PRINTF(...) printf(__VA_ARGS__); printf("\n")
#endif

extern unsigned queue_count;

#endif

#ifdef HSM_COMMUNICATOR_DEBUG
#include <stdio.h>
#include <stdlib.h>
#endif


#ifdef HSM_COMMUNICATOR_DEBUG
extern char *HSM_COMMUNICATOR_EVENT_NAMES[];
extern char *HSM_COMMUNICATOR_STATE_NAMES[];
#endif

typedef enum {
    hsmCommunicator_IDLE
    , hsmCommunicator_ESTABLISHING_SESSION
    , hsmCommunicator_IN_SESSION
    , hsmCommunicator_numStates
}  HSM_COMMUNICATOR_STATE;

typedef struct _hsmCommunicator_struct_ HSM_COMMUNICATOR;
#undef FSM_TYPE_PTR
#define FSM_TYPE_PTR pHSM_COMMUNICATOR
extern HSM_COMMUNICATOR hsmCommunicator;

typedef HSM_COMMUNICATOR_EVENT (*HSM_COMMUNICATOR_ACTION_FN)(FSM_TYPE_PTR);

typedef void (*HSM_COMMUNICATOR_FSM)(FSM_TYPE_PTR,HSM_COMMUNICATOR_EVENT);

void hsmCommunicatorFSM(FSM_TYPE_PTR,HSM_COMMUNICATOR_EVENT);
```

```
#include "hsmCommunicator_submach.h"
typedef ACTION_RETURN_TYPE (*HSM_COMMUNICATOR_STATE_FN)(pHSM_COMMUNICATOR,HSM_COMMUNICATOR_EVENT);

static const HSM_COMMUNICATOR_STATE_FN hsmCommunicator_state_fn_array[hsmCommunicator_numStates];

struct _hsmCommunicator_struct_ {
    HSM_COMMUNICATOR_STATE              state;
    HSM_COMMUNICATOR_EVENT              event;
    HSM_COMMUNICATOR_STATE_FN     const (*statesArray)[hsmCommunicator_numStates];
    pHSM_COMMUNICATOR_SUB_FSM_IF   const (*subMachineArray)[hsmCommunicator_numSubMachines];
    HSM_COMMUNICATOR_FSM                fsm;
};


ACTION_RETURN_TYPE hsmCommunicator_startSessionEstablishment(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_completeSessionStart(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_passMessageReceived(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_queueMessage(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_requestMessageTransmission(FSM_TYPE_PTR);



#endif
```

hsmCommunicator_submach.h

```
/**
        hsmCommunicator_submach.h

        This file automatically generated by FSMLang
*/

#ifndef _HSMCOMMUNICATOR_SUBMACH_H_
#define _HSMCOMMUNICATOR_SUBMACH_H_

#include "hsmCommunicator.h"
#ifndef HSM_COMMUNICATOR_NATIVE_PROLOG
#define HSM_COMMUNICATOR_NATIVE_PROLOG

#ifndef DBG_PRINTF
#include <stdio.h>
#define DBG_PRINTF(...) printf(__VA_ARGS__); printf("\n")
#endif

extern unsigned queue_count;

#endif
/* Sub Machine Declarations */

typedef enum {
    establishSession_e
    , hsmCommunicator_firstSubMachine = establishSession_e
    ,  sendMessage_e
    , hsmCommunicator_numSubMachines
} HSM_COMMUNICATOR_SUB_MACHINES;

typedef HSM_COMMUNICATOR_EVENT (*HSM_COMMUNICATOR_SUB_MACHINE_FN)(HSM_COMMUNICATOR_EVENT);
typedef struct _hsmCommunicator_sub_fsm_if_ HSM_COMMUNICATOR_SUB_FSM_IF, *pHSM_COMMUNICATOR_SUB_FSM_IF;
struct _hsmCommunicator_sub_fsm_if_
{
    HSM_COMMUNICATOR_EVENT           first_event;
    HSM_COMMUNICATOR_EVENT           last_event;
    HSM_COMMUNICATOR_SUB_MACHINE_FN  subFSM;
};

/* Some sub-machines share parent events. */
typedef struct _hsmCommunicator_shared_event_str_ HSM_COMMUNICATOR_SHARED_EVENT_STR, *pHSM_COMMUNICATOR_SHARED_EVENT_STR;
struct _hsmCommunicator_shared_event_str_
{
    HSM_COMMUNICATOR_EVENT             event;
    pHSM_COMMUNICATOR_SUB_FSM_IF       psub_fsm_if;
};
extern HSM_COMMUNICATOR_EVENT hsmCommunicator_pass_shared_event(pHSM_COMMUNICATOR_SHARED_EVENT_STR[]);

extern HSM_COMMUNICATOR_SHARED_EVENT_STR sendMessage_share_hsmCommunicator_SEND_MESSAGE_str;
extern pHSM_COMMUNICATOR_SHARED_EVENT_STR sharing_hsmCommunicator_SEND_MESSAGE[];

extern HSM_COMMUNICATOR_SHARED_EVENT_STR establishSession_share_hsmCommunicator_MESSAGE_RECEIVED_str;
extern HSM_COMMUNICATOR_SHARED_EVENT_STR sendMessage_share_hsmCommunicator_MESSAGE_RECEIVED_str;
extern pHSM_COMMUNICATOR_SHARED_EVENT_STR sharing_hsmCommunicator_MESSAGE_RECEIVED[];


#endif
```

hsmc-actions.c

Download

```
/*
  hsmc-actions.c

        Action functions for the hsmCommunicator top-level FSM.

    FSMLang (fsm) - A Finite State Machine description language.
```

```
    Copyright (C) 2024  Steven Stanton

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

    Steven Stanton
    sstanton@pesticidesoftware.com

    For the latest on FSMLang: https://fsmlang.github.io/

    And, finally, your possession of this source code implies nothing.

*/

#include "hsmCommunicator_priv.h"

HSM_COMMUNICATOR_EVENT  UFMN(startSessionEstablishment)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;

    queue_count++;

    return ESTABLISH_SESSION(ESTABLISH_SESSION_REQUEST);
}

HSM_COMMUNICATOR_EVENT  UFMN(completeSessionStart)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return hsmCommunicator_pass_shared_event(sharing_hsmCommunicator_SEND_MESSAGE);
}

HSM_COMMUNICATOR_EVENT  UFMN(passMessageReceived)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return hsmCommunicator_pass_shared_event(sharing_hsmCommunicator_MESSAGE_RECEIVED);
}

HSM_COMMUNICATOR_EVENT  UFMN(queueMessage)(FSM_TYPE_PTR pfsm)
{
```

```
    DBG_PRINTF("%s", __func__);
    (void) pfsm;

    queue_count++;

    return THIS(noEvent);
}

HSM_COMMUNICATOR_EVENT  UFMN(requestMessageTransmission)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;

    queue_count++;

    return hsmCommunicator_pass_shared_event(sharing_hsmCommunicator_SEND_MESSAGE);
}

HSM_COMMUNICATOR_EVENT  UFMN(noAction)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}

int main(void)
{
    run_hsmCommunicator(THIS(SEND_MESSAGE));
    run_hsmCommunicator(THIS(MESSAGE_RECEIVED));
    run_hsmCommunicator(THIS(SEND_MESSAGE));
    run_hsmCommunicator(THIS(MESSAGE_RECEIVED));
    run_hsmCommunicator(THIS(MESSAGE_RECEIVED));
    run_hsmCommunicator(THIS(MESSAGE_RECEIVED));

    return 0;
}
```

establishSession.c

```
/**
        establishSession.c

        This file automatically generated by FSMLang
*/

#include "establishSession_priv.h"
#include <stddef.h>


#ifndef DBG_PRINTF
#define DBG_PRINTF(...)
#endif

static HSM_COMMUNICATOR_EVENT establishSessionFSM(FSM_TYPE_PTR,HSM_COMMUNICATOR_EVENT);

static HSM_COMMUNICATOR_EVENT IDLE_stateFn(pESTABLISH_SESSION,HSM_COMMUNICATOR_EVENT);
static HSM_COMMUNICATOR_EVENT AWAITING_RESPONSE_stateFn(pESTABLISH_SESSION,HSM_COMMUNICATOR_EVENT);

static const ESTABLISH_SESSION_STATE_FN establishSession_state_fn_array[establishSession_numStates] =
{
    IDLE_stateFn
    , AWAITING_RESPONSE_stateFn
};

HSM_COMMUNICATOR_EVENT THIS(sub_machine_fn)(HSM_COMMUNICATOR_EVENT e)
{
    return establishSessionFSM(pestablishSession,e);
}

HSM_COMMUNICATOR_SUB_FSM_IF hsmCommunicator_establishSession_sub_fsm_if =
{
    .subFSM = THIS(sub_machine_fn)
    , .first_event = THIS(firstEvent)
    , .last_event = THIS(noEvent)
};

HSM_COMMUNICATOR_SHARED_EVENT_STR establishSession_share_hsmCommunicator_MESSAGE_RECEIVED_str = {
    .event               = THIS(MESSAGE_RECEIVED)
    , .psub_fsm_if         = &hsmCommunicator_establishSession_sub_fsm_if
};

ESTABLISH_SESSION establishSession = {
    establishSession_IDLE,
    THIS(ESTABLISH_SESSION_REQUEST),
    &establishSession_state_fn_array,
    establishSessionFSM
};

pESTABLISH_SESSION pestablishSession = &establishSession;



#ifndef EVENT_IS_NOT_EXCLUDED_FROM_LOG
#define EVENT_IS_NOT_EXCLUDED_FROM_LOG(e) (e == e)
#endif
HSM_COMMUNICATOR_EVENT establishSessionFSM(pESTABLISH_SESSION pfsm, HSM_COMMUNICATOR_EVENT event)
{
    HSM_COMMUNICATOR_EVENT e = event;

    while ((e != THIS(noEvent))
```

```
             && (e >= THIS(firstEvent))
           )
    {

#ifdef HSM_COMMUNICATOR_ESTABLISH_SESSION_DEBUG
        if ((EVENT_IS_NOT_EXCLUDED_FROM_LOG(e))
                && (e >= THIS(firstEvent))
                && (e < THIS(noEvent))
           )
        {
            DBG_PRINTF("event: %s; state: %s"
                       ,ESTABLISH_SESSION_EVENT_NAMES[e - THIS(firstEvent)]
                       ,ESTABLISH_SESSION_STATE_NAMES[pfsm->state]
                      );
        }
#endif

        /* This is read-only data to facilitate error reporting in action functions */
        pfsm->event = e;

        if ((e >= THIS(firstEvent))
                && (e < THIS(noEvent))
           )
        {
            e = ((* (*pfsm->statesArray)[pfsm->state])(pfsm,e));
        }

    }

    return e == THIS(noEvent) ? PARENT(noEvent) : e;
}

static HSM_COMMUNICATOR_EVENT IDLE_stateFn(pESTABLISH_SESSION pfsm,HSM_COMMUNICATOR_EVENT e)
{
    HSM_COMMUNICATOR_EVENT retVal = THIS(noEvent);

    switch(e)
    {
    case THIS(ESTABLISH_SESSION_REQUEST):
        retVal = UFMN(sendStep0Message)(pfsm);
        pfsm->state = establishSession_AWAITING_RESPONSE;
        break;
    default:
        DBG_PRINTF("hsmCommunicator_establishSession_noAction");
        break;
    }

    return retVal;
}

static HSM_COMMUNICATOR_EVENT AWAITING_RESPONSE_stateFn(pESTABLISH_SESSION pfsm,HSM_COMMUNICATOR_EVENT e)
{
    HSM_COMMUNICATOR_EVENT retVal = THIS(noEvent);


     switch(e)
     {
     case THIS(STEP0_RESPONSE):
         retVal = UFMN(sendStep1Message)(pfsm);
         break;
     case THIS(STEP1_RESPONSE):
         retVal = UFMN(notifyParent)(pfsm);
         pfsm->state = establishSession_IDLE;
```

```
            break;
        case THIS(MESSAGE_RECEIVED):
            retVal = UFMN(parseMessage)(pfsm);
            break;
        default:
            DBG_PRINTF("hsmCommunicator_establishSession_noAction");
            break;
        }

        return retVal;
    }



    #ifdef HSM_COMMUNICATOR_ESTABLISH_SESSION_DEBUG
    char *ESTABLISH_SESSION_EVENT_NAMES[] = {
        "hsmCommunicator_establishSession_ESTABLISH_SESSION_REQUEST"
        ,"hsmCommunicator_establishSession_STEP0_RESPONSE"
        ,"hsmCommunicator_establishSession_STEP1_RESPONSE"
        ,"hsmCommunicator_establishSession_MESSAGE_RECEIVED"
        , "establishSession_noEvent"
        , "establishSession_numEvents"
    };

    char *ESTABLISH_SESSION_STATE_NAMES[] = {
        "hsmCommunicator_establishSession_IDLE"
        ,"hsmCommunicator_establishSession_AWAITING_RESPONSE"
    };

    #endif
```

establishSession_priv.h

```
/**
        establishSession_priv.h

        This file automatically generated by FSMLang
*/

#ifndef _ESTABLISHSESSION_PRIV_H_
#define _ESTABLISHSESSION_PRIV_H_

#include "hsmCommunicator_submach.h"
#include "hsmCommunicator.h"

#ifdef ESTABLISH_SESSION_DEBUG
#include <stdio.h>
#include <stdlib.h>
#endif

/*
        sub-machine events are included in the top-level machine event enumeration.
        These macros set the appropriate names for events from THIS machine
        and those from the PARENT machine.

        They may be turned off as needed.
*/
#ifndef NO_CONVENIENCE_MACROS
#undef UFMN
#define UFMN(A) hsmCommunicator_establishSession_##A
#undef THIS
#define THIS(A) hsmCommunicator_establishSession_##A
#undef PARENT
#define PARENT(A) hsmCommunicator_##A
#endif
#undef STATE
#define STATE(A) establishSession_##A

#ifdef HSM_COMMUNICATOR_ESTABLISH_SESSION_DEBUG
extern char *ESTABLISH_SESSION_EVENT_NAMES[];
extern char *ESTABLISH_SESSION_STATE_NAMES[];
#endif

typedef enum {
    establishSession_IDLE
    , establishSession_AWAITING_RESPONSE
    , establishSession_numStates
}  ESTABLISH_SESSION_STATE;

typedef struct _establishSession_struct_ ESTABLISH_SESSION, *pESTABLISH_SESSION;
#undef FSM_TYPE_PTR
#define FSM_TYPE_PTR pESTABLISH_SESSION
extern ESTABLISH_SESSION establishSession;
```

```
extern pESTABLISH_SESSION pestablishSession;

typedef HSM_COMMUNICATOR_EVENT (*ESTABLISH_SESSION_ACTION_FN)(FSM_TYPE_PTR);

typedef HSM_COMMUNICATOR_EVENT (*ESTABLISH_SESSION_FSM)(FSM_TYPE_PTR,HSM_COMMUNICATOR_EVENT);

typedef ACTION_RETURN_TYPE (*ESTABLISH_SESSION_STATE_FN)(pESTABLISH_SESSION,HSM_COMMUNICATOR_EVENT);

static const ESTABLISH_SESSION_STATE_FN establishSession_state_fn_array[establishSession_numStates];

struct _establishSession_struct_ {
    ESTABLISH_SESSION_STATE            state;
    HSM_COMMUNICATOR_EVENT             event;
    ESTABLISH_SESSION_STATE_FN   const (*statesArray)[establishSession_numStates];
    ESTABLISH_SESSION_FSM              fsm;
};


ACTION_RETURN_TYPE hsmCommunicator_establishSession_sendStep0Message(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_establishSession_sendStep1Message(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_establishSession_notifyParent(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_establishSession_parseMessage(FSM_TYPE_PTR);


#endif
```

session-actions.c

Download

```
/*
  session_actions.c

        Action functions for the establishSession FSM.

        FSMLang (fsm) - A Finite State Machine description language.
        Copyright (C) 4/20/2025  Steven Stanton

        This program is free software; you can redistribute it and/or modify
        it under the terms of the GNU General Public License as published by
        the Free Software Foundation; either version 2 of the License, or
        (at your option) any later version.

        This program is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
        GNU General Public License for more details.

        You should have received a copy of the GNU General Public License
        along with this program; if not, write to the Free Software
        Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

        Steven Stanton
        sstanton@pesticidesoftware.com

        For the latest on FSMLang: http://fsmlang.github.io
```

```
        And, finally, your possession of this source code implies nothing.

*/

#include <stdbool.h>
#include "establishSession_priv.h"

HSM_COMMUNICATOR_EVENT UFMN(sendStep0Message)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}

HSM_COMMUNICATOR_EVENT UFMN(sendStep1Message)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}

HSM_COMMUNICATOR_EVENT UFMN(notifyParent)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return PARENT(SESSION_ESTABLISHED);
}

HSM_COMMUNICATOR_EVENT UFMN(parseMessage)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    static bool first = true;

    return first ? (first = false, THIS(STEP0_RESPONSE)) : THIS(STEP1_RESPONSE);
}

HSM_COMMUNICATOR_EVENT UFMN(noAction)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}
```

sendMessage.c

```
/**
        sendMessage.c

        This file automatically generated by FSMLang
*/

#include "sendMessage_priv.h"
#include <stddef.h>


#ifndef DBG_PRINTF
#define DBG_PRINTF(...)
#endif

static HSM_COMMUNICATOR_EVENT sendMessageFSM(FSM_TYPE_PTR,HSM_COMMUNICATOR_EVENT);

static HSM_COMMUNICATOR_EVENT IDLE_stateFn(pSEND_MESSAGE,HSM_COMMUNICATOR_EVENT);
static HSM_COMMUNICATOR_EVENT AWAITING_ACK_stateFn(pSEND_MESSAGE,HSM_COMMUNICATOR_EVENT);

static const SEND_MESSAGE_STATE_FN sendMessage_state_fn_array[sendMessage_numStates] =
{
    IDLE_stateFn
    , AWAITING_ACK_stateFn
};

HSM_COMMUNICATOR_EVENT THIS(sub_machine_fn)(HSM_COMMUNICATOR_EVENT e)
{
    return sendMessageFSM(psendMessage,e);
}

HSM_COMMUNICATOR_SUB_FSM_IF hsmCommunicator_sendMessage_sub_fsm_if =
{
    .subFSM = THIS(sub_machine_fn)
    , .first_event = THIS(firstEvent)
    , .last_event = THIS(noEvent)
};

HSM_COMMUNICATOR_SHARED_EVENT_STR sendMessage_share_hsmCommunicator_SEND_MESSAGE_str = {
    .event               = THIS(SEND_MESSAGE)
    , .psub_fsm_if         = &hsmCommunicator_sendMessage_sub_fsm_if
};

HSM_COMMUNICATOR_SHARED_EVENT_STR sendMessage_share_hsmCommunicator_MESSAGE_RECEIVED_str = {
    .event               = THIS(MESSAGE_RECEIVED)
    , .psub_fsm_if         = &hsmCommunicator_sendMessage_sub_fsm_if
};

SEND_MESSAGE sendMessage = {
    sendMessage_IDLE,
    THIS(SEND_MESSAGE),
    &sendMessage_state_fn_array,

     sendMessageFSM
};

pSEND_MESSAGE psendMessage = &sendMessage;
```

```
#ifndef EVENT_IS_NOT_EXCLUDED_FROM_LOG
#define EVENT_IS_NOT_EXCLUDED_FROM_LOG(e) (e == e)
#endif
HSM_COMMUNICATOR_EVENT sendMessageFSM(pSEND_MESSAGE pfsm, HSM_COMMUNICATOR_EVENT event)
{
    HSM_COMMUNICATOR_EVENT e = event;

    while ((e != THIS(noEvent))
            && (e >= THIS(firstEvent))
          )
    {

#ifdef HSM_COMMUNICATOR_SEND_MESSAGE_DEBUG
        if ((EVENT_IS_NOT_EXCLUDED_FROM_LOG(e))
                && (e >= THIS(firstEvent))
                && (e < THIS(noEvent))
           )
        {
            DBG_PRINTF("event: %s; state: %s"
                       ,SEND_MESSAGE_EVENT_NAMES[e - THIS(firstEvent)]
                       ,SEND_MESSAGE_STATE_NAMES[pfsm->state]
                      );
        }
#endif

        /* This is read-only data to facilitate error reporting in action functions */
        pfsm->event = e;

        if ((e >= THIS(firstEvent))
                && (e < THIS(noEvent))
           )
        {
            e = ((* (*pfsm->statesArray)[pfsm->state])(pfsm,e));
        }

    }

    return e == THIS(noEvent) ? PARENT(noEvent) : e;
}

static HSM_COMMUNICATOR_EVENT IDLE_stateFn(pSEND_MESSAGE pfsm,HSM_COMMUNICATOR_EVENT e)
{
    HSM_COMMUNICATOR_EVENT retVal = THIS(noEvent);

    switch(e)
    {
    case THIS(SEND_MESSAGE):
        retVal = UFMN(sendMessage)(pfsm);
        pfsm->state = sendMessage_AWAITING_ACK;
        break;
    default:
        DBG_PRINTF("hsmCommunicator_sendMessage_noAction");

         break;
     }

     return retVal;
```

```
}

static HSM_COMMUNICATOR_EVENT AWAITING_ACK_stateFn(pSEND_MESSAGE pfsm,HSM_COMMUNICATOR_EVENT e)
{
    HSM_COMMUNICATOR_EVENT retVal = THIS(noEvent);

    switch(e)
    {
    case THIS(ACK):
        retVal = UFMN(checkQueue)(pfsm);
        pfsm->state = sendMessage_IDLE;
        break;
    case THIS(MESSAGE_RECEIVED):
        retVal = UFMN(parseMessage)(pfsm);
        break;
    default:
        DBG_PRINTF("hsmCommunicator_sendMessage_noAction");
        break;
    }

    return retVal;
}



#ifdef HSM_COMMUNICATOR_SEND_MESSAGE_DEBUG
char *SEND_MESSAGE_EVENT_NAMES[] = {
    "hsmCommunicator_sendMessage_SEND_MESSAGE"
    ,"hsmCommunicator_sendMessage_MESSAGE_RECEIVED"
    ,"hsmCommunicator_sendMessage_ACK"
    , "sendMessage_noEvent"
    , "sendMessage_numEvents"
};

char *SEND_MESSAGE_STATE_NAMES[] = {
    "hsmCommunicator_sendMessage_IDLE"
    ,"hsmCommunicator_sendMessage_AWAITING_ACK"
};

#endif
```

sendMessage_priv.h

```
/**
        sendMessage_priv.h

        This file automatically generated by FSMLang
*/

#ifndef _SENDMESSAGE_PRIV_H_
#define _SENDMESSAGE_PRIV_H_

#include "hsmCommunicator_submach.h"
#include "hsmCommunicator.h"

#ifdef SEND_MESSAGE_DEBUG
#include <stdio.h>
#include <stdlib.h>
#endif

/*
        sub-machine events are included in the top-level machine event enumeration.
        These macros set the appropriate names for events from THIS machine
        and those from the PARENT machine.

