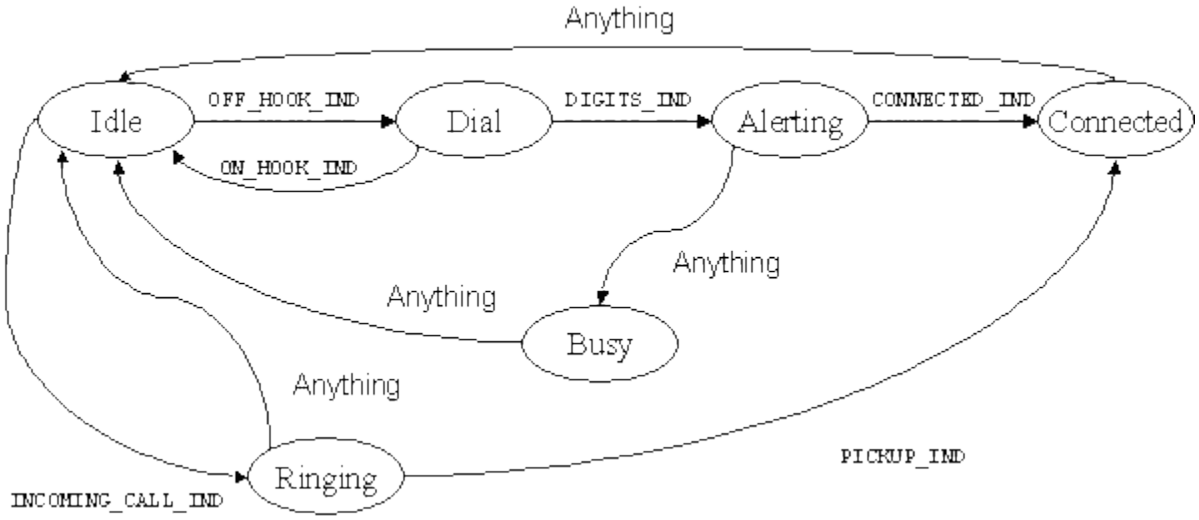
Implementing Efficient State Machines

States machines are a useful design technique for “event driven systems.” Event driven systems have all sorts of applications in the real world (e.g. manufacturing, appliances, and telephones). State machines are especially popular in telephony. Just thinking about the operation of a basic phone call you can easily realize the states your phone has (ringing, busy, idle, etc). In fact, most telephone switch software is implemented as a state machine.

State machines can be implemented in a variety of ways: look-up tables, switch-case blocks and functional states. The following example state machine of a telephone call will be used to demonstrate the various techniques:



This is the state machine for processing a basic telephone call.

Lookup tables are the most common way of implementing state machines. When using lookup tables, each state is identified by a unique integer. A two-dimensional array contains the transition rules. For example:

```
/* States. */
#define  IDLE                   0
#define  DIAL                   1
#define  ALERTING               2
#define  CONNECTED              3
#define  BUSY                   4
#define  RINGING                5

/* Possible inputs    */
#define  ON_HOOK_IND            0
#define  OFF_HOOK_IND           1
#define  DIGITS_IND             2
#define  CONNECTED_IND          3
#define  INCOMING_CALL_IND      4
#define  TIMEOUT_IND            5
#define  PICKUP_IND             6

int transtab[][] = /* transtab[Input][CurrentState] => NextState        */
{
```

```
/* IDLE     DIAL      ALERTING   CONNECTED  BUSY  RINGING       */

{ IDLE,    IDLE,     BUSY,      IDLE,      IDLE, IDLE      }, /*
ON_HOOK_IND       */
{ DIAL,    DIAL,     BUSY,      IDLE,      IDLE, IDLE      }, /*
OFF_HOOK_IND      */
{ IDLE,    ALERTING, IDLE,      IDLE,      IDLE, IDLE      }, /*
DIGITS_IND        */
{ IDLE,    DIAL,     ALERTING,  IDLE,      IDLE, IDLE      }, /*
CONNECTED_IND     */
{ RINGING, DIAL,     IDLE,      CONNECTED, BUSY, RINGING   }, /*
INCOMING_CALL_IND */
{ IDLE,    DIAL,     BUSY,      IDLE,      IDLE, IDLE      }, /*
TIMEOUT_IND       */
{ IDLE,    DIAL,     BUSY,      IDLE,      IDLE, CONNECTED }, /*
PICKUP_IND        */
};
```

There are many variations to this scheme (which axis contains what, how the numbers are arranged, etc). There even exist tools to automatically create state machine tables. The current state is held in an integer variable:

```
int curstate = IDLE;
```

As events (indications) come into the system the state machine is shifted with the following assignment:

```
curstate = transtab[event][curstate];
```

If you study the table for a moment, you can see that for each input (row) in the table every state can transit to another state. The table was derived from the state machine diagram by simply checking for each input what states that input transits to for every state.