        They may be turned off as needed.
*/
#ifndef NO_CONVENIENCE_MACROS
#undef UFMN
#define UFMN(A) hsmCommunicator_sendMessage_##A
#undef THIS
#define THIS(A) hsmCommunicator_sendMessage_##A
#undef PARENT
#define PARENT(A) hsmCommunicator_##A
#endif
#undef STATE
#define STATE(A) sendMessage_##A

#ifdef HSM_COMMUNICATOR_SEND_MESSAGE_DEBUG
extern char *SEND_MESSAGE_EVENT_NAMES[];
extern char *SEND_MESSAGE_STATE_NAMES[];
#endif

typedef enum {
    sendMessage_IDLE
    , sendMessage_AWAITING_ACK
    , sendMessage_numStates
}  SEND_MESSAGE_STATE;

typedef struct _sendMessage_struct_ SEND_MESSAGE, *pSEND_MESSAGE;
#undef FSM_TYPE_PTR
#define FSM_TYPE_PTR pSEND_MESSAGE
extern SEND_MESSAGE sendMessage;
```

```
extern pSEND_MESSAGE psendMessage;

typedef HSM_COMMUNICATOR_EVENT (*SEND_MESSAGE_ACTION_FN)(FSM_TYPE_PTR);

typedef HSM_COMMUNICATOR_EVENT (*SEND_MESSAGE_FSM)(FSM_TYPE_PTR,HSM_COMMUNICATOR_EVENT);

typedef ACTION_RETURN_TYPE (*SEND_MESSAGE_STATE_FN)(pSEND_MESSAGE,HSM_COMMUNICATOR_EVENT);

static const SEND_MESSAGE_STATE_FN sendMessage_state_fn_array[sendMessage_numStates];

struct _sendMessage_struct_ {
    SEND_MESSAGE_STATE            state;
    HSM_COMMUNICATOR_EVENT        event;
    SEND_MESSAGE_STATE_FN   const (*statesArray)[sendMessage_numStates];
    SEND_MESSAGE_FSM              fsm;
};


ACTION_RETURN_TYPE hsmCommunicator_sendMessage_sendMessage(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_sendMessage_checkQueue(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_sendMessage_parseMessage(FSM_TYPE_PTR);


#endif
```

message-actions.c

Download

```
/*
  message_actions.c

        Action functions for the sendMessage FSM.

        FSMLang (fsm) - A Finite State Machine description language.
        Copyright (C) 4/20/2025  Steven Stanton

        This program is free software; you can redistribute it and/or modify
        it under the terms of the GNU General Public License as published by
        the Free Software Foundation; either version 2 of the License, or
        (at your option) any later version.

        This program is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
        GNU General Public License for more details.

        You should have received a copy of the GNU General Public License
        along with this program; if not, write to the Free Software
        Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

        Steven Stanton
        sstanton@pesticidesoftware.com

        For the latest on FSMLang: http://fsmlang.github.io
```

```
        And, finally, your possession of this source code implies nothing.
*/

#include "sendMessage_priv.h"

HSM_COMMUNICATOR_EVENT UFMN(sendMessage)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;

    queue_count--;

    return THIS(noEvent);
}

HSM_COMMUNICATOR_EVENT UFMN(parseMessage)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return THIS(ACK);
}

HSM_COMMUNICATOR_EVENT UFMN(checkQueue)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;

    return (queue_count > 0) ? THIS(SEND_MESSAGE) : THIS(noEvent);
}

HSM_COMMUNICATOR_EVENT UFMN(noAction)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}
```

Data for Machines and Events

Both machines and events can have associated data. To explore the topic, we'll revisit The HSM Communicator, adding a place in the machine to hold the most recently received peer message, and giving the MESSAGE_RECEIVED event a way to actually convey that message.

Machine Data

We will use the *hsmCommunicator* as our example, since there is more to see in the hierarchical case. And, since the top level machine behaves in this respect like a flat machine, there is nothing lost in the illustration.

Our peer messages are of three types: the step 0 response, the step 1 response, and, the ACK. So, to differentiate, and to carry all of the necessary content, a simple enumeration will suffice:

```
typedef enum
{
   msg_none
   , msg_step0_response
   , msg_step1_response
   , msg_ack
} msg_e_t;
```

The simple nature of our requirements allow us to put all of the material directly into the state machine definition.

More complex data structures, of course, are better placed into headers and simply included into the machine.

Leaving out most of the machine structure for the moment, here is how we add the data and the mandatory data initialization macro:

Peer Message

```
native
{
   typedef enum
   {
      msg_none
      , msg_step0_response
      , msg_step1_response
      , msg_ack
   } msg_e_t;
}
machine hsmCommunicator
native impl
{
   #define INIT_FSM_DATA {msg_none, { 0 }}
}
{
   data
   {
      msg_e_t current_msg;
   }

   /* other machine stuff here */

}
```

That’s it.

Sub-machines can have their own data. Indeed, in our case, it is the sub-machines which parse the message, so each having its own copy would work well (and illustrate how machines share data at run-time). So, let’s give each sub-machine its own copy of the current message. While we’re at it, we’ll move the “queue” into the data block as well, and give it a structure.

```
native
{
   typedef enum
   {
      msg_none
      , msg_step0_response
      , msg_step1_response
      , msg_ack
```

```
    } msg_e_t;

    typedef struct _queue_str_
    {
       unsigned queue_count;
    } queue_str_t;
}
machine hsmCommunicator
native impl
{
   #define INIT_FSM_DATA {msg_none, { 0 }}
}
{
   data
   {
      msg_e_t current_msg;
      queue_str_t queue;
   }

   machine establishSession
   native impl
   {
      #define INIT_FSM_DATA {msg_none}
   }
   {

      data
      {
         msg_e_t current_msg;
      }

   }


   machine sendMessage
   native impl
   {
      #define INIT_FSM_DATA {msg_none, NULL}
   }
   {

      data
      {
         msg_e_t current_msg;
         queue_str_t *pqueue;
      }

   }

}
```

Looking at the generated code, we find the differences we might expect.

First, the Data structure is defined. For machines with children, this happens in the submachine header; for childless machines, it is in the private header.

Then, the data structure is placed in the FSM structure. For all machines, this is in the private header.

hsmCommunicator_submach.h

```
typedef struct _hsmCommunicator_data_struct_ HSM_COMMUNICATOR_DATA, *pHSM_COMMUNICATOR_DATA;

struct _hsmCommunicator_data_struct_ {
        msg_e_t current_msg;
        queue_str_t queue;
};
```

hsmCommunicator_priv.h:

```
struct _hsmCommunicator_struct_ {
        HSM_COMMUNICATOR_DATA           data;
        HSM_COMMUNICATOR_STATE          state;
        HSM_COMMUNICATOR_EVENT          event;
        HSM_COMMUNICATOR_ACTION_TRANS   const   (*actionArray)[THIS(numEvents)][hsmCommunicator_numStates];
        pHSM_COMMUNICATOR_SUB_FSM_IF     const  (*subMachineArray)[hsmCommunicator_numSubMachines];
        HSM_COMMUNICATOR_FSM           fsm;
};
```

establishSession_priv.h:

```
typedef struct _establishSession_data_struct_ ESTABLISH_SESSION_DATA, *pESTABLISH_SESSION_DATA;

struct _establishSession_data_struct_ {
        msg_e_t current_msg;
};

struct _establishSession_struct_ {
        ESTABLISH_SESSION_DATA           data;
        ESTABLISH_SESSION_STATE          state;
        HSM_COMMUNICATOR_EVENT           event;
        ESTABLISH_SESSION_ACTION_TRANS   const  (*actionArray)[THIS(numEvents)][establishSession_numStates];
        ESTABLISH_SESSION_FSM           fsm;
};
```

sendMessage_priv.h

```
typedef struct _sendMessage_data_struct_ SEND_MESSAGE_DATA, *pSEND_MESSAGE_DATA;

struct _sendMessage_data_struct_ {
        msg_e_t current_msg;
        queue_str_t * pqueue;
};

struct _sendMessage_struct_ {
        SEND_MESSAGE_DATA           data;
        SEND_MESSAGE_STATE          state;
        HSM_COMMUNICATOR_EVENT      event;
        SEND_MESSAGE_ACTION_TRANS   const       (*actionArray)[THIS(numEvents)][sendMessage_numStates];
        SEND_MESSAGE_FSM           fsm;
};
```

To sum up, data declared for a machine are wrapped into a structure which becomes a field in the main FSM structure. At run-time, user functions access this data structure through the pointer they are given to the machine structure.

How do submachines gain access?

Taking a look, again, into the submachine header, we find that the shared event structure has been altered.

```
struct _hsmCommunicator_shared_event_str_
{
        HSM_COMMUNICATOR_EVENT                  event;
        HSM_COMMUNICATOR_DATA_TRANSLATION_FN    data_translation_fn;
        pHSM_COMMUNICATOR_SUB_FSM_IF            psub_fsm_if;
};
```

Then, looking in the parent machine's source file, we find the sharing function has been modified to take advantage of the `data_translation_function` field.

Submachine Data Translation Function

```
HSM_COMMUNICATOR_EVENT hsmCommunicator_pass_shared_event(pHSM_COMMUNICATOR pfsm,pHSM_COMMUNICATOR_SHARED_EVENT_STR sharer_list[])
{
        HSM_COMMUNICATOR_EVENT return_event = THIS(noEvent);
        for (pHSM_COMMUNICATOR_SHARED_EVENT_STR *pcurrent_sharer = sharer_list;
             *pcurrent_sharer && return_event == THIS(noEvent);
             pcurrent_sharer++)
        {
            if ((*pcurrent_sharer)->data_translation_fn)
                    (*(*pcurrent_sharer)->data_translation_fn)(&pfsm->data);
            return_event = (*(*pcurrent_sharer)->psub_fsm_if->subFSM)((*pcurrent_sharer)->event);
        }

        return return_event;
}
```

These lines are present because both the parent and the child have data, and the child shares an event with the parent.

Note that the submachine event's data translation function is given a pointer only to the parent's data, not to the entire machine structure. Also note that the translation function is called once, before the event is passed to the submachine's FSM function. For any data to be visible to the submachine's user functions, the data must somehow be made part of the submachine's own data structure. This is why tranlators are only provided for when both parent and child have data.

Any submachine data which derive from the parent machine's data must be initialized at run-time. A common way to do this is to provide an initialization event which is shared to submachines as required. It is also common to then have an *uninitialized* state which only reacts to this event and is exited once the event is received.

Our top-level machine, then adds:

```
/** System initialization */
event INIT;

/** The wakeup state. */
state UNINITIALIZED;

/** Initialize the machine */
action initialize[INIT, UNINITIALIZED] transition IDLE;
```

sendMessage is the only sub-machine needing to be initialized, requiring, as it does, a pointer to the common queue.

```
event parent::INIT data translator init_data;

transition [INIT, UNINITIALIZED] IDLE;
```

The initialization work takes place in the data translator function, so only a transition to the IDLE state is needed. (Were there any sub-machines here requiring initialization, however, an action would be needed in order to share the event to them.)

sendMessages's init_data function looks like this:

```
void UFMN(init_data)(pHSM_COMMUNICATOR_DATA pfsm_data)
{
    DBG_PRINTF("%s", __func__);

        psendMessage->data.pqueue = &pfsm_data->queue;

}
```

At the top-level, we've seen no way to bring data to the machine from the outside, save by exposing the pointer to the machine, or by providing *setter* functions. The proper event-driven state machine way to do this is through the events themselves, which is the topic of our next section.

Event Data

Attaching data to events is the natural way to introduce those data to the state machine. Recall the definition of the top-level machine's current message data: Peer Message. This definition can be easily attached to the machines MESSAGE_RECEIVED event.

```
event MESSAGE_RECEIVED
      data
      {
         msg_e_t message;
      }
      ;
```

FSMLang creates a structure out of the declared data. Then, a union is made to house the data from all events declaring any. Finally, the machine's *event* becomes a structure containing the event enumeration and the union. This is done in the public header, since the events concerned originate in the outside world.

```
typedef struct _hsmCommunicator_MESSAGE_RECEIVED_data_ {
        msg_e_t message;
} HSM_COMMUNICATOR_MESSAGE_RECEIVED_DATA, *pHSM_COMMUNICATOR_MESSAGE_RECEIVED_DATA;

typedef union {
        HSM_COMMUNICATOR_MESSAGE_RECEIVED_DATA MESSAGE_RECEIVED_data;
} HSM_COMMUNICATOR_EVENT_DATA, *pHSM_COMMUNICATOR_EVENT_DATA;

typedef struct {
        HSM_COMMUNICATOR_EVENT_ENUM event;
        HSM_COMMUNICATOR_EVENT_DATA event_data;
} HSM_COMMUNICATOR_EVENT, *pHSM_COMMUNICATOR_EVENT;
```

The public function made available to run the machine now takes a pointer to an event structure, rather than the event enumeration.

```
void run_hsmCommunicator(pHSM_COMMUNICATOR_EVENT);
```

And, the ACTION_RETURN_TYPE macro is redefined to match the new name of the event enumeration.

```
#define ACTION_RETURN_TYPE HSM_COMMUNICATOR_EVENT_ENUM
```

FSMLang chose to rename the event enumeration to allow the event structure to bear the most natural name. Since handy convenience macros are provided, well written source code should not need to be adjusted to accommodate this.

The task of getting the event's data into the machine happens through a translator function called inside of the main FSM function.

```
void hsmCommunicatorFSM(pHSM_COMMUNICATOR pfsm, pHSM_COMMUNICATOR_EVENT event)
{
    HSM_COMMUNICATOR_EVENT_ENUM new_e;
    HSM_COMMUNICATOR_EVENT_ENUM e = event->event;
    translateEventData(&pfsm->data, event);

    while (e != THIS(noEvent)) {

#ifdef HSM_COMMUNICATOR_DEBUG
        if (EVENT_IS_NOT_EXCLUDED_FROM_LOG(e))
        {
            DBG_PRINTF("event: %s; state: %s"
                       ,HSM_COMMUNICATOR_EVENT_NAMES[e]
                       ,HSM_COMMUNICATOR_STATE_NAMES[pfsm->state]
                      );
        }
#endif

        /* This is read-only data to facilitate error reporting in action functions */
        pfsm->event = e;

        if (e < hsmCommunicator_noEvent)
        {

            new_e = ((* (*pfsm->actionArray)[e][pfsm->state].action)(pfsm));

            pfsm->state = (*pfsm->actionArray)[e][pfsm->state].transition;

            e = new_e;

        }
        else
        {
            e = findAndRunSubMachine(pfsm, e);
        }

    }

}
```

`translateEvent` is defined as:

```
static void translateEventData(pHSM_COMMUNICATOR_DATA pfsm_data, pHSM_COMMUNICATOR_EVENT pevent)
{
    switch(pevent->event)
    {
    case hsmCommunicator_MESSAGE_RECEIVED:
        UFMN(translate_MESSAGE_RECEIVED_data)(pfsm_data, &pevent->event_data.MESSAGE_RECEIVED_data);
        break;
    default:
        break;
    }

}
```

The name, `translate_MESSAGE_RECEIVED_data` was concted by FSMLang; to provide your own name, simply specify a translator as part of the event's data declaration.

```
event MESSAGE_RECEIVED
      data
      translator store_message
      {
         msg_e_t message;
      }
      ;
```

Then, the specified name is used in the generated code:

```
static void translateEventData(pHSM_COMMUNICATOR_DATA pfsm_data, pHSM_COMMUNICATOR_EVENT pevent)
{
        switch(pevent->event)
        {
        case hsmCommunicator_MESSAGE_RECEIVED:
                UFMN(store_message)(pfsm_data, &pevent->event_data.MESSAGE_RECEIVED_data);
                break;
        default:
                break;
        }

}
```

The signature for the translator functions is unique to each event, but, rather than publishing typedefs for each, the functions are declared in the private header.

```
void hsmCommunicator_store_message(pHSM_COMMUNICATOR_DATA,pHSM_COMMUNICATOR_MESSAGE_RECEIVED_DATA);
```

Finally, we noted in Submachine Data Translation Function, that when parent and submachine have data, provision is made for a data translation function to be called when events are shared down to the submachine. It only remains, then, to declare those functions in the submachines. Unlike the top-level events, only the keyword `translator` is insufficient; FSMLang will not concoct a suitible name.

```
event parent::MESSAGE_RECEIVED data translator copy_current_message;
```

Sources

These pages show the full source of the fsm and generated c and h files.

hsmCommunicator.fsm

Download

```
native
{
#ifndef DBG_PRINTF
#include <stdio.h>
#define DBG_PRINTF(...) printf(__VA_ARGS__); printf("\n");
#endif

typedef enum
{
        msg_none
        , msg_step0_response
        , msg_step1_response
```

```
        , msg_ack
} msg_e_t;

typedef struct _queue_str_
{
        unsigned queue_count;
} queue_str_t;

}

/**
<p>This machine manages communications using a "stop and wait" protocol. Only one message is allowed to be outstanding.</p>
<p>Before any message can be exchanged, however, a session must be established with the peer. Establishing a connection
requires several exchanges to authenticate. The session will remain active as long as messages continue to be
exchanged with a minimum frequency.</p>
<p>The user of this machine calls run_hsmCommunicator, passing the SEND_MESSAGE event.  For the first message,
the machine will be IDLE, and thus needs to queue the message, start the establishSession machine, and transition
to the ESTABLISHING_SESSION state.  Requests to send messages received in this state will simply be queued. </p>
<p>While the top level machine is in the ESTABLISHING_SESSION state, the establishSession machine does the establishment work.</p>
<p>When the establishSession machine receives the STEP1_RESPONSE event, it reports to the top level machine that
the session is established by returning the parent's SESSION_ESTABLISHED event.  This will move the top level
machine to its IN_SESSION state and cause it to send the message(s) which are enqueued.</p>
*/
machine hsmCommunicator
native impl
{
    #define INIT_FSM_DATA {msg_none, { 0 }}
}
{
    data
    {
        msg_e_t current_msg;
        queue_str_t queue;
    }

    /** System initialization */
    event INIT;

    /** This event comes from our client code, asking us to send a message.
    */
    event SEND_MESSAGE;

    /** This event comes from our <i>establishSession</i> submachine, indicating that it has successfully
        completed its work.  We then forward it to our <i>sendMessage</i> submachine to indicate that
        it may now begin to send messages.
    */
    event SESSION_ESTABLISHED;

    /** This event comes from our external timer, indicating that we've not tickled it in a while, and
        thus should close down our session.
    */
    event SESSION_TIMEOUT;



    /** This event comes from our lower comm layers, indicating that a peer message has arrived.
        While we're in the ESTABLISHING_SESSION state, we forward this event to the <i>establishSession</i>
        submachine; while in the IN_SESSION state, we forward it to the <i>sendMessage</i> submachine.
    */
    event MESSAGE_RECEIVED
          data
          translator store_message
          {
             msg_e_t message;
          }
          ;

    /** The wakeup state. */
    state UNINITIALIZED;

    /** The first initialized state.  Also, this is the state to which the machine
        returns when a session times out.


    */
    state IDLE;

    /** The machine is establishing a session.  The actual work is being done by the <i>establishSession</i>
        submachine.  While in this state, the <i>MESSAGE_RECEIVED</i> event is forwarded to that submachine.
```

```
*/
state ESTABLISHING_SESSION;

/** A session has been established, and messages are being exchanged with the peer.  While in this
    state, the <i>MESSAGE_RECEIVED</i> event is forwarded to the <i>sendMessage</i> submachine.
*/
state IN_SESSION;

/**
<p>Establish a connection with the peer.
</p>
<p>Two messages must be exchanged with the peer to successfully establish the session.  The machine needs
only two states, IDLE and AWAITING_RESPONSE since the top level machine tracks whether or not it is in a
session.  The AWAITING_RESPONSE state serves for both required messages, since the receipt of each message produces
a unique event.
</p>
<p>When the STEP1_RESPONSE event is received, the session is considered established.  This machine will then
return the parent's SESSION_ESTABLISHED message and move to its IDLE state.
</p>
*/
machine establishSession
native impl
{
    #define INIT_FSM_DATA {msg_none}
}
{
data
{
    msg_e_t current_msg;
}

event ESTABLISH_SESSION_REQUEST, STEP0_RESPONSE, STEP1_RESPONSE;
event parent::MESSAGE_RECEIVED data translator copy_current_message;

state IDLE, AWAITING_RESPONSE;

/** Start the session establishment process. */
action sendStep0Message[ESTABLISH_SESSION_REQUEST, IDLE]  transition AWAITING_RESPONSE;

/** Continue session establisment */
action sendStep1Message[STEP0_RESPONSE, AWAITING_RESPONSE];

/** Notify parent that session is established. */
action notifyParent[STEP1_RESPONSE, AWAITING_RESPONSE] transition IDLE;

/** Parse the incoming message */
action parseMessage[MESSAGE_RECEIVED, AWAITING_RESPONSE];

/* these lines are informational; they affect the html output, but do not affect any C code generated. */
sendStep0Message returns noEvent;

sendStep1Message returns noEvent;


notifyParent     returns parent::SESSION_ESTABLISHED;

parseMessage returns STEP0_RESPONSE, STEP1_RESPONSE, noEvent;
}

/**
<p>Send a message to the peer.
</p>
<p>Since the protocol allows only one message to be outsanding, the machine dequeues and transmits a message only
from the IDLE state, transitioning to the AWAITING_ACK state immediately thereafter.
</p>
<p>In the AWAITNG_ACK state, incomming messages are parsed and, when an ACK is found, the machine checks the queue
and transitions to the IDLE state.  Checking the queue can return the SEND_MESSAGE event, which will be handled
from the IDLE state, thus resulting in a transmission and return to the AWAITING_ACK state.
</p>
*/
machine sendMessage
```

```
native impl
{
    #define INIT_FSM_DATA {msg_none, NULL}
}
{
```

```
    data
    {
        msg_e_t current_msg;
        queue_str_t *pqueue;
    }

    event parent::INIT data translator init_data;

    event parent::SEND_MESSAGE
                , parent::MESSAGE_RECEIVED data translator copy_current_message
                , ACK;

    state UNINITIALIZED, IDLE, AWAITING_ACK;

    transition [INIT, UNINITIALIZED] IDLE;

    /** Dequeue and transmit message to the peer. */
    action      sendMessage[SEND_MESSAGE,IDLE] transition AWAITING_ACK;

    /** Check queue for messages; if found return SEND_MESSAGE; otherwise, return noEvent. */
    action      checkQueue[ACK,AWAITING_ACK]          transition IDLE;

    action      parseMessage[MESSAGE_RECEIVED, AWAITING_ACK];

    /* these lines are informational; they affect the html output, but do not affect any C code generated. */
    sendMessage  returns noEvent;

    checkQueue   returns SEND_MESSAGE, noEvent;

    parseMessage returns ACK, noEvent;

    }

    /* these are actions of the top level machine */

    /** Initialize the machine */
    action initialize[INIT, UNINITIALIZED] transition IDLE;

    /** Start the session establishment process by activating the <i>establishSession</i> machine. */
    action startSessionEstablishment[SEND_MESSAGE, IDLE] transition  ESTABLISHING_SESSION;

    /** Start the session timer and notify the <i>sendMessage</i> machine that the session is established. */
    action completeSessionStart[SESSION_ESTABLISHED, ESTABLISHING_SESSION] transition IN_SESSION;

    /** Pass the MESSAGE_RECEIVED event along. */
    action passMessageReceived[MESSAGE_RECEIVED, (ESTABLISHING_SESSION, IN_SESSION)];

    /** Extend the session timer and queue the message */
    action queueMessage[SEND_MESSAGE, ESTABLISHING_SESSION];

    /** Extend the session timer and pass the message to be sent to the <i>sendMessage</i> machine. */
    action requestMessageTransmission[SEND_MESSAGE, IN_SESSION];

    transition [SESSION_TIMEOUT, IN_SESSION] IDLE;


    /* these lines are informational; they affect the html output, but do not affect any C code generated. */
    startSessionEstablishment   returns establishSession::ESTABLISH_SESSION_REQUEST;

    completeSessionStart        returns noEvent;

    requestMessageTransmission  returns noEvent;

    queueMessage                returns noEvent;
}
```

Data for Machines and Events

hsmCommunicator.c

```
/**
        hsmCommunicator.c

        This file automatically generated by FSMLang
*/

#include "hsmCommunicator_priv.h"
#include <stddef.h>

/* Begin Native Implementation Prolog */


    #define INIT_FSM_DATA {msg_none, { 0 }}

/* End Native Implementation Prolog */


#ifndef DBG_PRINTF
#define DBG_PRINTF(...)
#endif

extern HSM_COMMUNICATOR_SUB_FSM_IF hsmCommunicator_establishSession_sub_fsm_if;
extern HSM_COMMUNICATOR_SUB_FSM_IF hsmCommunicator_sendMessage_sub_fsm_if;

static HSM_COMMUNICATOR_EVENT_ENUM UNINITIALIZED_stateFn(pHSM_COMMUNICATOR,HSM_COMMUNICATOR_EVENT_ENUM);
static HSM_COMMUNICATOR_EVENT_ENUM IDLE_stateFn(pHSM_COMMUNICATOR,HSM_COMMUNICATOR_EVENT_ENUM);
static HSM_COMMUNICATOR_EVENT_ENUM ESTABLISHING_SESSION_stateFn(pHSM_COMMUNICATOR,HSM_COMMUNICATOR_EVENT_ENUM);
static HSM_COMMUNICATOR_EVENT_ENUM IN_SESSION_stateFn(pHSM_COMMUNICATOR,HSM_COMMUNICATOR_EVENT_ENUM);
static const HSM_COMMUNICATOR_STATE_FN hsmCommunicator_state_fn_array[hsmCommunicator_numStates] =
{
          UNINITIALIZED_stateFn
        , IDLE_stateFn
        , ESTABLISHING_SESSION_stateFn
        , IN_SESSION_stateFn
};


const pHSM_COMMUNICATOR_SUB_FSM_IF hsmCommunicator_sub_fsm_if_array[THIS(numSubMachines)] =
{
        &hsmCommunicator_establishSession_sub_fsm_if
        , &hsmCommunicator_sendMessage_sub_fsm_if
};

pHSM_COMMUNICATOR_SHARED_EVENT_STR sharing_hsmCommunicator_INIT[] =
{
          &sendMessage_share_hsmCommunicator_INIT_str
        , NULL
};

pHSM_COMMUNICATOR_SHARED_EVENT_STR sharing_hsmCommunicator_SEND_MESSAGE[] =
{


          &sendMessage_share_hsmCommunicator_SEND_MESSAGE_str
        , NULL
};

pHSM_COMMUNICATOR_SHARED_EVENT_STR sharing_hsmCommunicator_MESSAGE_RECEIVED[] =
{
          &establishSession_share_hsmCommunicator_MESSAGE_RECEIVED_str
        , &sendMessage_share_hsmCommunicator_MESSAGE_RECEIVED_str
        , NULL
};

HSM_COMMUNICATOR_EVENT_ENUM hsmCommunicator_pass_shared_event(pHSM_COMMUNICATOR pfsm,pHSM_COMMUNICATOR_SHARED_EVENT_STR sharer_list[])
{
        HSM_COMMUNICATOR_EVENT_ENUM return_event = THIS(noEvent);
        for (pHSM_COMMUNICATOR_SHARED_EVENT_STR *pcurrent_sharer = sharer_list;
             *pcurrent_sharer && return_event == THIS(noEvent);
             pcurrent_sharer++)
        {


            if ((*pcurrent_sharer)->data_translation_fn)
                    (*(*pcurrent_sharer)->data_translation_fn)(&pfsm->data);
            return_event = (*(*pcurrent_sharer)->psub_fsm_if->subFSM)((*pcurrent_sharer)->event);
        }
```

Data for Machines and Events

```
        return return_event;
}

#ifndef INIT_FSM_DATA
#error INIT_FSM_DATA must be defined
#endif

HSM_COMMUNICATOR hsmCommunicator = {

        INIT_FSM_DATA,
        hsmCommunicator_UNINITIALIZED,
        THIS(INIT),
        &hsmCommunicator_state_fn_array,
        &hsmCommunicator_sub_fsm_if_array,
        hsmCommunicatorFSM
};

pHSM_COMMUNICATOR phsmCommunicator = &hsmCommunicator;

void run_hsmCommunicator(pHSM_COMMUNICATOR_EVENT e)
{
        if (phsmCommunicator)
        {
                phsmCommunicator->fsm(phsmCommunicator,e);
        }
}

static HSM_COMMUNICATOR_EVENT_ENUM findAndRunSubMachine(pHSM_COMMUNICATOR,HSM_COMMUNICATOR_EVENT_ENUM);

static void translateEventData(pHSM_COMMUNICATOR_DATA,pHSM_COMMUNICATOR_EVENT);


#ifndef EVENT_IS_NOT_EXCLUDED_FROM_LOG
#define EVENT_IS_NOT_EXCLUDED_FROM_LOG(e) ((e) == (e))
#endif
void hsmCommunicatorFSM(pHSM_COMMUNICATOR pfsm, pHSM_COMMUNICATOR_EVENT event)
{
        HSM_COMMUNICATOR_EVENT_ENUM e = event->event;

        translateEventData(&pfsm->data, event);

        while (e != hsmCommunicator_noEvent) {

#ifdef HSM_COMMUNICATOR_DEBUG
if (EVENT_IS_NOT_EXCLUDED_FROM_LOG(e))
{
        DBG_PRINTF("event: %s; state: %s"
,HSM_COMMUNICATOR_EVENT_NAMES[e]
,HSM_COMMUNICATOR_STATE_NAMES[pfsm->state]
);
}
#endif


                /* This is read-only data to facilitate error reporting in action functions */
                pfsm->event = e;

                if (e < hsmCommunicator_noEvent)
                {
                        e = ((* (*pfsm->statesArray)[pfsm->state])(pfsm,e));
                }
                else
                {
                                e = findAndRunSubMachine(pfsm, e);
                }

        }
}


static HSM_COMMUNICATOR_EVENT_ENUM findAndRunSubMachine(pHSM_COMMUNICATOR pfsm, HSM_COMMUNICATOR_EVENT_ENUM e)
{
```

```
        for (HSM_COMMUNICATOR_SUB_MACHINES machineIterator = THIS(firstSubMachine);
             machineIterator < THIS(numSubMachines);
             machineIterator++
            )
        {
                        if (
```

Data for Machines and Events

```
                        ((*pfsm->subMachineArray)[machineIterator]->first_event <= e)
                        && ((*pfsm->subMachineArray)[machineIterator]->last_event > e)
                         )
                    {
                          return ((*(*pfsm->subMachineArray)[machineIterator]->subFSM)(e));
                    }
        }

        return THIS(noEvent);

}

static void translateEventData(pHSM_COMMUNICATOR_DATA pfsm_data, pHSM_COMMUNICATOR_EVENT pevent)
{
        switch(pevent->event)
        {
        case hsmCommunicator_MESSAGE_RECEIVED:
                UFMN(store_message)(pfsm_data, &pevent->event_data.MESSAGE_RECEIVED_data);
                break;
        default:
                break;
        }

}

static HSM_COMMUNICATOR_EVENT_ENUM UNINITIALIZED_stateFn(pHSM_COMMUNICATOR pfsm,HSM_COMMUNICATOR_EVENT_ENUM e)
{
        HSM_COMMUNICATOR_EVENT_ENUM retVal = THIS(noEvent);

        switch(e)
        {
        case THIS(INIT):
                retVal = UFMN(initialize)(pfsm);
                pfsm->state = hsmCommunicator_IDLE;
                break;
        default:
                DBG_PRINTF("hsmCommunicator_noAction");
                break;
        }

        return retVal;
}

static HSM_COMMUNICATOR_EVENT_ENUM IDLE_stateFn(pHSM_COMMUNICATOR pfsm,HSM_COMMUNICATOR_EVENT_ENUM e)
{
        HSM_COMMUNICATOR_EVENT_ENUM retVal = THIS(noEvent);

        switch(e)
        {
        case THIS(SEND_MESSAGE):
                retVal = UFMN(startSessionEstablishment)(pfsm);
                pfsm->state = hsmCommunicator_ESTABLISHING_SESSION;
                break;


        default:
                DBG_PRINTF("hsmCommunicator_noAction");
                break;
        }

        return retVal;
}

static HSM_COMMUNICATOR_EVENT_ENUM ESTABLISHING_SESSION_stateFn(pHSM_COMMUNICATOR pfsm,HSM_COMMUNICATOR_EVENT_ENUM e)
{
        HSM_COMMUNICATOR_EVENT_ENUM retVal = THIS(noEvent);

        switch(e)
        {
        case THIS(SESSION_ESTABLISHED):
                retVal = UFMN(completeSessionStart)(pfsm);
                pfsm->state = hsmCommunicator_IN_SESSION;
                break;


            case THIS(MESSAGE_RECEIVED):
                    retVal = UFMN(passMessageReceived)(pfsm);
                    break;
            case THIS(SEND_MESSAGE):
                    retVal = UFMN(queueMessage)(pfsm);
                    break;
```

```
        default:
                DBG_PRINTF("hsmCommunicator_noAction");
                break;
        }

        return retVal;
}

static HSM_COMMUNICATOR_EVENT_ENUM IN_SESSION_stateFn(pHSM_COMMUNICATOR pfsm,HSM_COMMUNICATOR_EVENT_ENUM e)
{
        HSM_COMMUNICATOR_EVENT_ENUM retVal = THIS(noEvent);

        switch(e)
        {
        case THIS(MESSAGE_RECEIVED):
                retVal = UFMN(passMessageReceived)(pfsm);
                break;
        case THIS(SEND_MESSAGE):
                retVal = UFMN(requestMessageTransmission)(pfsm);
                break;
        case THIS(SESSION_TIMEOUT):
                DBG_PRINTF("hsmCommunicator_noAction");
                pfsm->state = hsmCommunicator_IDLE;
                break;
        default:
                DBG_PRINTF("hsmCommunicator_noAction");
                break;
        }

        return retVal;
}



#ifdef HSM_COMMUNICATOR_DEBUG
char *HSM_COMMUNICATOR_EVENT_NAMES[] = {
         "hsmCommunicator_INIT"
        ,"hsmCommunicator_SEND_MESSAGE"
        ,"hsmCommunicator_SESSION_ESTABLISHED"
        ,"hsmCommunicator_SESSION_TIMEOUT"
        ,"hsmCommunicator_MESSAGE_RECEIVED"
        , "hsmCommunicator_noEvent"
        , "hsmCommunicator_numEvents"
        , "hsmCommunicator_establishSession_ESTABLISH_SESSION_REQUEST"
        , "hsmCommunicator_establishSession_STEP0_RESPONSE"
        , "hsmCommunicator_establishSession_STEP1_RESPONSE"
        , "hsmCommunicator_establishSession_MESSAGE_RECEIVED"
        , "hsmCommunicator_establishSession_noEvent"
        , "hsmCommunicator_sendMessage_INIT"
        , "hsmCommunicator_sendMessage_SEND_MESSAGE"
        , "hsmCommunicator_sendMessage_MESSAGE_RECEIVED"
        , "hsmCommunicator_sendMessage_ACK"
        , "hsmCommunicator_sendMessage_noEvent"
};

char *HSM_COMMUNICATOR_STATE_NAMES[] = {
         "hsmCommunicator_UNINITIALIZED"
        ,"hsmCommunicator_IDLE"
        ,"hsmCommunicator_ESTABLISHING_SESSION"
        ,"hsmCommunicator_IN_SESSION"
};

#endif
```

hsmCommunicator.h

```
/**
        hsmCommunicator.h

        This file automatically generated by FSMLang
*/

#ifndef _HSMCOMMUNICATOR_H_
#define _HSMCOMMUNICATOR_H_

#include "hsmCommunicator_events.h"
#ifndef HSM_COMMUNICATOR_NATIVE_PROLOG
#define HSM_COMMUNICATOR_NATIVE_PROLOG

#ifndef DBG_PRINTF
#include <stdio.h>
#define DBG_PRINTF(...) printf(__VA_ARGS__); printf("\n");
#endif

typedef enum
{
        msg_none
        , msg_step0_response
        , msg_step1_response
        , msg_ack
} msg_e_t;

typedef struct _queue_str_
{
        unsigned queue_count;
} queue_str_t;


#endif
#define FSM_VERSION "1.45.1"

#ifndef NO_CONVENIENCE_MACROS
#undef UFMN
#define UFMN(A) hsmCommunicator_##A
#undef THIS
#define THIS(A) hsmCommunicator_##A
#endif
#undef STATE
#define STATE(A) hsmCommunicator_##A
#undef HSM_COMMUNICATOR
#define HSM_COMMUNICATOR(A) hsmCommunicator_##A
#undef ESTABLISH_SESSION
#define ESTABLISH_SESSION(A) hsmCommunicator_establishSession_##A
#undef SEND_MESSAGE
#define SEND_MESSAGE(A) hsmCommunicator_sendMessage_##A

#undef ACTION_RETURN_TYPE
```

Data for Machines and Events

```
#define ACTION_RETURN_TYPE HSM_COMMUNICATOR_EVENT_ENUM
typedef struct _hsmCommunicator_MESSAGE_RECEIVED_data_ {
        msg_e_t message;
} HSM_COMMUNICATOR_MESSAGE_RECEIVED_DATA, *pHSM_COMMUNICATOR_MESSAGE_RECEIVED_DATA;

typedef union {
        HSM_COMMUNICATOR_MESSAGE_RECEIVED_DATA MESSAGE_RECEIVED_data;
} HSM_COMMUNICATOR_EVENT_DATA, *pHSM_COMMUNICATOR_EVENT_DATA;

typedef struct {
        HSM_COMMUNICATOR_EVENT_ENUM event;
        HSM_COMMUNICATOR_EVENT_DATA event_data;
} HSM_COMMUNICATOR_EVENT, *pHSM_COMMUNICATOR_EVENT;


void run_hsmCommunicator(pHSM_COMMUNICATOR_EVENT);

typedef struct _hsmCommunicator_struct_ *pHSM_COMMUNICATOR;
extern pHSM_COMMUNICATOR phsmCommunicator;


#endif
```

sendMessage_priv.h

```
/**
        sendMessage_priv.h

        This file automatically generated by FSMLang
*/

#ifndef _SENDMESSAGE_PRIV_H_
#define _SENDMESSAGE_PRIV_H_

#include "hsmCommunicator_submach.h"
#include "hsmCommunicator.h"

#ifdef SEND_MESSAGE_DEBUG
#include <stdio.h>
#include <stdlib.h>
#endif

/*
        sub-machine events are included in the top-level machine event enumeration.
        These macros set the appropriate names for events from THIS machine
        and those from the PARENT machine.

        They may be turned off as needed.
*/
#ifndef NO_CONVENIENCE_MACROS
#undef UFMN
#define UFMN(A) hsmCommunicator_sendMessage_##A
#undef THIS
#define THIS(A) hsmCommunicator_sendMessage_##A
#undef PARENT
#define PARENT(A) hsmCommunicator_##A
#endif
#undef STATE
#define STATE(A) sendMessage_##A

#ifdef HSM_COMMUNICATOR_SEND_MESSAGE_DEBUG
extern char *SEND_MESSAGE_EVENT_NAMES[];
extern char *SEND_MESSAGE_STATE_NAMES[];
#endif

typedef enum {
        sendMessage_UNINITIALIZED
        , sendMessage_IDLE
        , sendMessage_AWAITING_ACK
        , sendMessage_numStates
}  SEND_MESSAGE_STATE;

typedef struct _sendMessage_data_struct_ SEND_MESSAGE_DATA, *pSEND_MESSAGE_DATA;
typedef struct _sendMessage_struct_ SEND_MESSAGE, *pSEND_MESSAGE;
#undef FSM_TYPE_PTR
#define FSM_TYPE_PTR pSEND_MESSAGE
```

Data for Machines and Events

```
extern SEND_MESSAGE sendMessage;

extern pSEND_MESSAGE psendMessage;

typedef HSM_COMMUNICATOR_EVENT_ENUM (*SEND_MESSAGE_ACTION_FN)(FSM_TYPE_PTR);

typedef HSM_COMMUNICATOR_EVENT_ENUM (*SEND_MESSAGE_FSM)(FSM_TYPE_PTR,HSM_COMMUNICATOR_EVENT_ENUM);

struct _sendMessage_data_struct_ {
        msg_e_t current_msg;
        queue_str_t * pqueue;
};

typedef ACTION_RETURN_TYPE (*SEND_MESSAGE_STATE_FN)(pSEND_MESSAGE,HSM_COMMUNICATOR_EVENT_ENUM);

static const SEND_MESSAGE_STATE_FN sendMessage_state_fn_array[sendMessage_numStates];

struct _sendMessage_struct_ {
        SEND_MESSAGE_DATA             data;
        SEND_MESSAGE_STATE            state;
        HSM_COMMUNICATOR_EVENT_ENUM   event;
        SEND_MESSAGE_STATE_FN   const (*statesArray)[sendMessage_numStates];
        SEND_MESSAGE_FSM              fsm;
};


ACTION_RETURN_TYPE hsmCommunicator_sendMessage_sendMessage(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_sendMessage_checkQueue(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_sendMessage_parseMessage(FSM_TYPE_PTR);

void hsmCommunicator_sendMessage_init_data(pHSM_COMMUNICATOR_DATA);
void hsmCommunicator_sendMessage_copy_current_message(pHSM_COMMUNICATOR_DATA);

#endif
```

hsmCommunicator_priv.h

```
/**
        hsmCommunicator_priv.h

        This file automatically generated by FSMLang
*/

#ifndef _HSMCOMMUNICATOR_PRIV_H_
#define _HSMCOMMUNICATOR_PRIV_H_

#include "hsmCommunicator.h"
#ifndef HSM_COMMUNICATOR_NATIVE_PROLOG
#define HSM_COMMUNICATOR_NATIVE_PROLOG

#ifndef DBG_PRINTF
#include <stdio.h>
#define DBG_PRINTF(...) printf(__VA_ARGS__); printf("\n");
#endif

typedef enum
{
        msg_none
        , msg_step0_response
        , msg_step1_response
        , msg_ack
} msg_e_t;

typedef struct _queue_str_
{
        unsigned queue_count;
} queue_str_t;


#endif

#ifdef HSM_COMMUNICATOR_DEBUG
#include <stdio.h>
#include <stdlib.h>
#endif


#ifdef HSM_COMMUNICATOR_DEBUG
extern char *HSM_COMMUNICATOR_EVENT_NAMES[];
extern char *HSM_COMMUNICATOR_STATE_NAMES[];
#endif

typedef enum {
        hsmCommunicator_UNINITIALIZED
        , hsmCommunicator_IDLE
        , hsmCommunicator_ESTABLISHING_SESSION
        , hsmCommunicator_IN_SESSION
        , hsmCommunicator_numStates
```

```
}  HSM_COMMUNICATOR_STATE;

typedef struct _hsmCommunicator_struct_ HSM_COMMUNICATOR;
#undef FSM_TYPE_PTR
#define FSM_TYPE_PTR pHSM_COMMUNICATOR
extern HSM_COMMUNICATOR hsmCommunicator;

typedef HSM_COMMUNICATOR_EVENT_ENUM (*HSM_COMMUNICATOR_ACTION_FN)(FSM_TYPE_PTR);

typedef void (*HSM_COMMUNICATOR_FSM)(FSM_TYPE_PTR,pHSM_COMMUNICATOR_EVENT);

void hsmCommunicatorFSM(FSM_TYPE_PTR,pHSM_COMMUNICATOR_EVENT);

#include "hsmCommunicator_submach.h"
typedef ACTION_RETURN_TYPE (*HSM_COMMUNICATOR_STATE_FN)(pHSM_COMMUNICATOR,HSM_COMMUNICATOR_EVENT_ENUM);

static const HSM_COMMUNICATOR_STATE_FN hsmCommunicator_state_fn_array[hsmCommunicator_numStates];

struct _hsmCommunicator_struct_ {
        HSM_COMMUNICATOR_DATA               data;
        HSM_COMMUNICATOR_STATE              state;
        HSM_COMMUNICATOR_EVENT_ENUM         event;
        HSM_COMMUNICATOR_STATE_FN     const (*statesArray)[hsmCommunicator_numStates];
        pHSM_COMMUNICATOR_SUB_FSM_IF   const (*subMachineArray)[hsmCommunicator_numSubMachines];
        HSM_COMMUNICATOR_FSM                fsm;
};


ACTION_RETURN_TYPE hsmCommunicator_initialize(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_startSessionEstablishment(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_completeSessionStart(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_passMessageReceived(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_queueMessage(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_requestMessageTransmission(FSM_TYPE_PTR);


void hsmCommunicator_store_message(pHSM_COMMUNICATOR_DATA,pHSM_COMMUNICATOR_MESSAGE_RECEIVED_DATA);


#endif
```

Data for Machines and Events

hsmCommunicator_submach.h

```
/**
        hsmCommunicator_submach.h

        This file automatically generated by FSMLang
*/

#ifndef _HSMCOMMUNICATOR_SUBMACH_H_
#define _HSMCOMMUNICATOR_SUBMACH_H_

#include "hsmCommunicator.h"
#ifndef HSM_COMMUNICATOR_NATIVE_PROLOG
#define HSM_COMMUNICATOR_NATIVE_PROLOG

#ifndef DBG_PRINTF
#include <stdio.h>
#define DBG_PRINTF(...) printf(__VA_ARGS__); printf("\n");
#endif

typedef enum
{
        msg_none
        , msg_step0_response
        , msg_step1_response
        , msg_ack
} msg_e_t;

typedef struct _queue_str_
{
        unsigned queue_count;
} queue_str_t;


#endif
typedef struct _hsmCommunicator_data_struct_ HSM_COMMUNICATOR_DATA, *pHSM_COMMUNICATOR_DATA;
/* Sub Machine Declarations */

typedef enum {
         establishSession_e
        , hsmCommunicator_firstSubMachine = establishSession_e
        ,  sendMessage_e
        , hsmCommunicator_numSubMachines
} HSM_COMMUNICATOR_SUB_MACHINES;

typedef HSM_COMMUNICATOR_EVENT_ENUM (*HSM_COMMUNICATOR_SUB_MACHINE_FN)(HSM_COMMUNICATOR_EVENT_ENUM);
typedef struct _hsmCommunicator_sub_fsm_if_ HSM_COMMUNICATOR_SUB_FSM_IF, *pHSM_COMMUNICATOR_SUB_FSM_IF;
struct _hsmCommunicator_sub_fsm_if_
{
        HSM_COMMUNICATOR_EVENT_ENUM      first_event;
        HSM_COMMUNICATOR_EVENT_ENUM      last_event;
        HSM_COMMUNICATOR_SUB_MACHINE_FN  subFSM;
};


typedef void (*HSM_COMMUNICATOR_DATA_TRANSLATION_FN)(pHSM_COMMUNICATOR_DATA);
/* Some sub-machines share parent events. */
typedef struct _hsmCommunicator_shared_event_str_ HSM_COMMUNICATOR_SHARED_EVENT_STR, *pHSM_COMMUNICATOR_SHARED_EVENT_STR;
struct _hsmCommunicator_shared_event_str_
{
        HSM_COMMUNICATOR_EVENT_ENUM           event;
        HSM_COMMUNICATOR_DATA_TRANSLATION_FN  data_translation_fn;
        pHSM_COMMUNICATOR_SUB_FSM_IF          psub_fsm_if;
};
extern HSM_COMMUNICATOR_EVENT_ENUM hsmCommunicator_pass_shared_event(pHSM_COMMUNICATOR,pHSM_COMMUNICATOR_SHARED_EVENT_STR[]);

extern HSM_COMMUNICATOR_SHARED_EVENT_STR sendMessage_share_hsmCommunicator_INIT_str;
extern pHSM_COMMUNICATOR_SHARED_EVENT_STR sharing_hsmCommunicator_INIT[];

extern HSM_COMMUNICATOR_SHARED_EVENT_STR sendMessage_share_hsmCommunicator_SEND_MESSAGE_str;


extern pHSM_COMMUNICATOR_SHARED_EVENT_STR sharing_hsmCommunicator_SEND_MESSAGE[];

extern HSM_COMMUNICATOR_SHARED_EVENT_STR establishSession_share_hsmCommunicator_MESSAGE_RECEIVED_str;
extern HSM_COMMUNICATOR_SHARED_EVENT_STR sendMessage_share_hsmCommunicator_MESSAGE_RECEIVED_str;
```

```
extern pHSM_COMMUNICATOR_SHARED_EVENT_STR sharing_hsmCommunicator_MESSAGE_RECEIVED[];


struct _hsmCommunicator_data_struct_ {
        msg_e_t current_msg;
        queue_str_t queue;
};


#endif
```

hsmc-actions.c

Download

```
/*
  hsmc-actions.c

        Action functions for the hsmCommunicator top-level FSM.