State machines implemented this way have some drawbacks:

- They are large (space-wise). For every combination of input and state there is a record of the next state.
- They are difficult to maintain by hand.
- No actions can be performed during tansitions without another action table.

Because of these drawbacks, some programmers resort to using nested switch-case blocks (one for the current state and one for the input indication) instead of a translation table. This takes care of the above problems. However, a performance penalty will arise because of the need to look up the current state number.

A more efficient approach is used that stores an address within the code of a state machine as to where processing is to begin next. Because of the power the C programming language has with function pointers this scheme can be implemented easily and efficiently. This technique also results in easily modified state machines that are easy to understand with a quick glance.

A simple algorithm exists to implement the state machines (like the example) with this method. First, all of the vertices of the state machine (the bubbles) are

functions. Second, the labels on the edges of the state machine (the lines) become the possible input indications to the machine. These can be organized in any number of ways. For the purposes of our example, we will use an enumerated type:

```
/* These are the events that can affect our */
/* StateMachine                             */
typedef enum
{
  ON_HOOK_IND,
  OFF_HOOK_IND,
  DIGITS_IND,
  CALL_PROCEEDING_IND,
  CONNECTED_IND,
  INCOMING_CALL_IND,
  TIMEOUT_IND,
  INVALID_DIGITS_IND,
  BUSY_IND,
  PICKUP_IND
} EIndication;
```

The next step in implementing this machine is to declare the type that is the actual machine (the variable which holds the current state). For example:

```
/* StateMachine is a structure which holds the                */
/* current state pointer.  This is just a forward             */
/* declaration for the next typedef                           */
typedef struct StateMachine TStateMachine;

/* StateProc is the function pointer type that                */
/* is used to represent each state of our machine.            */
/* The StateMachine *sm argument holds the current            */
/* information about the machine most importantly the         */
/* current state.  A StateProc function may receive           */
/* input that forces a change in the current state.  This     */
/* is done by setting the curState member of the StateMachine. */
typedef void (*StateProc)(StateMachine *sm, EIndication input);

/* Now that StateProc is defined, we can define the           */
/* actual layout of StateMachine.  Here we only have the      */
/* current state of the StateMachine.                         */
struct StateMachine
{
  StateProc curState;
};
```

What we have now is an object, called TStateMachine. Code written in each StateProc function should be reentrant to allow multiple TStateMachine instances to be functioning at the same time. Next, we must prototype all of our state functions. This allows us to forward-reference states as we define them. For our example machine, we do the following:

```
/* Here are the prototypes for the different                  */
/* states in our system.  We must prototype                   */
/* these functions to be able to set the                      */
/* curState of the StateMachine.  Our states                  */
/* react to an enumerated type as input.  This                */
/* can be changed to whatever data type (or                   */
```

```
/* types) necessary to report the input stimulus.              */
void Idle(StateMachine      *sm,  EIndication input);
void Dial(StateMachine      *sm,  EIndication input);
void Busy(StateMachine      *sm,  EIndication input);
void Alerting(StateMachine  *sm,  EIndication input);
void Connected(StateMachine *sm,  EIndication input);
void Ringing(StateMachine   *sm,  EIndication input);
```