    FSMLang (fsm) - A Finite State Machine description language.
    Copyright (C) 2024  Steven Stanton

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

    Steven Stanton
    sstanton@pesticidesoftware.com

    For the latest on FSMLang: https://fsmlang.github.io/

    And, finally, your possession of this source code implies nothing.

*/

#include "hsmCommunicator_priv.h"

ACTION_RETURN_TYPE  UFMN(initialize)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);

    return hsmCommunicator_pass_shared_event(pfsm, sharing_hsmCommunicator_INIT);
```

```
}

ACTION_RETURN_TYPE  UFMN(startSessionEstablishment)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);

    pfsm->data.queue.queue_count++;

    return ESTABLISH_SESSION(ESTABLISH_SESSION_REQUEST);
}

ACTION_RETURN_TYPE  UFMN(completeSessionStart)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    return hsmCommunicator_pass_shared_event(pfsm, sharing_hsmCommunicator_SEND_MESSAGE);
}

ACTION_RETURN_TYPE  UFMN(passMessageReceived)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    return hsmCommunicator_pass_shared_event(pfsm, sharing_hsmCommunicator_MESSAGE_RECEIVED);
}

ACTION_RETURN_TYPE  UFMN(queueMessage)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);

    pfsm->data.queue.queue_count++;

    return THIS(noEvent);
}

ACTION_RETURN_TYPE  UFMN(requestMessageTransmission)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);

    pfsm->data.queue.queue_count++;

    return hsmCommunicator_pass_shared_event(pfsm, sharing_hsmCommunicator_SEND_MESSAGE);
}

ACTION_RETURN_TYPE  UFMN(noAction)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}

void hsmCommunicator_store_message(pHSM_COMMUNICATOR_DATA pfsm_data, pHSM_COMMUNICATOR_MESSAGE_RECEIVED_DATA pedata)
{
    DBG_PRINTF("%s", __func__);

        pfsm_data->current_msg = pedata->message;


}

int main(void)
{
     HSM_COMMUNICATOR_EVENT event;

     event.event = THIS(INIT);
     run_hsmCommunicator(&event);

     event.event = THIS(SEND_MESSAGE);
     run_hsmCommunicator(&event);

     event.event = THIS(MESSAGE_RECEIVED);
```

```
          event.event_data.MESSAGE_RECEIVED_data.message = msg_step0_response;
     run_hsmCommunicator(&event);

     event.event = THIS(SEND_MESSAGE);
     run_hsmCommunicator(&event);

     event.event = THIS(MESSAGE_RECEIVED);
          event.event_data.MESSAGE_RECEIVED_data.message = msg_step1_response;
     run_hsmCommunicator(&event);

     event.event = THIS(MESSAGE_RECEIVED);
          event.event_data.MESSAGE_RECEIVED_data.message = msg_ack;
     run_hsmCommunicator(&event);

     event.event = THIS(MESSAGE_RECEIVED);
          event.event_data.MESSAGE_RECEIVED_data.message = msg_ack;
     run_hsmCommunicator(&event);

     return 0;
}
```

Data for Machines and Events

establishSession.c

```
/**
        establishSession.c

        This file automatically generated by FSMLang
*/

#include "establishSession_priv.h"
#include <stddef.h>

/* Begin Native Implementation Prolog */


        #define INIT_FSM_DATA {msg_none}

/* End Native Implementation Prolog */


#ifndef DBG_PRINTF
#define DBG_PRINTF(...)
#endif

static HSM_COMMUNICATOR_EVENT_ENUM establishSessionFSM(FSM_TYPE_PTR,HSM_COMMUNICATOR_EVENT_ENUM);

static HSM_COMMUNICATOR_EVENT_ENUM IDLE_stateFn(pESTABLISH_SESSION,HSM_COMMUNICATOR_EVENT_ENUM);
static HSM_COMMUNICATOR_EVENT_ENUM AWAITING_RESPONSE_stateFn(pESTABLISH_SESSION,HSM_COMMUNICATOR_EVENT_ENUM);

static const ESTABLISH_SESSION_STATE_FN establishSession_state_fn_array[establishSession_numStates] =
{
          IDLE_stateFn
        , AWAITING_RESPONSE_stateFn
};

HSM_COMMUNICATOR_EVENT_ENUM THIS(sub_machine_fn)(HSM_COMMUNICATOR_EVENT_ENUM e)
{
        return establishSessionFSM(pestablishSession,e);
}

HSM_COMMUNICATOR_SUB_FSM_IF hsmCommunicator_establishSession_sub_fsm_if =
{
                .subFSM = THIS(sub_machine_fn)
        , .first_event = THIS(firstEvent)
        , .last_event = THIS(noEvent)
};

HSM_COMMUNICATOR_SHARED_EVENT_STR establishSession_share_hsmCommunicator_MESSAGE_RECEIVED_str = {
          .event               = THIS(MESSAGE_RECEIVED)
        , .data_translation_fn = UFMN(copy_current_message)
        , .psub_fsm_if         = &hsmCommunicator_establishSession_sub_fsm_if
};

#ifndef INIT_FSM_DATA

#error INIT_FSM_DATA must be defined
#endif

ESTABLISH_SESSION establishSession = {

         INIT_FSM_DATA,
         establishSession_IDLE,
         THIS(ESTABLISH_SESSION_REQUEST),
         &establishSession_state_fn_array,
         establishSessionFSM
};
```

```
pESTABLISH_SESSION pestablishSession = &establishSession;


#ifndef EVENT_IS_NOT_EXCLUDED_FROM_LOG
#define EVENT_IS_NOT_EXCLUDED_FROM_LOG(e) (e == e)
#endif
HSM_COMMUNICATOR_EVENT_ENUM establishSessionFSM(pESTABLISH_SESSION pfsm, HSM_COMMUNICATOR_EVENT_ENUM event)
{
        HSM_COMMUNICATOR_EVENT_ENUM e = event;

        while ((e != THIS(noEvent))
               && (e >= THIS(firstEvent))
              )
        {

#ifdef HSM_COMMUNICATOR_ESTABLISH_SESSION_DEBUG
if ((EVENT_IS_NOT_EXCLUDED_FROM_LOG(e))
    && (e >= THIS(firstEvent))
    && (e < THIS(noEvent))
   )
{
        DBG_PRINTF("event: %s; state: %s"
,ESTABLISH_SESSION_EVENT_NAMES[e - THIS(firstEvent)]
,ESTABLISH_SESSION_STATE_NAMES[pfsm->state]
);
}
#endif

                /* This is read-only data to facilitate error reporting in action functions */
                pfsm->event = e;

                if ((e >= THIS(firstEvent))
                    && (e < THIS(noEvent))
                        )
                {
                e = ((* (*pfsm->statesArray)[pfsm->state])(pfsm,e));
                }

        }

        return e == THIS(noEvent) ? PARENT(noEvent) : e;
}

static HSM_COMMUNICATOR_EVENT_ENUM IDLE_stateFn(pESTABLISH_SESSION pfsm,HSM_COMMUNICATOR_EVENT_ENUM e)
{
        HSM_COMMUNICATOR_EVENT_ENUM retVal = THIS(noEvent);

        switch(e)
        {
        case THIS(ESTABLISH_SESSION_REQUEST):
                retVal = UFMN(sendStep0Message)(pfsm);
                pfsm->state = establishSession_AWAITING_RESPONSE;
                break;


        default:
                DBG_PRINTF("hsmCommunicator_establishSession_noAction");
                break;
        }

        return retVal;
}

static HSM_COMMUNICATOR_EVENT_ENUM AWAITING_RESPONSE_stateFn(pESTABLISH_SESSION pfsm,HSM_COMMUNICATOR_EVENT_ENUM e)
{
        HSM_COMMUNICATOR_EVENT_ENUM retVal = THIS(noEvent);

        switch(e)

          {
          case THIS(STEP0_RESPONSE):
```

```
                retVal = UFMN(sendStep1Message)(pfsm);
                break;
        case THIS(STEP1_RESPONSE):
                retVal = UFMN(notifyParent)(pfsm);
                pfsm->state = establishSession_IDLE;
                break;
        case THIS(MESSAGE_RECEIVED):
                retVal = UFMN(parseMessage)(pfsm);
                break;
        default:
                DBG_PRINTF("hsmCommunicator_establishSession_noAction");
                break;
        }

        return retVal;
}


#ifdef HSM_COMMUNICATOR_ESTABLISH_SESSION_DEBUG
char *ESTABLISH_SESSION_EVENT_NAMES[] = {
         "hsmCommunicator_establishSession_ESTABLISH_SESSION_REQUEST"
        ,"hsmCommunicator_establishSession_STEP0_RESPONSE"
        ,"hsmCommunicator_establishSession_STEP1_RESPONSE"
        ,"hsmCommunicator_establishSession_MESSAGE_RECEIVED"
        , "establishSession_noEvent"
        , "establishSession_numEvents"
};

char *ESTABLISH_SESSION_STATE_NAMES[] = {
         "hsmCommunicator_establishSession_IDLE"
        ,"hsmCommunicator_establishSession_AWAITING_RESPONSE"
};

#endif
```

establishSession_priv.h

```
/**
        establishSession_priv.h

        This file automatically generated by FSMLang
*/

#ifndef _ESTABLISHSESSION_PRIV_H_
#define _ESTABLISHSESSION_PRIV_H_

#include "hsmCommunicator_submach.h"
#include "hsmCommunicator.h"

#ifdef ESTABLISH_SESSION_DEBUG
#include <stdio.h>
#include <stdlib.h>
#endif

/*
        sub-machine events are included in the top-level machine event enumeration.
        These macros set the appropriate names for events from THIS machine
        and those from the PARENT machine.

        They may be turned off as needed.
*/
#ifndef NO_CONVENIENCE_MACROS
#undef UFMN
#define UFMN(A) hsmCommunicator_establishSession_##A
#undef THIS
#define THIS(A) hsmCommunicator_establishSession_##A
#undef PARENT
#define PARENT(A) hsmCommunicator_##A
#endif
#undef STATE
#define STATE(A) establishSession_##A

#ifdef HSM_COMMUNICATOR_ESTABLISH_SESSION_DEBUG
extern char *ESTABLISH_SESSION_EVENT_NAMES[];
extern char *ESTABLISH_SESSION_STATE_NAMES[];
#endif

typedef enum {
        establishSession_IDLE
        , establishSession_AWAITING_RESPONSE
        , establishSession_numStates
}  ESTABLISH_SESSION_STATE;

typedef struct _establishSession_data_struct_ ESTABLISH_SESSION_DATA, *pESTABLISH_SESSION_DATA;
typedef struct _establishSession_struct_ ESTABLISH_SESSION, *pESTABLISH_SESSION;
#undef FSM_TYPE_PTR
#define FSM_TYPE_PTR pESTABLISH_SESSION
extern ESTABLISH_SESSION establishSession;


extern pESTABLISH_SESSION pestablishSession;

typedef HSM_COMMUNICATOR_EVENT_ENUM (*ESTABLISH_SESSION_ACTION_FN)(FSM_TYPE_PTR);

typedef HSM_COMMUNICATOR_EVENT_ENUM (*ESTABLISH_SESSION_FSM)(FSM_TYPE_PTR,HSM_COMMUNICATOR_EVENT_ENUM);

struct _establishSession_data_struct_ {
        msg_e_t current_msg;
```

```
};

typedef ACTION_RETURN_TYPE (*ESTABLISH_SESSION_STATE_FN)(pESTABLISH_SESSION,HSM_COMMUNICATOR_EVENT_ENUM);

static const ESTABLISH_SESSION_STATE_FN establishSession_state_fn_array[establishSession_numStates];

struct _establishSession_struct_ {
        ESTABLISH_SESSION_DATA             data;
        ESTABLISH_SESSION_STATE            state;
        HSM_COMMUNICATOR_EVENT_ENUM        event;
        ESTABLISH_SESSION_STATE_FN   const (*statesArray)[establishSession_numStates];
        ESTABLISH_SESSION_FSM              fsm;
};


ACTION_RETURN_TYPE hsmCommunicator_establishSession_sendStep0Message(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_establishSession_sendStep1Message(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_establishSession_notifyParent(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_establishSession_parseMessage(FSM_TYPE_PTR);

void hsmCommunicator_establishSession_copy_current_message(pHSM_COMMUNICATOR_DATA);

#endif
```

session-actions.c

Download

```
/*
  session_actions.c

        Action functions for the establishSession FSM.

        FSMLang (fsm) - A Finite State Machine description language.
        Copyright (C) 4/20/2025  Steven Stanton

        This program is free software; you can redistribute it and/or modify
        it under the terms of the GNU General Public License as published by
        the Free Software Foundation; either version 2 of the License, or
        (at your option) any later version.

        This program is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
        GNU General Public License for more details.

        You should have received a copy of the GNU General Public License
        along with this program; if not, write to the Free Software
        Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

        Steven Stanton
        sstanton@pesticidesoftware.com

        For the latest on FSMLang: http://fsmlang.github.io

        And, finally, your possession of this source code implies nothing.

*/
```

Data for Machines and Events

```
#include <stdbool.h>
#include "establishSession_priv.h"

ACTION_RETURN_TYPE UFMN(sendStep0Message)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}

ACTION_RETURN_TYPE UFMN(sendStep1Message)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}

ACTION_RETURN_TYPE UFMN(notifyParent)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return PARENT(SESSION_ESTABLISHED);
}

ACTION_RETURN_TYPE UFMN(parseMessage)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;

    return pfsm->data.current_msg == msg_step0_response ? THIS(STEP0_RESPONSE) : THIS(STEP1_RESPONSE);
}

ACTION_RETURN_TYPE UFMN(noAction)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}

void UFMN(copy_current_message)(pHSM_COMMUNICATOR_DATA pfsm_data)
{
    DBG_PRINTF("%s", __func__);

        pestablishSession->data.current_msg = pfsm_data->current_msg;
}
```

Data for Machines and Events

sendMessage.c

```
/**
        sendMessage.c

        This file automatically generated by FSMLang
*/

#include "sendMessage_priv.h"
#include <stddef.h>

/* Begin Native Implementation Prolog */


        #define INIT_FSM_DATA {msg_none, NULL}

/* End Native Implementation Prolog */


#ifndef DBG_PRINTF
#define DBG_PRINTF(...)
#endif

static HSM_COMMUNICATOR_EVENT_ENUM sendMessageFSM(FSM_TYPE_PTR,HSM_COMMUNICATOR_EVENT_ENUM);

static HSM_COMMUNICATOR_EVENT_ENUM UNINITIALIZED_stateFn(pSEND_MESSAGE,HSM_COMMUNICATOR_EVENT_ENUM);
static HSM_COMMUNICATOR_EVENT_ENUM IDLE_stateFn(pSEND_MESSAGE,HSM_COMMUNICATOR_EVENT_ENUM);
static HSM_COMMUNICATOR_EVENT_ENUM AWAITING_ACK_stateFn(pSEND_MESSAGE,HSM_COMMUNICATOR_EVENT_ENUM);

static const SEND_MESSAGE_STATE_FN sendMessage_state_fn_array[sendMessage_numStates] =
{
          UNINITIALIZED_stateFn
        , IDLE_stateFn
        , AWAITING_ACK_stateFn
};

HSM_COMMUNICATOR_EVENT_ENUM THIS(sub_machine_fn)(HSM_COMMUNICATOR_EVENT_ENUM e)
{
        return sendMessageFSM(psendMessage,e);
}

HSM_COMMUNICATOR_SUB_FSM_IF hsmCommunicator_sendMessage_sub_fsm_if =
{
                .subFSM = THIS(sub_machine_fn)
        , .first_event = THIS(firstEvent)
        , .last_event = THIS(noEvent)
};

HSM_COMMUNICATOR_SHARED_EVENT_STR sendMessage_share_hsmCommunicator_INIT_str = {
          .event               = THIS(INIT)
        , .data_translation_fn = UFMN(init_data)
        , .psub_fsm_if         = &hsmCommunicator_sendMessage_sub_fsm_if
};


HSM_COMMUNICATOR_SHARED_EVENT_STR sendMessage_share_hsmCommunicator_SEND_MESSAGE_str = {
          .event               = THIS(SEND_MESSAGE)
        , .psub_fsm_if         = &hsmCommunicator_sendMessage_sub_fsm_if
};

HSM_COMMUNICATOR_SHARED_EVENT_STR sendMessage_share_hsmCommunicator_MESSAGE_RECEIVED_str = {
          .event               = THIS(MESSAGE_RECEIVED)
        , .data_translation_fn = UFMN(copy_current_message)
```

Data for Machines and Events

```
        , .psub_fsm_if         = &hsmCommunicator_sendMessage_sub_fsm_if
};

#ifndef INIT_FSM_DATA
#error INIT_FSM_DATA must be defined
#endif

SEND_MESSAGE sendMessage = {

        INIT_FSM_DATA,
        sendMessage_UNINITIALIZED,
        THIS(INIT),
        &sendMessage_state_fn_array,
        sendMessageFSM
};

pSEND_MESSAGE psendMessage = &sendMessage;


#ifndef EVENT_IS_NOT_EXCLUDED_FROM_LOG
#define EVENT_IS_NOT_EXCLUDED_FROM_LOG(e) (e == e)
#endif
HSM_COMMUNICATOR_EVENT_ENUM sendMessageFSM(pSEND_MESSAGE pfsm, HSM_COMMUNICATOR_EVENT_ENUM event)
{
        HSM_COMMUNICATOR_EVENT_ENUM e = event;

        while ((e != THIS(noEvent))
               && (e >= THIS(firstEvent))
              )
        {

#ifdef HSM_COMMUNICATOR_SEND_MESSAGE_DEBUG
if ((EVENT_IS_NOT_EXCLUDED_FROM_LOG(e))
    && (e >= THIS(firstEvent))
    && (e < THIS(noEvent))
   )
{
        DBG_PRINTF("event: %s; state: %s"
,SEND_MESSAGE_EVENT_NAMES[e - THIS(firstEvent)]
,SEND_MESSAGE_STATE_NAMES[pfsm->state]
);
}
#endif

                /* This is read-only data to facilitate error reporting in action functions */
                pfsm->event = e;

                if ((e >= THIS(firstEvent))
                    && (e < THIS(noEvent))
                        )
                {
                e = ((* (*pfsm->statesArray)[pfsm->state])(pfsm,e));
                }


        }

        return e == THIS(noEvent) ? PARENT(noEvent) : e;
}

static HSM_COMMUNICATOR_EVENT_ENUM UNINITIALIZED_stateFn(pSEND_MESSAGE pfsm,HSM_COMMUNICATOR_EVENT_ENUM e)
{
        HSM_COMMUNICATOR_EVENT_ENUM retVal = THIS(noEvent);
```

```
        switch(e)
        {
        case THIS(INIT):
                DBG_PRINTF("hsmCommunicator_sendMessage_noAction");
                pfsm->state = sendMessage_IDLE;
                break;
        default:
                DBG_PRINTF("hsmCommunicator_sendMessage_noAction");
                break;
        }

        return retVal;
}

static HSM_COMMUNICATOR_EVENT_ENUM IDLE_stateFn(pSEND_MESSAGE pfsm,HSM_COMMUNICATOR_EVENT_ENUM e)
{
        HSM_COMMUNICATOR_EVENT_ENUM retVal = THIS(noEvent);

        switch(e)
        {
        case THIS(SEND_MESSAGE):
                retVal = UFMN(sendMessage)(pfsm);
                pfsm->state = sendMessage_AWAITING_ACK;
                break;
        default:
                DBG_PRINTF("hsmCommunicator_sendMessage_noAction");
                break;
        }

        return retVal;
}

static HSM_COMMUNICATOR_EVENT_ENUM AWAITING_ACK_stateFn(pSEND_MESSAGE pfsm,HSM_COMMUNICATOR_EVENT_ENUM e)
{
        HSM_COMMUNICATOR_EVENT_ENUM retVal = THIS(noEvent);

        switch(e)
        {
        case THIS(ACK):
                retVal = UFMN(checkQueue)(pfsm);
                pfsm->state = sendMessage_IDLE;
                break;
        case THIS(MESSAGE_RECEIVED):
                retVal = UFMN(parseMessage)(pfsm);
                break;
        default:
                DBG_PRINTF("hsmCommunicator_sendMessage_noAction");
                break;
        }

        return retVal;
}
```

```
#ifdef HSM_COMMUNICATOR_SEND_MESSAGE_DEBUG
char *SEND_MESSAGE_EVENT_NAMES[] = {
          "hsmCommunicator_sendMessage_INIT"
        ,"hsmCommunicator_sendMessage_SEND_MESSAGE"
        ,"hsmCommunicator_sendMessage_MESSAGE_RECEIVED"
        ,"hsmCommunicator_sendMessage_ACK"
```

```
        , "sendMessage_noEvent"
        , "sendMessage_numEvents"
};

char *SEND_MESSAGE_STATE_NAMES[] = {
         "hsmCommunicator_sendMessage_UNINITIALIZED"
        ,"hsmCommunicator_sendMessage_IDLE"
        ,"hsmCommunicator_sendMessage_AWAITING_ACK"
};

#endif
```

sendMessage_priv.h

```
/**
        sendMessage_priv.h

        This file automatically generated by FSMLang
*/

#ifndef _SENDMESSAGE_PRIV_H_
#define _SENDMESSAGE_PRIV_H_

#include "hsmCommunicator_submach.h"
#include "hsmCommunicator.h"

#ifdef SEND_MESSAGE_DEBUG
#include <stdio.h>
#include <stdlib.h>
#endif

/*
        sub-machine events are included in the top-level machine event enumeration.
        These macros set the appropriate names for events from THIS machine
        and those from the PARENT machine.

        They may be turned off as needed.
*/
#ifndef NO_CONVENIENCE_MACROS
#undef UFMN
#define UFMN(A) hsmCommunicator_sendMessage_##A
#undef THIS
#define THIS(A) hsmCommunicator_sendMessage_##A
#undef PARENT
#define PARENT(A) hsmCommunicator_##A
#endif
#undef STATE
#define STATE(A) sendMessage_##A

#ifdef HSM_COMMUNICATOR_SEND_MESSAGE_DEBUG
extern char *SEND_MESSAGE_EVENT_NAMES[];
extern char *SEND_MESSAGE_STATE_NAMES[];
#endif

typedef enum {
        sendMessage_UNINITIALIZED
        , sendMessage_IDLE
        , sendMessage_AWAITING_ACK
        , sendMessage_numStates
}  SEND_MESSAGE_STATE;

typedef struct _sendMessage_data_struct_ SEND_MESSAGE_DATA, *pSEND_MESSAGE_DATA;
typedef struct _sendMessage_struct_ SEND_MESSAGE, *pSEND_MESSAGE;
#undef FSM_TYPE_PTR
#define FSM_TYPE_PTR pSEND_MESSAGE
```

```
extern SEND_MESSAGE sendMessage;

extern pSEND_MESSAGE psendMessage;

typedef HSM_COMMUNICATOR_EVENT_ENUM (*SEND_MESSAGE_ACTION_FN)(FSM_TYPE_PTR);

typedef HSM_COMMUNICATOR_EVENT_ENUM (*SEND_MESSAGE_FSM)(FSM_TYPE_PTR,HSM_COMMUNICATOR_EVENT_ENUM);

struct _sendMessage_data_struct_ {
        msg_e_t current_msg;
        queue_str_t * pqueue;
};

typedef ACTION_RETURN_TYPE (*SEND_MESSAGE_STATE_FN)(pSEND_MESSAGE,HSM_COMMUNICATOR_EVENT_ENUM);

static const SEND_MESSAGE_STATE_FN sendMessage_state_fn_array[sendMessage_numStates];

struct _sendMessage_struct_ {
        SEND_MESSAGE_DATA             data;
        SEND_MESSAGE_STATE            state;
        HSM_COMMUNICATOR_EVENT_ENUM   event;
        SEND_MESSAGE_STATE_FN   const (*statesArray)[sendMessage_numStates];
        SEND_MESSAGE_FSM              fsm;
};


ACTION_RETURN_TYPE hsmCommunicator_sendMessage_sendMessage(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_sendMessage_checkQueue(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_sendMessage_parseMessage(FSM_TYPE_PTR);

void hsmCommunicator_sendMessage_init_data(pHSM_COMMUNICATOR_DATA);
void hsmCommunicator_sendMessage_copy_current_message(pHSM_COMMUNICATOR_DATA);

#endif
```

message-actions.c

Download

```
/*
  message_actions.c

        Action functions for the sendMessage FSM.

        FSMLang (fsm) - A Finite State Machine description language.
        Copyright (C) 4/20/2025  Steven Stanton

        This program is free software; you can redistribute it and/or modify
        it under the terms of the GNU General Public License as published by
        the Free Software Foundation; either version 2 of the License, or
        (at your option) any later version.

        This program is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
        GNU General Public License for more details.

        You should have received a copy of the GNU General Public License
```

```
        along with this program; if not, write to the Free Software
        Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

        Steven Stanton
        sstanton@pesticidesoftware.com

        For the latest on FSMLang: http://fsmlang.github.io

        And, finally, your possession of this source code implies nothing.

*/

#include "sendMessage_priv.h"

ACTION_RETURN_TYPE UFMN(sendMessage)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);

    pfsm->data.pqueue->queue_count--;

    return THIS(noEvent);
}

ACTION_RETURN_TYPE UFMN(parseMessage)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return THIS(ACK);
}

ACTION_RETURN_TYPE UFMN(checkQueue)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);

    return (pfsm->data.pqueue->queue_count > 0) ? THIS(SEND_MESSAGE) : THIS(noEvent);
}

ACTION_RETURN_TYPE UFMN(noAction)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}

void UFMN(copy_current_message)(pHSM_COMMUNICATOR_DATA pfsm_data)
{
    DBG_PRINTF("%s", __func__);

        psendMessage->data.current_msg = pfsm_data->current_msg;

}

void UFMN(init_data)(pHSM_COMMUNICATOR_DATA pfsm_data)
```

```
{
    DBG_PRINTF("%s", __func__);

        psendMessage->data.pqueue = &pfsm_data->queue;

}
```

Makefile

Download

```
###############################################################
#
#  Makefile for the hsmCommunicator FSMLang example
#

SRC = $(wildcard *actions.c)
FSM_SRC = $(wildcard *.fsm)

FSM_FLAGS=-ts --generate-weak-fns=false

CFLAGS=-DHSM_COMMUNICATOR_DEBUG \
       -DESTABLISH_SESSION_DEBUG \
       -DSEND_MESSAGE_DEBUG \
       -g -ggdb

## FSMLang start

.SUFFIXES: .fsm .html .plantuml

FSM ?= fsm

GENERATED_SRC = $(shell $(FSM) -M $(FSM_FLAGS) $(FSM_SRC))
GENERATED_HDR = $(shell $(FSM) -Mh $(FSM_FLAGS) $(FSM_SRC))

cleanfsm:
        @-rm -f $(GENERATED_SRC) 2> /dev/null
        @-rm -f $(GENERATED_HDR) 2> /dev/null
        @-rm -f *.fsmd           2> /dev/null

%.fsmd: %.fsm
        @set -e; $(FSM) -Md $(FSM_FLAGS) $< > $@

.fsm.o:
        @$(FSM) $(FSM_FLAGS) $< > fsmout
        @$(CC) -c $(CFLAGS) $*.c
        @rm -f $*.c

.fsm.c:
        @$(FSM) $(FSM_FLAGS) $< > fsmout

.fsm.h:
        @$(FSM) $(FSM_FLAGS) $< > fsmout
```

```
ifneq ($(MAKECMDGOALS),clean)
-include $(FSM_SRC:.fsm=.fsmd)
endif

## FSMLang end

OBJS=$(SRC:.c=.o) $(GENERATED_SRC:.c=.o)

all: hsmCommunicator

test.out: hsmCommunicator
        ./$< >$@

hsmCommunicator: $(OBJS) $(FSM) Makefile
        $(CC) -o $@ $(LDFLAGS) $(OBJS)

clean: cleanfsm
        @-rm -f $(OBJS)            2> /dev/null
        @-rm -f hsmCommunicator 2> /dev/null
        @-rm -f test.out           2> /dev/null

$(SRC): hsmCommunicator_priv.h
```

States and Transitions

States can have entry and exit functions. They can inhibit the running of sub-machines. Additionally, a machine can have a transition function which will be called on every state transitions. Finally, state transitions can be specifed by explicit declaration, or they can be done through a function. To explore these topics, we'll continue to build on the hsmCommunicator, starting where we left it at the end of the Event Data discussion.

State Entry and Exit

The following shows how to declare entry or exit (inclusive or) functions for a state.

```
state IN_SESSION
    on entry start_session_timer
    on exit  stop_session_timer
    ;
```

As with some data translation functions, the names here are optional. When not supplied, names will be concocted similar to these:

```
void hsmCommunicator_onEntryTo_IN_SESSION(pHSM_COMMUNICATOR_DATA);
void hsmCommunicator_onExitFrom_IN_SESSION(pHSM_COMMUNICATOR_DATA);
```

As it is, though, we have:

```
void hsmCommunicator_start_session(pHSM_COMMUNICATOR_DATA);
void hsmCommunicator_stop_session(pHSM_COMMUNICATOR_DATA);
```

Before looking at the changes to the main FSM function, we'll add a machine transition function. Here, the thought should be "on transition," because it is a function that will be called on every state transition, rather than being a function called in order to determine what transition should be taken (those are the subject of the next chapter).

To declare a machine transition function, we write:

```
machine hsmCommunicator
on transition track_transitions;
native impl
{
    #define INIT_FSM_DATA {msg_none, { 0 }}
}
```

Looking again in the private header we find:

```
void UFMN(track_transitions)(pHSM_COMMUNICATOR,HSM_COMMUNICATOR_STATE);
```

The function will get the machine's current state from the fsm pointer; it will have the new state passed in as the second argument. As with *all* user functions, the code *must not* alter anything in the machine structure other than data.

The machine transition function feature was created in response to a user request for additional debugging help. Though there may be other valid uses, one should assiduously avoid putting too much logic into the function.

With that, let's look at how this works out in the action array (`-tc`) main FSM function:

```
void hsmCommunicatorFSM(pHSM_COMMUNICATOR pfsm, pHSM_COMMUNICATOR_EVENT event)
{
    HSM_COMMUNICATOR_EVENT_ENUM new_e;
    HSM_COMMUNICATOR_EVENT_ENUM e = event->event;
    HSM_COMMUNICATOR_STATE new_s;

    translateEventData(&pfsm->data, event);

    while (e != THIS(noEvent)) {

#ifdef HSM_COMMUNICATOR_DEBUG
        if (EVENT_IS_NOT_EXCLUDED_FROM_LOG(e))
        {
            DBG_PRINTF("event: %s; state: %s"
                       ,HSM_COMMUNICATOR_EVENT_NAMES[e]
                       ,HSM_COMMUNICATOR_STATE_NAMES[pfsm->state]
                      );
        }
#endif

        /* This is read-only data to facilitate error reporting in action functions */
        pfsm->event = e;

        if (e < hsmCommunicator_noEvent)
        {

            new_e = ((* (*pfsm->actionArray)[e][pfsm->state].action)(pfsm));

            new_s = (*pfsm->actionArray)[e][pfsm->state].transition;


            if (pfsm->state != new_s)
            {
                UFMN(track_transitions)(pfsm,new_s);
                runAppropriateExitFunction(&pfsm->data, pfsm->state);
```

```
                runAppropriateEntryFunction(&pfsm->data, new_s);
                pfsm->state = new_s;

            }

            e = new_e;

        }
        else
        {
            e = findAndRunSubMachine(pfsm, e);
        }

    }

}
```

The new state is captured as a local variable and is checked against the current state; if different, the machine transition function is called, then any Exit and Entry functions. The machine transition function is provided the machine pointer and the new state; the Exit function gets a pointer to the machine's data and the current state; the Entry function gets a pointer to the machine's data and the new state. Once these functions complete, the new state is assigned into the machine structure.

For state and event function array generation (`-ts` and `-te`), this logic is in the state or event handlers, respectively.

State Function

```
static HSM_COMMUNICATOR_EVENT_ENUM UNINITIALIZED_stateFn(pHSM_COMMUNICATOR pfsm,HSM_COMMUNICATOR_EVENT_ENUM e)
{
        HSM_COMMUNICATOR_EVENT_ENUM retVal = THIS(noEvent);
        HSM_COMMUNICATOR_STATE new_s = hsmCommunicator_UNINITIALIZED;

        switch(e)
        {
        case THIS(INIT):
                retVal = UFMN(initialize)(pfsm);
                new_s = hsmCommunicator_IDLE;
                break;
        default:
                DBG_PRINTF("hsmCommunicator_noAction");
                break;
        }


        if (hsmCommunicator_UNINITIALIZED != new_s)
        {
                UFMN(track_transitions)(pfsm, new_s);
                runAppropriateExitFunction(&pfsm->data, hsmCommunicator_UNINITIALIZED);
                runAppropriateEntryFunction(&pfsm->data, new_s);
                pfsm->state = new_s;

        }


        return retVal;
}
```

Event Function

```
static ACTION_RETURN_TYPE hsmCommunicator_handle_INIT(FSM_TYPE_PTR pfsm)
{
        HSM_COMMUNICATOR_STATE s = pfsm->state;
```

```
        ACTION_RETURN_TYPE event = THIS(noEvent);

        switch (pfsm->state)
        {
                case STATE(UNINITIALIZED):
                        event = UFMN(initialize)(pfsm);
                        s = STATE(IDLE);
                        break;
                default:
                        DBG_PRINTF("hsmCommunicator_noAction");
                        break;
        }

        if (s != pfsm->state)
        {
                UFMN(track_transitions)(pfsm,s);
                runAppropriateExitFunction(&pfsm->data, pfsm->state);
                runAppropriateEntryFunction(&pfsm->data, s);
        }

        pfsm->state = s;


        return event;
}
```

Our implementation of these new user functions is vacuous.

```
void hsmCommunicator_store_message(pHSM_COMMUNICATOR_DATA pfsm_data, pHSM_COMMUNICATOR_MESSAGE_RECEIVED_DATA pedata)
{
    DBG_PRINTF("%s", __func__);

    pfsm_data->current_msg = pedata->message;
}

void UFMN(track_transitions)(pHSM_COMMUNICATOR pfsm, HSM_COMMUNICATOR_STATE s)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
}

void hsmCommunicator_start_session_timer(pHSM_COMMUNICATOR_DATA pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
}

void hsmCommunicator_stop_session_timer(pHSM_COMMUNICATOR_DATA pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
}
```

Transition Functions

All examples shown thus far have declared transitions in terms of the state to which the machine will move. It is possible, however to specify a function to be called, which will return the appropriate transition state. This is dangerous, because logic which should be in the main body of the machine design can be hidden away in transition functions. But, there are legitimate uses. For example, consider the following:

```
event e1, e2;
state s1, s2;

action do_nothing[e1, s1];
transition [e2, s1] s2;

do_nothing returns e2, noEvent;
```

In this snippet we see that the machine should remain in state s1 when do_nothing returns noEvent, but should transition to s2 when do_nothing instead returns e2. *do_something* is acting as a predicate determining the end state of the machine, based on some criteria.

The same effect can be accomplished, perhaps with better clarity, with this construction:

```
event e1;
state s1, s2;

transition [e1, s1] where_to_go;

where_to_go  returns s2, noTransition
```

In this case, the criteria for the state change are coded in *where_to_go*.

In both constructions, when e1 occurs in s1, nothing is done and the machine ends up in either s1 or s2.

The following scenario can be used to illustrate this in our HSM Communicator: Consider an “unhappy” path in the session setup wherein the peer rejects the session. The first steps toward handling this scenario are to add another element to our msg_e_t enumeration, and to give the top-level machine a SESSION_REJECTED event, the receipt of which causes the machine to clear the queue and return to the IDLE state (other strategies are imaginable, of course). For this, we need:

hsmCommunicator Top-Level

```
native
{
#ifndef DBG_PRINTF
#include <stdio.h>
#define DBG_PRINTF(...) printf(__VA_ARGS__); printf("\n");
#endif

typedef enum
{
        msg_none
        , msg_step0_response
        , msg_step1_response
        , msg_session_rejected
        , msg_ack
} msg_e_t;

typedef struct _queue_str_
{
        unsigned queue_count;
} queue_str_t;

}

    /** Our peer has rejected our attempt to establish a session. */
```

```
event SESSION_REJECTED;

state IDLE, ESTABLISHING_SESSION;

/** Clear the queue and return to the IDLE state */
action clearQueue[SESSION_REJECTED, ESTABLISHING_SESSION] transition IDLE;
```

What should be done in *establishSession*?

One approach woudld be to give *establishSession* its own REJECTION_RESPONSE event (returned by parseMessage) and have an action function which would notifySessionRejected upon its receipt:

establishSession - with new action

```
/* parent events */
event MESSAGE_RECEIVED, SESSION_REJECTED;

event ESTABLISH_SESSION_REQUEST, STEP0_RESPONSE, STEP1_RESPONSE, REJECTION_RESPONSE;
event parent::MESSAGE_RECEIVED data translator copy_current_message;

state IDLE, AWAITING_RESPONSE;

action notifySessionRejected[REJECTION_RESPONSE, AWAITING_RESPONSE] transition IDLE;

/** Parse the incoming message */
action parseMessage[MESSAGE_RECEIVED, AWAITING_RESPONSE];

parseMessage returns STEP0_RESPONSE, STEP1_RESPONSE, REJECTION_RESPONSE, noEvent;

notifySessionRejected returns parent::SESSION_REJECTED;
```

Or, using a transition function, we could write:

establishSession - with transition function

```
/* parent events */
event MESSAGE_RECEIVED, SESSION_REJECTED;

event ESTABLISH_SESSION_REQUEST, STEP0_RESPONSE, STEP1_RESPONSE;

state IDLE, AWAITING_RESPONSE;

/** Parse the incoming message */
action parseMessage[MESSAGE_RECEIVED, AWAITING_RESPONSE] transition handle_rejection;

parseMessage returns STEP0_RESPONSE, STEP1_RESPONSE, parent::SESSION_REJECTED, noEvent;

handle_rejection returns IDLE, noTransition;
```

In this solution, parseMessage returns the parent SESSION_REJECTED event itself, and handle_rejection decides whether the machine will remain in the AWAITING_RESPONSE state or transition to the IDLE state. In this, we save an action function, but gain a transition function.

But, now that we have the transition function, one further refinment can be made. The action, *notifyParent* is no longer needed; *parseMessage* can return parent::SESSION_ESTABLISHED just as well as it can return parent::SESSION_REJECTED. And, if we're going to do that, then *handle_rejection* is no longer a good name for the transition function, so we'll go with the blander *decide_parse_transition*.

So, we have this:

establishSession - refined

```
/* parent events */
event MESSAGE_RECEIVED, SESSION_ESTABLISHED, SESSION_REJECTED;

machine establishSession
native impl
{
    #define INIT_FSM_DATA {msg_none}
}
{
data
{
    msg_e_t current_msg;
}

event ESTABLISH_SESSION_REQUEST, STEP0_RESPONSE;
event parent::MESSAGE_RECEIVED data translator copy_current_message;

state IDLE, AWAITING_RESPONSE;

/** Start the session establishment process. */
action sendStep0Message[ESTABLISH_SESSION_REQUEST, IDLE]  transition AWAITING_RESPONSE;

/** Continue session establisment */
action sendStep1Message[STEP0_RESPONSE, AWAITING_RESPONSE];

/** Parse the incoming message */
action parseMessage[MESSAGE_RECEIVED, AWAITING_RESPONSE] transition decide_parse_transition;

/* these lines are informational; they affect the html output, but do not affect any C code generated. */
sendStep0Message returns noEvent;

sendStep1Message returns noEvent;

parseMessage returns STEP0_RESPONSE, parent::SESSION_ESTABLISHED, parent::SESSION_REJECTED, noEvent;

decide_parse_transition returns IDLE, noTransition;

}
```

Here is the implementation of the new parseMessage action:

```
ACTION_RETURN_TYPE UFMN(parseMessage)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    ACTION_RETURN_TYPE new_event = THIS(noEvent);

    switch (pfsm->data.current_msg)
    {
    case msg_step0_response:
        new_event = THIS(STEP0_RESPONSE);
        break;
    case msg_step1_response:
        new_event = PARENT(SESSION_ESTABLISHED);
        break;
    case msg_session_rejected:
        new_event = PARENT(SESSION_REJECTED);
        break;
    default:
        break;
```

```
    }

    return new_event;
}
```

The new *decide_parse_transition* looks like this:

```
ESTABLISH_SESSION_STATE UFMN(decide_parse_transition)(pESTABLISH_SESSION pfsm, HSM_COMMUNICATOR_EVENT_ENUM e)
{
    DBG_PRINTF("%s", __func__);

    (void) e;
    ESTABLISH_SESSION_STATE new_state = pfsm->state;

    switch (pfsm->data.current_msg)
    {
    default:
    case msg_step0_response:
        break;
    case msg_step1_response:
    case msg_session_rejected:
        new_state = STATE(IDLE);
        break;
    }

    return new_state;
}
```

This causes the machine to move to the IDLE state when the session is either established or rejected, but remain in the AWAITING_RESPONSE state when the step0 response is received. The key thing to notice here is that this function takes no actions; it merely chooses a path. This is a key to good state machine design using transition functions.

At the top-level, we have merely to add the *clearQueue* action:

```
ACTION_RETURN_TYPE  UFMN(clearQueue)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);

    pfsm->data.queue.queue_count = 0;

    return THIS(noEvent);
}
```

Sources

These pages show the full source of the fsm and generated c and h files.

hsmCommunicator.fsm

Download

```
native
{
#ifndef DBG_PRINTF
#include <stdio.h>
#define DBG_PRINTF(...) printf(__VA_ARGS__); printf("\n");
#endif

typedef enum
```

```
{
        msg_none
        , msg_step0_response
        , msg_step1_response
        , msg_session_rejected
        , msg_ack
} msg_e_t;

typedef struct _queue_str_
{
        unsigned queue_count;
} queue_str_t;