Next comes writing the actual state functions. Each function can perform any actions it wishes to. Shifting to different states requires only an assignment operation. Each state function (bubble) can then be written to transit on the proper input indication. Any conditional construct in the language may be used. Most common are `switch`/`case` blocks and `if` statements:

```
void Idle(StateMachine *sm, EIndication input)
{
  if (input == OFF_HOOK_IND)    /* On OFF_HOOK          */
    sm->curState = Dial;                /*  - shift to Dial state       */
  else if (input == INCOMING_CALL_IND)  /* On INCOMING_CALL     */
    sm->curState = Ringing;             /*  - shift to Ringing state    */
                                        /* else stay the same   */
}

void Dial(StateMachine *sm, EIndication input)
{
  switch (input)
  {
    case DIGITS_IND:            /* On DIGITS_IND                    */
      sm->curState = Alerting;          /*   - shift to Alerting state */
      break;

    case ON_HOOK_IND:           /* Hang up---go idle.     */
      sm->curState = Idle;
      break;

    default:
      NotifyInvalidInput(input);        /* Send output to LBU     */
      sm->curState = Idle;              /* Invalid, go back to idle
*/
  }
}

void Busy(StateMachine *sm, EIndication input)
{
  sm->curState = Idle;    /* shift to Idle state on anything */
}

void Alerting(StateMachine *sm, EIndication input)
{
  if (input == CONNECT_IND)     /* On CONNECT                     */
    sm->curState = Connect;             /*   - shift to Connect state
*/
  else                          /* else invalid, go back to idle */
    sm->curState = Idle;                /*   - shift to Idle state
*/
}
```

```
void Connect(StateMachine *sm, EIndication input)
{
  sm->curState = Idle;  /*   - shift to Idle state on anything */
}

void Ringing(StateMachine *sm, EIndication input)
{
  if (input == PICKUP_IND)      /* On PICKUP                      */
    sm->curState = Connect;             /*   - shift to Connect state
*/
  else                                  /* else invalid, go back to idle
*/
    sm->curState = Idle;                /*   - shift to Idle state
*/
}
```

That concludes the actual state machine. However, the machine must still be initialized and a method to drive the machine must also be written. Initialization of the machine is simple; any method that can be used to initialize a structure can be used to initialize the machine. Shown here are two examples of how to initialize a state machine. The first is a static initialization of a global variable. The second is a function that creates an initialized state machine.

```
/* Here is a global state machine for some emergency          */
/* device.                                                    */
TStateMachine   EmergencyDevice = { Idle };

/* This function can be called to create a new state machine  */
/* from dynamic memory for devices as they come into service. */
TStateMachine  *NewMachine(void)
{
  TStateMachine  *result;

  /* Allocate storage for the object. */
  result = malloc(sizeof(TStateMachine));

  /* Initialize the fields (in particular, current state). */
  result->curState = Idle;

  /* Return the new machine */
  return result;
}
```

The only remaining task is pushing events through the state machine. For this we will write a fictitious function that routes periphery and dependability messages from a ficticious phone switch to the various state machines:

```
/* Here is the main CP task of our fictitious */
/* switch.                                    */
void MainCPTask(void)
{
  QEVENT  evt;

  for(;;)
  {
    GetEvent(device_task, &evt);
```

```
    if (evt.IsAnEmergency)
    {
      TStateMachine  *machine;

      /* We push events into the machine by calling     */
      /* the current state function. The current        */
      /* state function will adjust the curState        */
      /* member if a shift is necessary.                */
      machine = FindStateMachineFromDevice(evt.device);
      (machine->curState)(machine, evt.indication);
    }
    else /* Do something with the emergency phone. */
    {
      /* Uh-oh. This means that we should                */
      /* cause the emergency phone to ring.              */
      /*                                                 */
      /* Notice that emergencyDevice is a TStateMachine, */
      /* not a pointer to one. Thus, we need a '.'       */
      /* and an ampersand to push the event.             */
      (EmergencyDevice.curState)(&EmergencyDevice,
                                 INCOMING_CALL_IND);
    }
  }
}
```

This flexible method can be used for the construction of all kinds of state machines in C. Furthermore, the state machine code it produces is extremely efficient and compact. The only drawback to this method is that the state can not be easily printed for debugging. Modem debuggers (such as gdb) alleviate this problem though.

It is also possible to test if a machine is in a particular state with a comparison operator, for example:

```
/* This routine determines if a state machine is not in a     */
/* "display-less" state; meaning the user is engaged in some  */
/* activity that requires the display.                        */
BOOL DisplayBusy(TStateMachine *sm)
{
  return
     ((sm->curState != Dial)      && /* For digit viewing        */
      (sm->curState != Connected) && /* For connect time         */
      (sm->curState != Ringing));    /* For calling party number */
}
```

State machines are a powerful programming concept that can solve a large number of programming problems. They should be looked at as an alternative to a spaghetti-pile of code (they are not restricted to protocol stacks).