}

/**
<p>This machine manages communications using a "stop and wait" protocol. Only one message is allowed to be outstanding.</p>
<p>Before any message can be exchanged, however, a session must be established with the peer. Establishing a connection
requires several exchanges to authenticate. The session will remain active as long as messages continue to be
exchanged with a minimum frequency.</p>
<p>The user of this machine calls run_hsmCommunicator, passing the SEND_MESSAGE event.  For the first message,
the machine will be IDLE, and thus needs to queue the message, start the establishSession machine, and transition
to the ESTABLISHING_SESSION state.  Requests to send messages received in this state will simply be queued. </p>
<p>While the top level machine is in the ESTABLISHING_SESSION state, the establishSession machine does the establishment work.</p>
<p>When the establishSession machine receives the STEP1_RESPONSE event, it reports to the top level machine that
the session is established by returning the parent's SESSION_ESTABLISHED event.  This will move the top level
machine to its IN_SESSION state and cause it to send the message(s) which are enqueued.</p>
*/
machine hsmCommunicator
on transition track_transitions;
native impl
{
    #define INIT_FSM_DATA {msg_none, { 0 }}
}
{
    data
    {
        msg_e_t current_msg;
        queue_str_t queue;
    }

    /** System initialization */
    event INIT;

    /** This event comes from our client code, asking us to send a message.
    */
    event SEND_MESSAGE;

    /** This event comes from our <i>establishSession</i> submachine, indicating that it has successfully
        completed its work.  We then forward it to our <i>sendMessage</i> submachine to indicate that
        it may now begin to send messages.
    */
    event SESSION_ESTABLISHED;


    /** Our peer has rejected our attempt to establish a session. */
    event SESSION_REJECTED;

    /** This event comes from our external timer, indicating that we've not tickled it in a while, and
        thus should close down our session.
    */
    event SESSION_TIMEOUT;

    /** This event comes from our lower comm layers, indicating that a peer message has arrived.
        While we're in the ESTABLISHING_SESSION state, we forward this event to the <i>establishSession</i>
        submachine; while in the IN_SESSION state, we forward it to the <i>sendMessage</i> submachine.
    */
    event MESSAGE_RECEIVED
          data
          translator store_message
          {

                msg_e_t message;
            }
            ;

      /** The wakeup state. */
```

```
state UNINITIALIZED;

/** The first initialized state.  Also, this is the state to which the machine
    returns when a session times out.
*/
state IDLE;

/** The machine is establishing a session.  The actual work is being done by the <i>establishSession</i>
    submachine.  While in this state, the <i>MESSAGE_RECEIVED</i> event is forwarded to that submachine.
*/
state ESTABLISHING_SESSION;

/** A session has been established, and messages are being exchanged with the peer.  While in this
    state, the <i>MESSAGE_RECEIVED</i> event is forwarded to the <i>sendMessage</i> submachine.
*/
state IN_SESSION
    on entry start_session_timer
    on exit  stop_session_timer
    ;

/**
<p>Establish a connection with the peer.
</p>
<p>Two messages must be exchanged with the peer to successfully establish the session.  The machine needs
only two states, IDLE and AWAITING_RESPONSE since the top level machine tracks whether or not it is in a
session.  The AWAITING_RESPONSE state serves for both required messages, since the receipt of each message produces
a unique event.
</p>
<p>When the STEP1_RESPONSE event is received, the session is considered established.  This machine will then
return the parent's SESSION_ESTABLISHED message and move to its IDLE state.
</p>
*/
machine establishSession
native impl
{
    #define INIT_FSM_DATA {msg_none}
}
{
data
{
    msg_e_t current_msg;
}

event ESTABLISH_SESSION_REQUEST, STEP0_RESPONSE;
event parent::MESSAGE_RECEIVED data translator copy_current_message;

state IDLE, AWAITING_RESPONSE;

/** Start the session establishment process. */
action sendStep0Message[ESTABLISH_SESSION_REQUEST, IDLE]  transition AWAITING_RESPONSE;

/** Continue session establisment */
action sendStep1Message[STEP0_RESPONSE, AWAITING_RESPONSE];


/** Parse the incoming message */
action parseMessage[MESSAGE_RECEIVED, AWAITING_RESPONSE] transition decide_parse_transition;

/* these lines are informational; they affect the html output, but do not affect any C code generated. */
sendStep0Message returns noEvent;

sendStep1Message returns noEvent;

parseMessage returns STEP0_RESPONSE, parent::SESSION_ESTABLISHED, parent::SESSION_REJECTED, noEvent;

decide_parse_transition returns IDLE, noTransition;

}

/**
<p>Send a message to the peer.


</p>
<p>Since the protocol allows only one message to be outsanding, the machine dequeues and transmits a message only
from the IDLE state, transitioning to the AWAITING_ACK state immediately thereafter.
</p>
<p>In the AWAITNG_ACK state, incomming messages are parsed and, when an ACK is found, the machine checks the queue
```

```
    and transitions to the IDLE state.  Checking the queue can return the SEND_MESSAGE event, which will be handled
    from the IDLE state, thus resulting in a transmission and return to the AWAITING_ACK state.
    </p>
    */
    machine sendMessage
    native impl
    {
        #define INIT_FSM_DATA {msg_none, NULL}
    }
    {
    data
    {
        msg_e_t current_msg;
        queue_str_t *pqueue;
    }

    event parent::INIT data translator init_data;

    event parent::MESSAGE_RECEIVED data translator copy_current_message
          , SEND_MESSAGE
          , ACK;

    state UNINITIALIZED, IDLE, AWAITING_ACK;

    transition [INIT, UNINITIALIZED] IDLE;

    /** Dequeue and transmit message to the peer. */
    action      sendMessage[SEND_MESSAGE,IDLE] transition AWAITING_ACK;

    /** Check queue for messages; if found return SEND_MESSAGE; otherwise, return noEvent. */
    action      checkQueue[ACK,AWAITING_ACK]          transition IDLE;

    action      parseMessage[MESSAGE_RECEIVED, AWAITING_ACK];

    /* these lines are informational; they affect the html output, but do not affect any C code generated. */
    sendMessage  returns noEvent;

    checkQueue   returns SEND_MESSAGE, noEvent;

    parseMessage returns ACK, noEvent;

    }

    /* these are actions of the top level machine */

    /** Initialize the machine */
    action initialize[INIT, UNINITIALIZED] transition IDLE;

    /** Start the session establishment process by activating the <i>establishSession</i> machine. */
    action startSessionEstablishment[SEND_MESSAGE, IDLE] transition  ESTABLISHING_SESSION;

    /** Pass the MESSAGE_RECEIVED event along. */
    action passMessageReceived[MESSAGE_RECEIVED, (ESTABLISHING_SESSION, IN_SESSION)];

    /** Clear the queue and return to the IDLE state */
    action clearQueue[SESSION_REJECTED, ESTABLISHING_SESSION] transition IDLE;

    /** Notify the <i>sendMessage</i> machine that the session is established. */
    action completeSessionStart[SESSION_ESTABLISHED, ESTABLISHING_SESSION] transition IN_SESSION;

    /** Extend the session timer, queue the message, and alert the <i>sendMessage</i> machine. */
    action requestMessageTransmission[SEND_MESSAGE, (ESTABLISHING_SESSION, IN_SESSION)];

    transition [SESSION_TIMEOUT, IN_SESSION] IDLE;

    startSessionEstablishment   returns establishSession::ESTABLISH_SESSION_REQUEST;

    completeSessionStart        returns sendMessage::SEND_MESSAGE;

    requestMessageTransmission  returns sendMessage::SEND_MESSAGE;

}
```

hsmCommunicator.c

```
/**
        hsmCommunicator.c

        This file automatically generated by FSMLang
*/

#include "hsmCommunicator_priv.h"
#include <stddef.h>

/* Begin Native Implementation Prolog */


#define INIT_FSM_DATA {msg_none, { 0 }}

/* End Native Implementation Prolog */


#ifndef DBG_PRINTF
#define DBG_PRINTF(...)
#endif

extern HSM_COMMUNICATOR_SUB_FSM_IF hsmCommunicator_establishSession_sub_fsm_if;
extern HSM_COMMUNICATOR_SUB_FSM_IF hsmCommunicator_sendMessage_sub_fsm_if;

static HSM_COMMUNICATOR_EVENT_ENUM UNINITIALIZED_stateFn(pHSM_COMMUNICATOR,HSM_COMMUNICATOR_EVENT_ENUM);
static HSM_COMMUNICATOR_EVENT_ENUM IDLE_stateFn(pHSM_COMMUNICATOR,HSM_COMMUNICATOR_EVENT_ENUM);
static HSM_COMMUNICATOR_EVENT_ENUM ESTABLISHING_SESSION_stateFn(pHSM_COMMUNICATOR,HSM_COMMUNICATOR_EVENT_ENUM);
static HSM_COMMUNICATOR_EVENT_ENUM IN_SESSION_stateFn(pHSM_COMMUNICATOR,HSM_COMMUNICATOR_EVENT_ENUM);
static HSM_COMMUNICATOR_EVENT_ENUM findAndRunSubMachine(pHSM_COMMUNICATOR,HSM_COMMUNICATOR_EVENT_ENUM);

static void translateEventData(pHSM_COMMUNICATOR_DATA,pHSM_COMMUNICATOR_EVENT);

static void runAppropriateEntryFunction(pHSM_COMMUNICATOR_DATA,HSM_COMMUNICATOR_STATE);
static void runAppropriateExitFunction(pHSM_COMMUNICATOR_DATA,HSM_COMMUNICATOR_STATE);
static const HSM_COMMUNICATOR_STATE_FN hsmCommunicator_state_fn_array[hsmCommunicator_numStates] =
{
    UNINITIALIZED_stateFn
    , IDLE_stateFn
    , ESTABLISHING_SESSION_stateFn
    , IN_SESSION_stateFn
};


const pHSM_COMMUNICATOR_SUB_FSM_IF hsmCommunicator_sub_fsm_if_array[THIS(numSubMachines)] =
{
    &hsmCommunicator_establishSession_sub_fsm_if
    , &hsmCommunicator_sendMessage_sub_fsm_if
};

pHSM_COMMUNICATOR_SHARED_EVENT_STR sharing_hsmCommunicator_INIT[] =
{


    &sendMessage_share_hsmCommunicator_INIT_str
    , NULL
};

pHSM_COMMUNICATOR_SHARED_EVENT_STR sharing_hsmCommunicator_MESSAGE_RECEIVED[] =
{
    &establishSession_share_hsmCommunicator_MESSAGE_RECEIVED_str
    , &sendMessage_share_hsmCommunicator_MESSAGE_RECEIVED_str
    , NULL
};

HSM_COMMUNICATOR_EVENT_ENUM hsmCommunicator_pass_shared_event(pHSM_COMMUNICATOR pfsm,pHSM_COMMUNICATOR_SHARED_EVENT_STR sharer_list[])
{
    HSM_COMMUNICATOR_EVENT_ENUM return_event = THIS(noEvent);

    for (pHSM_COMMUNICATOR_SHARED_EVENT_STR *pcurrent_sharer = sharer_list;
            *pcurrent_sharer && return_event == THIS(noEvent);
            pcurrent_sharer++)


    {
        if ((*pcurrent_sharer)->data_translation_fn)
            (*(*pcurrent_sharer)->data_translation_fn)(&pfsm->data);
        return_event = (*(*pcurrent_sharer)->psub_fsm_if->subFSM)((*pcurrent_sharer)->event);
    }
```

```
    return return_event;
}

#ifndef INIT_FSM_DATA
#error INIT_FSM_DATA must be defined
#endif

HSM_COMMUNICATOR hsmCommunicator = {

    INIT_FSM_DATA,
    hsmCommunicator_UNINITIALIZED,
    THIS(INIT),
    &hsmCommunicator_state_fn_array,
    &hsmCommunicator_sub_fsm_if_array,
    hsmCommunicatorFSM
};

pHSM_COMMUNICATOR phsmCommunicator = &hsmCommunicator;

void run_hsmCommunicator(pHSM_COMMUNICATOR_EVENT e)
{
    if (phsmCommunicator)
    {
        phsmCommunicator->fsm(phsmCommunicator,e);
    }
}


#ifndef EVENT_IS_NOT_EXCLUDED_FROM_LOG
#define EVENT_IS_NOT_EXCLUDED_FROM_LOG(e) ((e) == (e))
#endif
void hsmCommunicatorFSM(pHSM_COMMUNICATOR pfsm, pHSM_COMMUNICATOR_EVENT event)
{
    HSM_COMMUNICATOR_EVENT_ENUM e = event->event;

    translateEventData(&pfsm->data, event);

    while (e != hsmCommunicator_noEvent) {

#ifdef HSM_COMMUNICATOR_DEBUG
        if (EVENT_IS_NOT_EXCLUDED_FROM_LOG(e))
        {
            DBG_PRINTF("event: %s; state: %s"
                       ,HSM_COMMUNICATOR_EVENT_NAMES[e]
                       ,HSM_COMMUNICATOR_STATE_NAMES[pfsm->state]
                      );
        }
#endif

        /* This is read-only data to facilitate error reporting in action functions */
        pfsm->event = e;


        if (e < hsmCommunicator_noEvent)
        {
            e = ((* (*pfsm->statesArray)[pfsm->state])(pfsm,e));
        }
        else
        {
            e = findAndRunSubMachine(pfsm, e);
        }

    }
}


static HSM_COMMUNICATOR_EVENT_ENUM findAndRunSubMachine(pHSM_COMMUNICATOR pfsm, HSM_COMMUNICATOR_EVENT_ENUM e)
{
    for (HSM_COMMUNICATOR_SUB_MACHINES machineIterator = THIS(firstSubMachine);
            machineIterator < THIS(numSubMachines);
            machineIterator++


            )
      {
           if (
               ((*pfsm->subMachineArray)[machineIterator]->first_event <= e)
               && ((*pfsm->subMachineArray)[machineIterator]->last_event > e)
           )
```

```
        {
            return ((*(*pfsm->subMachineArray)[machineIterator]->subFSM)(e));
        }
    }

    return THIS(noEvent);

}

static void runAppropriateEntryFunction(pHSM_COMMUNICATOR_DATA pdata,HSM_COMMUNICATOR_STATE s)
{
    switch(s)
    {
    case hsmCommunicator_IN_SESSION:
        hsmCommunicator_start_session_timer(pdata);
        break;
    default:
        break;
    }
}

static void runAppropriateExitFunction(pHSM_COMMUNICATOR_DATA pdata,HSM_COMMUNICATOR_STATE s)
{
    switch(s)
    {
    case hsmCommunicator_IN_SESSION:
        hsmCommunicator_stop_session_timer(pdata);
        break;
    default:
        break;
    }
}

static void translateEventData(pHSM_COMMUNICATOR_DATA pfsm_data, pHSM_COMMUNICATOR_EVENT pevent)
{
    switch(pevent->event)
    {
    case hsmCommunicator_MESSAGE_RECEIVED:
        UFMN(store_message)(pfsm_data, &pevent->event_data.MESSAGE_RECEIVED_data);
        break;
    default:
        break;
    }

}

static HSM_COMMUNICATOR_EVENT_ENUM UNINITIALIZED_stateFn(pHSM_COMMUNICATOR pfsm,HSM_COMMUNICATOR_EVENT_ENUM e)
{
    HSM_COMMUNICATOR_EVENT_ENUM retVal = THIS(noEvent);
    HSM_COMMUNICATOR_STATE new_s = hsmCommunicator_UNINITIALIZED;

    switch(e)
    {


      case THIS(INIT):
          retVal = UFMN(initialize)(pfsm);
          new_s = hsmCommunicator_IDLE;
          break;
      default:
          DBG_PRINTF("hsmCommunicator_noAction");
          break;
      }


      if (hsmCommunicator_UNINITIALIZED != new_s)
      {
          UFMN(track_transitions)(pfsm, new_s);
          runAppropriateExitFunction(&pfsm->data, hsmCommunicator_UNINITIALIZED);
          runAppropriateEntryFunction(&pfsm->data, new_s);
          pfsm->state = new_s;

      }
```

```
    return retVal;
}

static HSM_COMMUNICATOR_EVENT_ENUM IDLE_stateFn(pHSM_COMMUNICATOR pfsm,HSM_COMMUNICATOR_EVENT_ENUM e)
{
    HSM_COMMUNICATOR_EVENT_ENUM retVal = THIS(noEvent);
    HSM_COMMUNICATOR_STATE new_s = hsmCommunicator_IDLE;

    switch(e)
    {
    case THIS(SEND_MESSAGE):
        retVal = UFMN(startSessionEstablishment)(pfsm);
        new_s = hsmCommunicator_ESTABLISHING_SESSION;
        break;
    default:
        DBG_PRINTF("hsmCommunicator_noAction");
        break;
    }


    if (hsmCommunicator_IDLE != new_s)
    {
        UFMN(track_transitions)(pfsm, new_s);
        runAppropriateExitFunction(&pfsm->data, hsmCommunicator_IDLE);
        runAppropriateEntryFunction(&pfsm->data, new_s);
        pfsm->state = new_s;

    }


    return retVal;
}

static HSM_COMMUNICATOR_EVENT_ENUM ESTABLISHING_SESSION_stateFn(pHSM_COMMUNICATOR pfsm,HSM_COMMUNICATOR_EVENT_ENUM e)
{
    HSM_COMMUNICATOR_EVENT_ENUM retVal = THIS(noEvent);
    HSM_COMMUNICATOR_STATE new_s = hsmCommunicator_ESTABLISHING_SESSION;

    switch(e)
    {
    case THIS(MESSAGE_RECEIVED):
        retVal = UFMN(passMessageReceived)(pfsm);
        break;
    case THIS(SESSION_REJECTED):
        retVal = UFMN(clearQueue)(pfsm);
        new_s = hsmCommunicator_IDLE;
        break;
    case THIS(SESSION_ESTABLISHED):
        retVal = UFMN(completeSessionStart)(pfsm);
        new_s = hsmCommunicator_IN_SESSION;
        break;


    case THIS(SEND_MESSAGE):
        retVal = UFMN(requestMessageTransmission)(pfsm);
        break;
    default:
        DBG_PRINTF("hsmCommunicator_noAction");
        break;
    }


    if (hsmCommunicator_ESTABLISHING_SESSION != new_s)
    {
        UFMN(track_transitions)(pfsm, new_s);
        runAppropriateExitFunction(&pfsm->data, hsmCommunicator_ESTABLISHING_SESSION);
        runAppropriateEntryFunction(&pfsm->data, new_s);
        pfsm->state = new_s;

    }



    return retVal;
}

static HSM_COMMUNICATOR_EVENT_ENUM IN_SESSION_stateFn(pHSM_COMMUNICATOR pfsm,HSM_COMMUNICATOR_EVENT_ENUM e)
{
```

```
    HSM_COMMUNICATOR_EVENT_ENUM retVal = THIS(noEvent);
    HSM_COMMUNICATOR_STATE new_s = hsmCommunicator_IN_SESSION;

    switch(e)
    {
    case THIS(MESSAGE_RECEIVED):
        retVal = UFMN(passMessageReceived)(pfsm);
        break;
    case THIS(SEND_MESSAGE):
        retVal = UFMN(requestMessageTransmission)(pfsm);
        break;
    case THIS(SESSION_TIMEOUT):
        DBG_PRINTF("hsmCommunicator_noAction");
        new_s = hsmCommunicator_IDLE;
        break;
    default:
        DBG_PRINTF("hsmCommunicator_noAction");
        break;
    }


    if (hsmCommunicator_IN_SESSION != new_s)
    {
        UFMN(track_transitions)(pfsm, new_s);
        runAppropriateExitFunction(&pfsm->data, hsmCommunicator_IN_SESSION);
        runAppropriateEntryFunction(&pfsm->data, new_s);
        pfsm->state = new_s;

    }


    return retVal;
}


HSM_COMMUNICATOR_EVENT_ENUM UFMN(initialize)(pHSM_COMMUNICATOR pfsm)
{
    DBG_PRINTF("%s", __func__);
    return hsmCommunicator_pass_shared_event(pfsm, sharing_hsmCommunicator_INIT);
}

HSM_COMMUNICATOR_EVENT_ENUM UFMN(passMessageReceived)(pHSM_COMMUNICATOR pfsm)
{
    DBG_PRINTF("%s", __func__);
    return hsmCommunicator_pass_shared_event(pfsm, sharing_hsmCommunicator_MESSAGE_RECEIVED);
}


#ifdef HSM_COMMUNICATOR_DEBUG
char *HSM_COMMUNICATOR_EVENT_NAMES[] = {
    "hsmCommunicator_INIT"
    ,"hsmCommunicator_SEND_MESSAGE"
    ,"hsmCommunicator_SESSION_ESTABLISHED"
    ,"hsmCommunicator_SESSION_REJECTED"
    ,"hsmCommunicator_SESSION_TIMEOUT"
    ,"hsmCommunicator_MESSAGE_RECEIVED"
    , "hsmCommunicator_noEvent"
    , "hsmCommunicator_numEvents"
    , "hsmCommunicator_establishSession_ESTABLISH_SESSION_REQUEST"
    , "hsmCommunicator_establishSession_STEP0_RESPONSE"
    , "hsmCommunicator_establishSession_MESSAGE_RECEIVED"
    , "hsmCommunicator_establishSession_noEvent"
    , "hsmCommunicator_sendMessage_INIT"
    , "hsmCommunicator_sendMessage_MESSAGE_RECEIVED"
    , "hsmCommunicator_sendMessage_SEND_MESSAGE"
    , "hsmCommunicator_sendMessage_ACK"
    , "hsmCommunicator_sendMessage_noEvent"
};

char *HSM_COMMUNICATOR_STATE_NAMES[] = {
    "hsmCommunicator_UNINITIALIZED"
    ,"hsmCommunicator_IDLE"
    ,"hsmCommunicator_ESTABLISHING_SESSION"
    ,"hsmCommunicator_IN_SESSION"
};

#endif
```

hsmCommunicator.h

```
/**
        hsmCommunicator.h

        This file automatically generated by FSMLang
*/

#ifndef _HSMCOMMUNICATOR_H_
#define _HSMCOMMUNICATOR_H_

#include "hsmCommunicator_events.h"
#ifndef HSM_COMMUNICATOR_NATIVE_PROLOG
#define HSM_COMMUNICATOR_NATIVE_PROLOG

#ifndef DBG_PRINTF
#include <stdio.h>
#define DBG_PRINTF(...) printf(__VA_ARGS__); printf("\n");
#endif

typedef enum
{
    msg_none
    , msg_step0_response
    , msg_step1_response
    , msg_session_rejected
    , msg_ack
} msg_e_t;

typedef struct _queue_str_
{
    unsigned queue_count;
} queue_str_t;


#endif
#define FSM_VERSION "1.45.1"

#ifndef NO_CONVENIENCE_MACROS
#undef UFMN
#define UFMN(A) hsmCommunicator_##A
#undef THIS
#define THIS(A) hsmCommunicator_##A
#endif
#undef STATE
#define STATE(A) hsmCommunicator_##A
#undef HSM_COMMUNICATOR
#define HSM_COMMUNICATOR(A) hsmCommunicator_##A
#undef ESTABLISH_SESSION
#define ESTABLISH_SESSION(A) hsmCommunicator_establishSession_##A
#undef SEND_MESSAGE
#define SEND_MESSAGE(A) hsmCommunicator_sendMessage_##A
```

```
#undef ACTION_RETURN_TYPE
#define ACTION_RETURN_TYPE HSM_COMMUNICATOR_EVENT_ENUM
typedef struct _hsmCommunicator_MESSAGE_RECEIVED_data_ {
    msg_e_t message;
} HSM_COMMUNICATOR_MESSAGE_RECEIVED_DATA, *pHSM_COMMUNICATOR_MESSAGE_RECEIVED_DATA;

typedef union {
    HSM_COMMUNICATOR_MESSAGE_RECEIVED_DATA MESSAGE_RECEIVED_data;
} HSM_COMMUNICATOR_EVENT_DATA, *pHSM_COMMUNICATOR_EVENT_DATA;

typedef struct {
    HSM_COMMUNICATOR_EVENT_ENUM event;
    HSM_COMMUNICATOR_EVENT_DATA event_data;
} HSM_COMMUNICATOR_EVENT, *pHSM_COMMUNICATOR_EVENT;


void run_hsmCommunicator(pHSM_COMMUNICATOR_EVENT);

typedef struct _hsmCommunicator_struct_ *pHSM_COMMUNICATOR;
extern pHSM_COMMUNICATOR phsmCommunicator;


#endif
```

sendMessage_priv.h

```
/**
        sendMessage_priv.h

        This file automatically generated by FSMLang
*/

#ifndef _SENDMESSAGE_PRIV_H_
#define _SENDMESSAGE_PRIV_H_

#include "hsmCommunicator_submach.h"
#include "hsmCommunicator.h"

#ifdef SEND_MESSAGE_DEBUG
#include <stdio.h>
#include <stdlib.h>
#endif

/*
        sub-machine events are included in the top-level machine event enumeration.
        These macros set the appropriate names for events from THIS machine
        and those from the PARENT machine.

        They may be turned off as needed.
*/
#ifndef NO_CONVENIENCE_MACROS
#undef UFMN
#define UFMN(A) hsmCommunicator_sendMessage_##A
#undef THIS
#define THIS(A) hsmCommunicator_sendMessage_##A
#undef PARENT
#define PARENT(A) hsmCommunicator_##A
#endif
#undef STATE
#define STATE(A) sendMessage_##A

#ifdef HSM_COMMUNICATOR_SEND_MESSAGE_DEBUG
extern char *SEND_MESSAGE_EVENT_NAMES[];
extern char *SEND_MESSAGE_STATE_NAMES[];
#endif

typedef enum {
    sendMessage_UNINITIALIZED
    , sendMessage_IDLE
    , sendMessage_AWAITING_ACK
    , sendMessage_numStates
}  SEND_MESSAGE_STATE;

typedef struct _sendMessage_data_struct_ SEND_MESSAGE_DATA, *pSEND_MESSAGE_DATA;
typedef struct _sendMessage_struct_ SEND_MESSAGE, *pSEND_MESSAGE;
#undef FSM_TYPE_PTR
#define FSM_TYPE_PTR pSEND_MESSAGE
```

```
extern SEND_MESSAGE sendMessage;

extern pSEND_MESSAGE psendMessage;

typedef HSM_COMMUNICATOR_EVENT_ENUM (*SEND_MESSAGE_ACTION_FN)(FSM_TYPE_PTR);

typedef HSM_COMMUNICATOR_EVENT_ENUM (*SEND_MESSAGE_FSM)(FSM_TYPE_PTR,HSM_COMMUNICATOR_EVENT_ENUM);

struct _sendMessage_data_struct_ {
    msg_e_t current_msg;
    queue_str_t * pqueue;
};

typedef ACTION_RETURN_TYPE (*SEND_MESSAGE_STATE_FN)(pSEND_MESSAGE,HSM_COMMUNICATOR_EVENT_ENUM);

static const SEND_MESSAGE_STATE_FN sendMessage_state_fn_array[sendMessage_numStates];

struct _sendMessage_struct_ {
    SEND_MESSAGE_DATA             data;
    SEND_MESSAGE_STATE            state;
    HSM_COMMUNICATOR_EVENT_ENUM   event;
    SEND_MESSAGE_STATE_FN   const (*statesArray)[sendMessage_numStates];
    SEND_MESSAGE_FSM              fsm;
};


ACTION_RETURN_TYPE hsmCommunicator_sendMessage_sendMessage(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_sendMessage_checkQueue(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_sendMessage_parseMessage(FSM_TYPE_PTR);

void hsmCommunicator_sendMessage_init_data(pHSM_COMMUNICATOR_DATA);
void hsmCommunicator_sendMessage_copy_current_message(pHSM_COMMUNICATOR_DATA);

#endif
```

hsmCommunicator_priv.h

```
/**
        hsmCommunicator_priv.h

        This file automatically generated by FSMLang
*/

#ifndef _HSMCOMMUNICATOR_PRIV_H_
#define _HSMCOMMUNICATOR_PRIV_H_

#include "hsmCommunicator.h"
#ifndef HSM_COMMUNICATOR_NATIVE_PROLOG
#define HSM_COMMUNICATOR_NATIVE_PROLOG

#ifndef DBG_PRINTF
#include <stdio.h>
#define DBG_PRINTF(...) printf(__VA_ARGS__); printf("\n");
#endif

typedef enum
{
    msg_none
    , msg_step0_response
    , msg_step1_response
    , msg_session_rejected
    , msg_ack
} msg_e_t;

typedef struct _queue_str_
{
    unsigned queue_count;
} queue_str_t;


#endif

#ifdef HSM_COMMUNICATOR_DEBUG
#include <stdio.h>
#include <stdlib.h>
#endif


#ifdef HSM_COMMUNICATOR_DEBUG
extern char *HSM_COMMUNICATOR_EVENT_NAMES[];
extern char *HSM_COMMUNICATOR_STATE_NAMES[];
#endif

typedef enum {
    hsmCommunicator_UNINITIALIZED
    , hsmCommunicator_IDLE
    , hsmCommunicator_ESTABLISHING_SESSION
    , hsmCommunicator_IN_SESSION
```

```
    , hsmCommunicator_numStates
}  HSM_COMMUNICATOR_STATE;

typedef struct _hsmCommunicator_struct_ HSM_COMMUNICATOR;
#undef FSM_TYPE_PTR
#define FSM_TYPE_PTR pHSM_COMMUNICATOR
extern HSM_COMMUNICATOR hsmCommunicator;

typedef HSM_COMMUNICATOR_EVENT_ENUM (*HSM_COMMUNICATOR_ACTION_FN)(FSM_TYPE_PTR);

typedef void (*HSM_COMMUNICATOR_FSM)(FSM_TYPE_PTR,pHSM_COMMUNICATOR_EVENT);

void hsmCommunicatorFSM(FSM_TYPE_PTR,pHSM_COMMUNICATOR_EVENT);

#include "hsmCommunicator_submach.h"
typedef ACTION_RETURN_TYPE (*HSM_COMMUNICATOR_STATE_FN)(pHSM_COMMUNICATOR,HSM_COMMUNICATOR_EVENT_ENUM);

static const HSM_COMMUNICATOR_STATE_FN hsmCommunicator_state_fn_array[hsmCommunicator_numStates];

struct _hsmCommunicator_struct_ {
    HSM_COMMUNICATOR_DATA               data;
    HSM_COMMUNICATOR_STATE              state;
    HSM_COMMUNICATOR_EVENT_ENUM         event;
    HSM_COMMUNICATOR_STATE_FN     const (*statesArray)[hsmCommunicator_numStates];
    pHSM_COMMUNICATOR_SUB_FSM_IF   const (*subMachineArray)[hsmCommunicator_numSubMachines];
    HSM_COMMUNICATOR_FSM                fsm;
};


ACTION_RETURN_TYPE hsmCommunicator_initialize(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_startSessionEstablishment(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_passMessageReceived(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_clearQueue(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_completeSessionStart(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_requestMessageTransmission(FSM_TYPE_PTR);

void UFMN(track_transitions)(pHSM_COMMUNICATOR,HSM_COMMUNICATOR_STATE);


void hsmCommunicator_start_session_timer(pHSM_COMMUNICATOR_DATA);
void hsmCommunicator_stop_session_timer(pHSM_COMMUNICATOR_DATA);

void hsmCommunicator_store_message(pHSM_COMMUNICATOR_DATA,pHSM_COMMUNICATOR_MESSAGE_RECEIVED_DATA);


#endif
```

hsmCommunicator_submach.h

```
/**
        hsmCommunicator_submach.h

        This file automatically generated by FSMLang
*/

#ifndef _HSMCOMMUNICATOR_SUBMACH_H_
#define _HSMCOMMUNICATOR_SUBMACH_H_

#include "hsmCommunicator.h"
#ifndef HSM_COMMUNICATOR_NATIVE_PROLOG
#define HSM_COMMUNICATOR_NATIVE_PROLOG

#ifndef DBG_PRINTF
#include <stdio.h>
#define DBG_PRINTF(...) printf(__VA_ARGS__); printf("\n");
#endif

typedef enum
{
    msg_none
    , msg_step0_response
    , msg_step1_response
    , msg_session_rejected
    , msg_ack
} msg_e_t;

typedef struct _queue_str_
{
    unsigned queue_count;
} queue_str_t;


#endif
typedef struct _hsmCommunicator_data_struct_ HSM_COMMUNICATOR_DATA, *pHSM_COMMUNICATOR_DATA;
/* Sub Machine Declarations */

typedef enum {
    establishSession_e
    , hsmCommunicator_firstSubMachine = establishSession_e
    ,  sendMessage_e
    , hsmCommunicator_numSubMachines
} HSM_COMMUNICATOR_SUB_MACHINES;

typedef HSM_COMMUNICATOR_EVENT_ENUM (*HSM_COMMUNICATOR_SUB_MACHINE_FN)(HSM_COMMUNICATOR_EVENT_ENUM);
typedef struct _hsmCommunicator_sub_fsm_if_ HSM_COMMUNICATOR_SUB_FSM_IF, *pHSM_COMMUNICATOR_SUB_FSM_IF;
struct _hsmCommunicator_sub_fsm_if_
{
    HSM_COMMUNICATOR_EVENT_ENUM      first_event;
    HSM_COMMUNICATOR_EVENT_ENUM      last_event;
    HSM_COMMUNICATOR_SUB_MACHINE_FN  subFSM;


};

typedef void (*HSM_COMMUNICATOR_DATA_TRANSLATION_FN)(pHSM_COMMUNICATOR_DATA);
/* Some sub-machines share parent events. */
typedef struct _hsmCommunicator_shared_event_str_ HSM_COMMUNICATOR_SHARED_EVENT_STR, *pHSM_COMMUNICATOR_SHARED_EVENT_STR;
struct _hsmCommunicator_shared_event_str_
{
    HSM_COMMUNICATOR_EVENT_ENUM           event;
    HSM_COMMUNICATOR_DATA_TRANSLATION_FN  data_translation_fn;
    pHSM_COMMUNICATOR_SUB_FSM_IF          psub_fsm_if;
};
extern HSM_COMMUNICATOR_EVENT_ENUM hsmCommunicator_pass_shared_event(pHSM_COMMUNICATOR,pHSM_COMMUNICATOR_SHARED_EVENT_STR[]);

extern HSM_COMMUNICATOR_SHARED_EVENT_STR sendMessage_share_hsmCommunicator_INIT_str;
extern pHSM_COMMUNICATOR_SHARED_EVENT_STR sharing_hsmCommunicator_INIT[];


extern HSM_COMMUNICATOR_SHARED_EVENT_STR establishSession_share_hsmCommunicator_MESSAGE_RECEIVED_str;
extern HSM_COMMUNICATOR_SHARED_EVENT_STR sendMessage_share_hsmCommunicator_MESSAGE_RECEIVED_str;
extern pHSM_COMMUNICATOR_SHARED_EVENT_STR sharing_hsmCommunicator_MESSAGE_RECEIVED[];
```

```
struct _hsmCommunicator_data_struct_ {
    msg_e_t current_msg;
    queue_str_t queue;
};


#endif
```

hsmc-actions.c

Download

```
/*
  hsmc-actions.c

        Action functions for the hsmCommunicator top-level FSM.

    FSMLang (fsm) - A Finite State Machine description language.
    Copyright (C) 2024  Steven Stanton

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

    Steven Stanton
    sstanton@pesticidesoftware.com

    For the latest on FSMLang: https://fsmlang.github.io/

    And, finally, your possession of this source code implies nothing.

*/

#include "hsmCommunicator_priv.h"

ACTION_RETURN_TYPE  UFMN(startSessionEstablishment)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);

    pfsm->data.queue.queue_count++;

    return ESTABLISH_SESSION(ESTABLISH_SESSION_REQUEST);
```

```
}

ACTION_RETURN_TYPE  UFMN(completeSessionStart)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    return SEND_MESSAGE(SEND_MESSAGE);
}

ACTION_RETURN_TYPE  UFMN(queueMessage)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);

    pfsm->data.queue.queue_count++;

    return SEND_MESSAGE(SEND_MESSAGE);
}

ACTION_RETURN_TYPE  UFMN(clearQueue)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);

    pfsm->data.queue.queue_count = 0;

    return THIS(noEvent);
}

ACTION_RETURN_TYPE  UFMN(requestMessageTransmission)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);

    pfsm->data.queue.queue_count++;

    return SEND_MESSAGE(SEND_MESSAGE);
}

ACTION_RETURN_TYPE  UFMN(noAction)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}

void hsmCommunicator_store_message(pHSM_COMMUNICATOR_DATA pfsm_data, pHSM_COMMUNICATOR_MESSAGE_RECEIVED_DATA pedata)
{
    DBG_PRINTF("%s", __func__);

    pfsm_data->current_msg = pedata->message;
}

void UFMN(track_transitions)(pHSM_COMMUNICATOR pfsm, HSM_COMMUNICATOR_STATE s)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;


}

void hsmCommunicator_start_session_timer(pHSM_COMMUNICATOR_DATA pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
}

void hsmCommunicator_stop_session_timer(pHSM_COMMUNICATOR_DATA pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
}
```

```c
int main(void)
{
    HSM_COMMUNICATOR_EVENT event;

    event.event = THIS(INIT);
    run_hsmCommunicator(&event);

    event.event = THIS(SEND_MESSAGE);
    run_hsmCommunicator(&event);

    event.event = THIS(MESSAGE_RECEIVED);
    event.event_data.MESSAGE_RECEIVED_data.message = msg_step0_response;
    run_hsmCommunicator(&event);

    event.event = THIS(SEND_MESSAGE);
    run_hsmCommunicator(&event);

    event.event = THIS(MESSAGE_RECEIVED);
    event.event_data.MESSAGE_RECEIVED_data.message = msg_step1_response;
    run_hsmCommunicator(&event);

    event.event = THIS(MESSAGE_RECEIVED);
    event.event_data.MESSAGE_RECEIVED_data.message = msg_ack;
    run_hsmCommunicator(&event);

    event.event = THIS(MESSAGE_RECEIVED);
    event.event_data.MESSAGE_RECEIVED_data.message = msg_ack;
    run_hsmCommunicator(&event);

    return 0;
}
```

establishSession.c

```
/**
        establishSession.c

        This file automatically generated by FSMLang
*/

#include "establishSession_priv.h"
#include <stddef.h>

/* Begin Native Implementation Prolog */


#define INIT_FSM_DATA {msg_none}

/* End Native Implementation Prolog */


#ifndef DBG_PRINTF
#define DBG_PRINTF(...)
#endif

static HSM_COMMUNICATOR_EVENT_ENUM establishSessionFSM(FSM_TYPE_PTR,HSM_COMMUNICATOR_EVENT_ENUM);

static HSM_COMMUNICATOR_EVENT_ENUM IDLE_stateFn(pESTABLISH_SESSION,HSM_COMMUNICATOR_EVENT_ENUM);
static HSM_COMMUNICATOR_EVENT_ENUM AWAITING_RESPONSE_stateFn(pESTABLISH_SESSION,HSM_COMMUNICATOR_EVENT_ENUM);

static const ESTABLISH_SESSION_STATE_FN establishSession_state_fn_array[establishSession_numStates] =
{
    IDLE_stateFn
    , AWAITING_RESPONSE_stateFn
};

HSM_COMMUNICATOR_EVENT_ENUM THIS(sub_machine_fn)(HSM_COMMUNICATOR_EVENT_ENUM e)
{
    return establishSessionFSM(pestablishSession,e);
}

HSM_COMMUNICATOR_SUB_FSM_IF hsmCommunicator_establishSession_sub_fsm_if =
{
    .subFSM = THIS(sub_machine_fn)
    , .first_event = THIS(firstEvent)
    , .last_event = THIS(noEvent)
};

HSM_COMMUNICATOR_SHARED_EVENT_STR establishSession_share_hsmCommunicator_MESSAGE_RECEIVED_str = {
    .event               = THIS(MESSAGE_RECEIVED)
    , .data_translation_fn = UFMN(copy_current_message)
    , .psub_fsm_if         = &hsmCommunicator_establishSession_sub_fsm_if
};

#ifndef INIT_FSM_DATA


#error INIT_FSM_DATA must be defined
#endif

ESTABLISH_SESSION establishSession = {

    INIT_FSM_DATA,
    establishSession_IDLE,
    THIS(ESTABLISH_SESSION_REQUEST),
    &establishSession_state_fn_array,
    establishSessionFSM
};
```

```
pESTABLISH_SESSION pestablishSession = &establishSession;


#ifndef EVENT_IS_NOT_EXCLUDED_FROM_LOG
#define EVENT_IS_NOT_EXCLUDED_FROM_LOG(e) (e == e)
#endif
HSM_COMMUNICATOR_EVENT_ENUM establishSessionFSM(pESTABLISH_SESSION pfsm, HSM_COMMUNICATOR_EVENT_ENUM event)
{
    HSM_COMMUNICATOR_EVENT_ENUM e = event;

    while ((e != THIS(noEvent))
            && (e >= THIS(firstEvent))
          )
    {

#ifdef HSM_COMMUNICATOR_ESTABLISH_SESSION_DEBUG
        if ((EVENT_IS_NOT_EXCLUDED_FROM_LOG(e))
                && (e >= THIS(firstEvent))
                && (e < THIS(noEvent))
           )
        {
            DBG_PRINTF("event: %s; state: %s"
                       ,ESTABLISH_SESSION_EVENT_NAMES[e - THIS(firstEvent)]
                       ,ESTABLISH_SESSION_STATE_NAMES[pfsm->state]
                      );
        }
#endif

        /* This is read-only data to facilitate error reporting in action functions */
        pfsm->event = e;

        if ((e >= THIS(firstEvent))
                && (e < THIS(noEvent))
           )
        {
            e = ((* (*pfsm->statesArray)[pfsm->state])(pfsm,e));
        }

    }

    return e == THIS(noEvent) ? PARENT(noEvent) : e;
}

static HSM_COMMUNICATOR_EVENT_ENUM IDLE_stateFn(pESTABLISH_SESSION pfsm,HSM_COMMUNICATOR_EVENT_ENUM e)
{
    HSM_COMMUNICATOR_EVENT_ENUM retVal = THIS(noEvent);

    switch(e)
    {
    case THIS(ESTABLISH_SESSION_REQUEST):
        retVal = UFMN(sendStep0Message)(pfsm);
        pfsm->state = establishSession_AWAITING_RESPONSE;
        break;


    default:
        DBG_PRINTF("hsmCommunicator_establishSession_noAction");
        break;
    }

    return retVal;
}

static HSM_COMMUNICATOR_EVENT_ENUM AWAITING_RESPONSE_stateFn(pESTABLISH_SESSION pfsm,HSM_COMMUNICATOR_EVENT_ENUM e)
{
    HSM_COMMUNICATOR_EVENT_ENUM retVal = THIS(noEvent);

    switch(e)

     {
     case THIS(STEP0_RESPONSE):
```

```
            retVal = UFMN(sendStep1Message)(pfsm);
            break;
        case THIS(MESSAGE_RECEIVED):
            retVal = UFMN(parseMessage)(pfsm);
            pfsm->state = UFMN(decide_parse_transition)(pfsm,e);
            break;
        default:
            DBG_PRINTF("hsmCommunicator_establishSession_noAction");
            break;
        }

        return retVal;
    }



    #ifdef HSM_COMMUNICATOR_ESTABLISH_SESSION_DEBUG
    char *ESTABLISH_SESSION_EVENT_NAMES[] = {
        "hsmCommunicator_establishSession_ESTABLISH_SESSION_REQUEST"
        ,"hsmCommunicator_establishSession_STEP0_RESPONSE"
        ,"hsmCommunicator_establishSession_MESSAGE_RECEIVED"
        , "establishSession_noEvent"
        , "establishSession_numEvents"
    };

    char *ESTABLISH_SESSION_STATE_NAMES[] = {
        "hsmCommunicator_establishSession_IDLE"
        ,"hsmCommunicator_establishSession_AWAITING_RESPONSE"
    };

    #endif
```

establishSession_priv.h

```
/**
        establishSession_priv.h

        This file automatically generated by FSMLang
*/

#ifndef _ESTABLISHSESSION_PRIV_H_
#define _ESTABLISHSESSION_PRIV_H_

#include "hsmCommunicator_submach.h"
#include "hsmCommunicator.h"

#ifdef ESTABLISH_SESSION_DEBUG
#include <stdio.h>
#include <stdlib.h>
#endif

/*
        sub-machine events are included in the top-level machine event enumeration.
        These macros set the appropriate names for events from THIS machine
        and those from the PARENT machine.

        They may be turned off as needed.
*/
#ifndef NO_CONVENIENCE_MACROS
#undef UFMN
#define UFMN(A) hsmCommunicator_establishSession_##A
#undef THIS
#define THIS(A) hsmCommunicator_establishSession_##A
#undef PARENT
#define PARENT(A) hsmCommunicator_##A
#endif
#undef STATE
#define STATE(A) establishSession_##A

#ifdef HSM_COMMUNICATOR_ESTABLISH_SESSION_DEBUG
extern char *ESTABLISH_SESSION_EVENT_NAMES[];
extern char *ESTABLISH_SESSION_STATE_NAMES[];
#endif

typedef enum {
    establishSession_IDLE
    , establishSession_AWAITING_RESPONSE
    , establishSession_numStates
}  ESTABLISH_SESSION_STATE;

typedef struct _establishSession_data_struct_ ESTABLISH_SESSION_DATA, *pESTABLISH_SESSION_DATA;
typedef struct _establishSession_struct_ ESTABLISH_SESSION, *pESTABLISH_SESSION;
#undef FSM_TYPE_PTR
#define FSM_TYPE_PTR pESTABLISH_SESSION
extern ESTABLISH_SESSION establishSession;


extern pESTABLISH_SESSION pestablishSession;

typedef HSM_COMMUNICATOR_EVENT_ENUM (*ESTABLISH_SESSION_ACTION_FN)(FSM_TYPE_PTR);

typedef ESTABLISH_SESSION_STATE (*ESTABLISH_SESSION_TRANSITION_FN)(FSM_TYPE_PTR,HSM_COMMUNICATOR_EVENT_ENUM);

typedef HSM_COMMUNICATOR_EVENT_ENUM (*ESTABLISH_SESSION_FSM)(FSM_TYPE_PTR,HSM_COMMUNICATOR_EVENT_ENUM);

struct _establishSession_data_struct_ {
```

```
    msg_e_t current_msg;
};

typedef ACTION_RETURN_TYPE (*ESTABLISH_SESSION_STATE_FN)(pESTABLISH_SESSION,HSM_COMMUNICATOR_EVENT_ENUM);

static const ESTABLISH_SESSION_STATE_FN establishSession_state_fn_array[establishSession_numStates];

struct _establishSession_struct_ {
    ESTABLISH_SESSION_DATA              data;
    ESTABLISH_SESSION_STATE             state;
    HSM_COMMUNICATOR_EVENT_ENUM         event;
    ESTABLISH_SESSION_STATE_FN    const (*statesArray)[establishSession_numStates];
    ESTABLISH_SESSION_FSM               fsm;
};


ACTION_RETURN_TYPE hsmCommunicator_establishSession_sendStep0Message(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_establishSession_sendStep1Message(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_establishSession_parseMessage(FSM_TYPE_PTR);

ESTABLISH_SESSION_STATE UFMN(decide_parse_transition)(pESTABLISH_SESSION,HSM_COMMUNICATOR_EVENT_ENUM);
void hsmCommunicator_establishSession_copy_current_message(pHSM_COMMUNICATOR_DATA);

#endif
```

session-actions.c

Download

```
/*
  session_actions.c

        Action functions for the establishSession FSM.

        FSMLang (fsm) - A Finite State Machine description language.
        Copyright (C) 4/20/2025  Steven Stanton

        This program is free software; you can redistribute it and/or modify
        it under the terms of the GNU General Public License as published by
        the Free Software Foundation; either version 2 of the License, or
        (at your option) any later version.

        This program is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
        GNU General Public License for more details.

        You should have received a copy of the GNU General Public License
        along with this program; if not, write to the Free Software
        Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

        Steven Stanton
        sstanton@pesticidesoftware.com

        For the latest on FSMLang: http://fsmlang.github.io

        And, finally, your possession of this source code implies nothing.

*/
```

```
#include <stdbool.h>
#include "establishSession_priv.h"

ACTION_RETURN_TYPE UFMN(sendStep0Message)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}

ACTION_RETURN_TYPE UFMN(sendStep1Message)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}

ACTION_RETURN_TYPE UFMN(notifyParent)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return PARENT(SESSION_ESTABLISHED);
}

ACTION_RETURN_TYPE UFMN(parseMessage)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    ACTION_RETURN_TYPE new_event = THIS(noEvent);

    switch (pfsm->data.current_msg)
    {
    case msg_step0_response:
        new_event = THIS(STEP0_RESPONSE);
        break;
    case msg_step1_response:
        new_event = PARENT(SESSION_ESTABLISHED);
        break;
    case msg_session_rejected:
        new_event = PARENT(SESSION_REJECTED);
        break;
    default:
        break;
    }

    return new_event;
}

ACTION_RETURN_TYPE UFMN(noAction)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return THIS(noEvent);
```

```
}

void UFMN(copy_current_message)(pHSM_COMMUNICATOR_DATA pfsm_data)
{
    DBG_PRINTF("%S", __func__);

    pestablishSession->data.current_msg = pfsm_data->current_msg;
}

ESTABLISH_SESSION_STATE UFMN(decide_parse_transition)(pESTABLISH_SESSION pfsm, HSM_COMMUNICATOR_EVENT_ENUM e)
{
    DBG_PRINTF("%S", __func__);

    (void) e;
    ESTABLISH_SESSION_STATE new_state = pfsm->state;

    switch (pfsm->data.current_msg)
    {
    default:
    case msg_step0_response:
        break;
    case msg_step1_response:
    case msg_session_rejected:
        new_state = STATE(IDLE);
        break;
    }

    return new_state;
}
```

sendMessage.c

```
/**
        sendMessage.c

        This file automatically generated by FSMLang
*/

#include "sendMessage_priv.h"
#include <stddef.h>

/* Begin Native Implementation Prolog */


#define INIT_FSM_DATA {msg_none, NULL}

/* End Native Implementation Prolog */


#ifndef DBG_PRINTF
#define DBG_PRINTF(...)
#endif

static HSM_COMMUNICATOR_EVENT_ENUM sendMessageFSM(FSM_TYPE_PTR,HSM_COMMUNICATOR_EVENT_ENUM);

static HSM_COMMUNICATOR_EVENT_ENUM UNINITIALIZED_stateFn(pSEND_MESSAGE,HSM_COMMUNICATOR_EVENT_ENUM);
static HSM_COMMUNICATOR_EVENT_ENUM IDLE_stateFn(pSEND_MESSAGE,HSM_COMMUNICATOR_EVENT_ENUM);
static HSM_COMMUNICATOR_EVENT_ENUM AWAITING_ACK_stateFn(pSEND_MESSAGE,HSM_COMMUNICATOR_EVENT_ENUM);

static const SEND_MESSAGE_STATE_FN sendMessage_state_fn_array[sendMessage_numStates] =
{
    UNINITIALIZED_stateFn
    , IDLE_stateFn
    , AWAITING_ACK_stateFn
};

HSM_COMMUNICATOR_EVENT_ENUM THIS(sub_machine_fn)(HSM_COMMUNICATOR_EVENT_ENUM e)
{
    return sendMessageFSM(psendMessage,e);
}

HSM_COMMUNICATOR_SUB_FSM_IF hsmCommunicator_sendMessage_sub_fsm_if =
{
    .subFSM = THIS(sub_machine_fn)
    , .first_event = THIS(firstEvent)
    , .last_event = THIS(noEvent)
};

HSM_COMMUNICATOR_SHARED_EVENT_STR sendMessage_share_hsmCommunicator_INIT_str = {
    .event               = THIS(INIT)
    , .data_translation_fn = UFMN(init_data)
    , .psub_fsm_if         = &hsmCommunicator_sendMessage_sub_fsm_if
};


HSM_COMMUNICATOR_SHARED_EVENT_STR sendMessage_share_hsmCommunicator_MESSAGE_RECEIVED_str = {
    .event               = THIS(MESSAGE_RECEIVED)
    , .data_translation_fn = UFMN(copy_current_message)
    , .psub_fsm_if         = &hsmCommunicator_sendMessage_sub_fsm_if
};

#ifndef INIT_FSM_DATA
#error INIT_FSM_DATA must be defined
```

```
#endif

SEND_MESSAGE sendMessage = {

    INIT_FSM_DATA,
    sendMessage_UNINITIALIZED,
    THIS(INIT),
    &sendMessage_state_fn_array,
    sendMessageFSM
};

pSEND_MESSAGE psendMessage = &sendMessage;


#ifndef EVENT_IS_NOT_EXCLUDED_FROM_LOG
#define EVENT_IS_NOT_EXCLUDED_FROM_LOG(e) (e == e)
#endif
HSM_COMMUNICATOR_EVENT_ENUM sendMessageFSM(pSEND_MESSAGE pfsm, HSM_COMMUNICATOR_EVENT_ENUM event)
{
    HSM_COMMUNICATOR_EVENT_ENUM e = event;

    while ((e != THIS(noEvent))
            && (e >= THIS(firstEvent))
          )
    {

#ifdef HSM_COMMUNICATOR_SEND_MESSAGE_DEBUG
        if ((EVENT_IS_NOT_EXCLUDED_FROM_LOG(e))
                && (e >= THIS(firstEvent))
                && (e < THIS(noEvent))
           )
        {
            DBG_PRINTF("event: %s; state: %s"
                       ,SEND_MESSAGE_EVENT_NAMES[e - THIS(firstEvent)]
                       ,SEND_MESSAGE_STATE_NAMES[pfsm->state]
                      );
        }
#endif

        /* This is read-only data to facilitate error reporting in action functions */
        pfsm->event = e;

        if ((e >= THIS(firstEvent))
                && (e < THIS(noEvent))
           )
        {
            e = ((* (*pfsm->statesArray)[pfsm->state])(pfsm,e));
        }

    }

    return e == THIS(noEvent) ? PARENT(noEvent) : e;
}


static HSM_COMMUNICATOR_EVENT_ENUM UNINITIALIZED_stateFn(pSEND_MESSAGE pfsm,HSM_COMMUNICATOR_EVENT_ENUM e)
{
    HSM_COMMUNICATOR_EVENT_ENUM retVal = THIS(noEvent);

    switch(e)
    {
    case THIS(INIT):
        DBG_PRINTF("hsmCommunicator_sendMessage_noAction");
```

```
        pfsm->state = sendMessage_IDLE;
        break;
    default:
        DBG_PRINTF("hsmCommunicator_sendMessage_noAction");
        break;
    }

    return retVal;
}

static HSM_COMMUNICATOR_EVENT_ENUM IDLE_stateFn(pSEND_MESSAGE pfsm,HSM_COMMUNICATOR_EVENT_ENUM e)
{
    HSM_COMMUNICATOR_EVENT_ENUM retVal = THIS(noEvent);

    switch(e)
    {
    case THIS(SEND_MESSAGE):
        retVal = UFMN(sendMessage)(pfsm);
        pfsm->state = sendMessage_AWAITING_ACK;
        break;
    default:
        DBG_PRINTF("hsmCommunicator_sendMessage_noAction");
        break;
    }

    return retVal;
}

static HSM_COMMUNICATOR_EVENT_ENUM AWAITING_ACK_stateFn(pSEND_MESSAGE pfsm,HSM_COMMUNICATOR_EVENT_ENUM e)
{
    HSM_COMMUNICATOR_EVENT_ENUM retVal = THIS(noEvent);

    switch(e)
    {
    case THIS(ACK):
        retVal = UFMN(checkQueue)(pfsm);
        pfsm->state = sendMessage_IDLE;
        break;
    case THIS(MESSAGE_RECEIVED):
        retVal = UFMN(parseMessage)(pfsm);
        break;
    default:
        DBG_PRINTF("hsmCommunicator_sendMessage_noAction");
        break;
    }

    return retVal;
}



#ifdef HSM_COMMUNICATOR_SEND_MESSAGE_DEBUG
char *SEND_MESSAGE_EVENT_NAMES[] = {

     "hsmCommunicator_sendMessage_INIT"
     ,"hsmCommunicator_sendMessage_MESSAGE_RECEIVED"
     ,"hsmCommunicator_sendMessage_SEND_MESSAGE"
     ,"hsmCommunicator_sendMessage_ACK"
     , "sendMessage_noEvent"
     , "sendMessage_numEvents"
};

char *SEND_MESSAGE_STATE_NAMES[] = {
```

```
    "hsmCommunicator_sendMessage_UNINITIALIZED"
    ,"hsmCommunicator_sendMessage_IDLE"
    ,"hsmCommunicator_sendMessage_AWAITING_ACK"
};

#endif
```

sendMessage_priv.h

```
/**
        sendMessage_priv.h

        This file automatically generated by FSMLang
*/

#ifndef _SENDMESSAGE_PRIV_H_
#define _SENDMESSAGE_PRIV_H_

#include "hsmCommunicator_submach.h"
#include "hsmCommunicator.h"

#ifdef SEND_MESSAGE_DEBUG
#include <stdio.h>
#include <stdlib.h>
#endif

/*
        sub-machine events are included in the top-level machine event enumeration.
        These macros set the appropriate names for events from THIS machine
        and those from the PARENT machine.

        They may be turned off as needed.
*/
#ifndef NO_CONVENIENCE_MACROS
#undef UFMN
#define UFMN(A) hsmCommunicator_sendMessage_##A
#undef THIS
#define THIS(A) hsmCommunicator_sendMessage_##A
#undef PARENT
#define PARENT(A) hsmCommunicator_##A
#endif
#undef STATE
#define STATE(A) sendMessage_##A

#ifdef HSM_COMMUNICATOR_SEND_MESSAGE_DEBUG
extern char *SEND_MESSAGE_EVENT_NAMES[];
extern char *SEND_MESSAGE_STATE_NAMES[];
#endif

typedef enum {
    sendMessage_UNINITIALIZED
    , sendMessage_IDLE
    , sendMessage_AWAITING_ACK
    , sendMessage_numStates
}  SEND_MESSAGE_STATE;

typedef struct _sendMessage_data_struct_ SEND_MESSAGE_DATA, *pSEND_MESSAGE_DATA;
typedef struct _sendMessage_struct_ SEND_MESSAGE, *pSEND_MESSAGE;
#undef FSM_TYPE_PTR
#define FSM_TYPE_PTR pSEND_MESSAGE
```

```
extern SEND_MESSAGE sendMessage;

extern pSEND_MESSAGE psendMessage;

typedef HSM_COMMUNICATOR_EVENT_ENUM (*SEND_MESSAGE_ACTION_FN)(FSM_TYPE_PTR);

typedef HSM_COMMUNICATOR_EVENT_ENUM (*SEND_MESSAGE_FSM)(FSM_TYPE_PTR,HSM_COMMUNICATOR_EVENT_ENUM);

struct _sendMessage_data_struct_ {
    msg_e_t current_msg;
    queue_str_t * pqueue;
};

typedef ACTION_RETURN_TYPE (*SEND_MESSAGE_STATE_FN)(pSEND_MESSAGE,HSM_COMMUNICATOR_EVENT_ENUM);

static const SEND_MESSAGE_STATE_FN sendMessage_state_fn_array[sendMessage_numStates];

struct _sendMessage_struct_ {
    SEND_MESSAGE_DATA             data;
    SEND_MESSAGE_STATE            state;
    HSM_COMMUNICATOR_EVENT_ENUM   event;
    SEND_MESSAGE_STATE_FN   const (*statesArray)[sendMessage_numStates];
    SEND_MESSAGE_FSM              fsm;
};


ACTION_RETURN_TYPE hsmCommunicator_sendMessage_sendMessage(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_sendMessage_checkQueue(FSM_TYPE_PTR);
ACTION_RETURN_TYPE hsmCommunicator_sendMessage_parseMessage(FSM_TYPE_PTR);

void hsmCommunicator_sendMessage_init_data(pHSM_COMMUNICATOR_DATA);
void hsmCommunicator_sendMessage_copy_current_message(pHSM_COMMUNICATOR_DATA);

#endif
```

message-actions.c

Download

```
/*
  message_actions.c

        Action functions for the sendMessage FSM.

        FSMLang (fsm) - A Finite State Machine description language.
        Copyright (C) 4/20/2025  Steven Stanton

        This program is free software; you can redistribute it and/or modify
        it under the terms of the GNU General Public License as published by
        the Free Software Foundation; either version 2 of the License, or
        (at your option) any later version.

        This program is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
        GNU General Public License for more details.

        You should have received a copy of the GNU General Public License
```

```
        along with this program; if not, write to the Free Software
        Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

        Steven Stanton
        sstanton@pesticidesoftware.com

        For the latest on FSMLang: http://fsmlang.github.io

        And, finally, your possession of this source code implies nothing.

*/

#include "sendMessage_priv.h"

ACTION_RETURN_TYPE UFMN(sendMessage)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);

    pfsm->data.pqueue->queue_count--;

    return THIS(noEvent);
}

ACTION_RETURN_TYPE UFMN(parseMessage)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return THIS(ACK);
}

ACTION_RETURN_TYPE UFMN(checkQueue)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);

    return (pfsm->data.pqueue->queue_count > 0) ? THIS(SEND_MESSAGE) : THIS(noEvent);
}

ACTION_RETURN_TYPE UFMN(noAction)(FSM_TYPE_PTR pfsm)
{
    DBG_PRINTF("%s", __func__);
    (void) pfsm;
    return THIS(noEvent);
}

void UFMN(copy_current_message)(pHSM_COMMUNICATOR_DATA pfsm_data)
{
    DBG_PRINTF("%s", __func__);

    psendMessage->data.current_msg = pfsm_data->current_msg;

}

void UFMN(init_data)(pHSM_COMMUNICATOR_DATA pfsm_data)
```

```
{
    DBG_PRINTF("%s", __func__);

    psendMessage->data.pqueue = &pfsm_data->queue;

}
```

Makefile

Download

```
#################################################################
#
#  Makefile for the hsmCommunicator FSMLang example
#

SRC = $(wildcard *actions.c)
FSM_SRC = $(wildcard *.fsm)

FSM_FLAGS=-ts --generate-weak-fns=false --force-generation-of-event-passing-actions

CFLAGS=-DHSM_COMMUNICATOR_DEBUG                   \
       -DHSM_COMMUNICATOR_ESTABLISH_SESSION_DEBUG \
       -DHSM_COMMUNICATOR_SEND_MESSAGE_DEBUG      \
       -g -ggdb

## FSMLang start

.SUFFIXES: .fsm .html .plantuml

FSM ?= fsm

GENERATED_SRC = $(shell $(FSM) -M $(FSM_FLAGS) $(FSM_SRC))
GENERATED_HDR = $(shell $(FSM) -Mh $(FSM_FLAGS) $(FSM_SRC))

cleanfsm:
        @-rm -f $(GENERATED_SRC) 2> /dev/null
        @-rm -f $(GENERATED_HDR) 2> /dev/null
        @-rm -f *.fsmd           2> /dev/null

%.fsmd: %.fsm
        @set -e; $(FSM) -Md $(FSM_FLAGS) $< > $@

.fsm.o:
        @$(FSM) $(FSM_FLAGS) $< > fsmout
        @$(CC) -c $(CFLAGS) $*.c
        @rm -f $*.c

.fsm.c:
        @$(FSM) $(FSM_FLAGS) $< > fsmout

.fsm.h:
        @$(FSM) $(FSM_FLAGS) $< > fsmout
```

```
ifneq ($(MAKECMDGOALS),clean)
-include $(FSM_SRC:.fsm=.fsmd)
endif

## FSMLang end

OBJS=$(SRC:.c=.o) $(GENERATED_SRC:.c=.o)

all: hsmCommunicator

test.out: hsmCommunicator
        ./$< >$@

hsmCommunicator: $(OBJS) $(FSM) Makefile
        $(CC) -o $@ $(LDFLAGS) $(OBJS)
        @-astyle $(GENERATED_SRC) $(GENERATED_HDR)

clean: cleanfsm
        @-rm -f $(OBJS)            2> /dev/null
        @-rm -f hsmCommunicator    2> /dev/null
        @-rm -f test.out           2> /dev/null
        @-rm -f *.orig             2> /dev/null

$(SRC): hsmCommunicator_priv.h
```

Usage and Command Line Options

Usage : `/mnt/c/GitHub/FSMLang2/linux/fsm [-tc|s|e|h|p|r] [-o outfile] <filename>.fsm`

'c'
gets you c code output based on an event/state table,

's'
gets you c code output with individual state functions using switch constructions,

'e'
gets you c code output with a table of functions for each event using switch constructions,

'h'
gets you html output

'p'
gets you PlantUML output

'r'
gets you reStructuredText output

-i0
inhibits the creation of a machine instance any other argument to ‘i’ allows the creation of an instance; this is the default

-c
will create a more compact event/state table when -tc is used with machines having actions which return states

-s
prints some useful statistics and exits

-o <outfile>
will use <outfile> as the filename for the top-level machine output. Any sub-machines will be put into files based on the sub-machine names.

--generate-weak-fns=false
suppresses the generation of weak function stubs.

--short-user-fn-names=true
causes user functions (such as action functions to use only the machine name when the sub-machine depth is 1).

--force-generation-of-event-passing-actions
forces the generation of actions which pass events when weak function generation is disabled. The generated functions are not weak.

--core-logging-only=true
suppresses the generation of debug log messages in all but the core FSM function.

--generate-run-function<=true|false>
this option is deprecated. The run function is always generated; no RUN_STATE_MACHINE macro is provided.

--include-svg-img<=*true|false>
adds <img/> tag referencing <filename>.svg to include an image at the top of the web page.

--css-content-internal=true
puts the CSS directly into the html.

--css-content-filename=<filename>
uses the named file for the css citation, or for the content copy.

--add-plantuml-title=<*true|false>
adds the machine name as a title to the plantuml.

--add-plantuml-legend=<*center|left|right|top|*bottm>
adds a legend to the plantuml. Center, bottom are the defaults. Horizontal and vertial parameters can be added in a quoted string. Center is a horizontal parameter. By default, event, state, and action lists are included in the legend, and event descriptions are removed from the body of the diagram.

--exclude-states-from-plantuml-legend=<*true|false>
excludes state information from the plantuml legend. When excluded from legend, state comments are included in the diagram body.

--exclude-events-from-plantuml-legend=<*true|false>
excludes event information from the plantuml legend.

--exclude-actions-from-plantuml-legend=<*true|false>
excludes action information from the plantuml legend.

--convenience-macros-in-public-header[=<*true|false>]
includes convenience macros (THIS, UFMN, e.g.) in the public header of the top-level machine; otherwise, they are placed in the private header.

--add-machine-name
adds the machine name when using the –short-debug-names option

--add-event-cross-reference<=true|*false>
adds a cross-reference list as a comment block in front of the machine event enumeration. Omitting the optional argument is equivalent to specifying “true”

--event-cross-ref-only<=*true|false>
creates a cross-reference list as a separate file. When the format is not specified by –event-cross-ref-format, json is provided. The file created is <filename>.[json|csv|tab|xml]

--event-cross-ref-format=[json|csv|tab|xml]

specifies the output format for –event-cross-ref-only. Specifying this option obviates –event-cross-ref-only.

--include-state-cross-refs=<*true|false>
Includes cross reference for states. Top-level states are placed in the same file as the events; sub-machines each get their own files.

--add-plantuml-prefix-string=<text>
will add the specified text to the plantuml output before any generated output. This option can be specified multiple times; all text will be added in the order given for the content copy.

--add-plantuml-prefix-file=<text>
will add the text in the specified file to the plantuml output before any generated output. This option can be specified multiple times; all text will be added in the order given for the content copy.

-M
prints the file name(s) of the source files that would have been created to stdout. This is useful in Makefiles for getting the list of files that will be generated (e.g. GENERATED_SRC=$(shell $(FSM) -M -tc $(FSM_SRC))). This option must preceed the -t option.

-Mh
prints the file name(s) of the headers that would have been created to stdout. This is useful in Makefiles for getting the list of files that will be generated (e.g. GENERATED_HDRS=$(shell $(FSM) -M -tc $(FSM_SRC))). This option must preceed the -t option. And, only tc or ts are applicable.

-Md
print a line suitable for inclusion in a Makefile giving the recipe for creating dependent files. This option must preceed the -t option.

--add-profiling-macros<=true|*false>
adds profiling macros at the beginning and end of the FSM function, and before and after invocation of action functions.

--profile-sub-fsms<=true|*false>
adds profiling macros at the beginning and end of the FSM function in sub-machines. Profiling macros must also be enabled.

--empty-cell-fn=<name>
designates a function to be called when an event/state cell is empty.

--inhibiting-states-share-events<=*true|false>
When true, events are shared to sub-machines even in states which inhibit them. The default is to not share. This option allows sharing behavior of version 1.45.1 and before to be preserved.

-v
prints the version and exits

Index

Symbols

C

U