Finite State Machines. A model for Newton Communications.

Communications between the Newton and other devices tends to be a complex task. Between managing the endpoint, and handling the interactions with the user there is a lot going on. Modeling the communications using a Finite State Machine provides a way of simplifying the task of designing a communications based application.

What exactly is a Finite State Machine

A Finite State Machine (FSM or simply state machine) is a collection of states, events, actions and transitions between states. Here we see an example of a simple FSM:

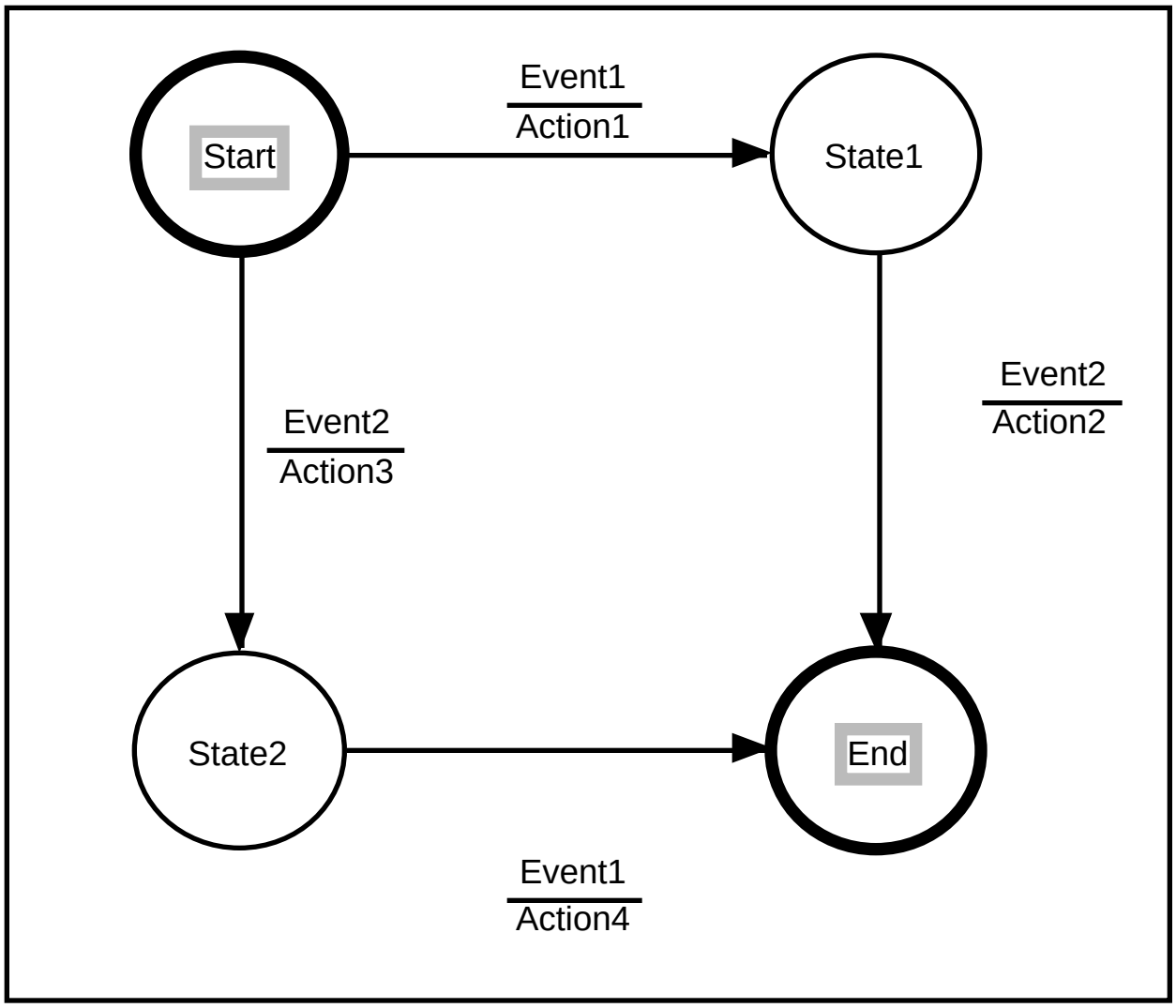


Figure 1: Simple Finite State Machine

This FSM has four states (Start, State1, State2 and End) and responds to two different events (Event1 and Event2). The actions occur when moving from one state to the next. For example, if the FSM is in the Start state and Event1 occurs then Action1 will be performed as the FSM is moving to State1. It is often easier to draw a State Transition Diagram (like Figure 1) than it is to describe in words (and often code) what the actions are.

Given a State Transition Diagram it is easy to create a State Transition Table like the following:

	Event	
Current State	Event1	Event2
Start	(Action1, State1)	(Action3, State2)
State1	-	(Action2, End)
State2	(Action4, End)	-
End	-	-

The State Transition Table is the key to creating FSMs for Newton Communications. With a State Transition Table, NTK and protoFSM, building Finite State Machines is actually quite easy.

protoFSM

ProtoFSM, protoState and protoEvent are a set of user-prototypes that can be used to easily construct Finite State Machines. The state machine, the states and the events are laid out as if a view was being created. The state machine is the parent, with the states as children. Each state contains event children for each event that the state responds to. The event contains an action function and the symbol of the next state to transition to after the action has completed.

The state machine itself has a few additional slots. The `vars` slot is a frame that contains any additional variables that the actions may need to use. An endpoint is a good example of something to put into the `vars` frame because many of the action procedures will need to access the endpoint in a Communications based state machine. There are also slots that reflect the current state, the current event and the current action procedure.[1]

Once a state machine is set up, using it is simply a matter of calling the `doEvent` method of the machine. `DoEvent` takes two parameters. The first is the symbol of the event, the second is an array of additional parameters for the action procedure. The action procedure is invoked in the context of the state machine, with the current state, event and additional parameters passed

[1] ProtoFSM is a Newton DTS Sample that should be available by press time. You can find ProtoFSM and the accompanying sample code on AppleLink and the Newton WWW Site. The next Newton Developer CD will also contain this sample

in. After the action procedure returns the state will be changed according to the `nextState` defined for the event.

Why use a state machine?

Many Newton applications that perform communications have two main 'tasks' operating essentially in parallel.[2] The main 'tasks' of a communications application are user-interface management (and in general the primary operation of the application) and communications management. An example of this separation is the Llama-Talk sample from Newton DTS. This application has user-interface elements to send various kinds of objects over an ADSP connection. The user-interface elements (buttons) queue up requests to send the objects and an idle-script actually performs the communications.

A state-machine runs in a similar fashion. The user-interface elements will typically post events to the state machine based on what the user has requested. This includes things like initiating a connection, disconnecting, sending items, etc. etc. The response to the event (the action procedure) will perform the actual endpoint calls asynchronously with the completion scripts also posting events to indicate the success or failure of the action.

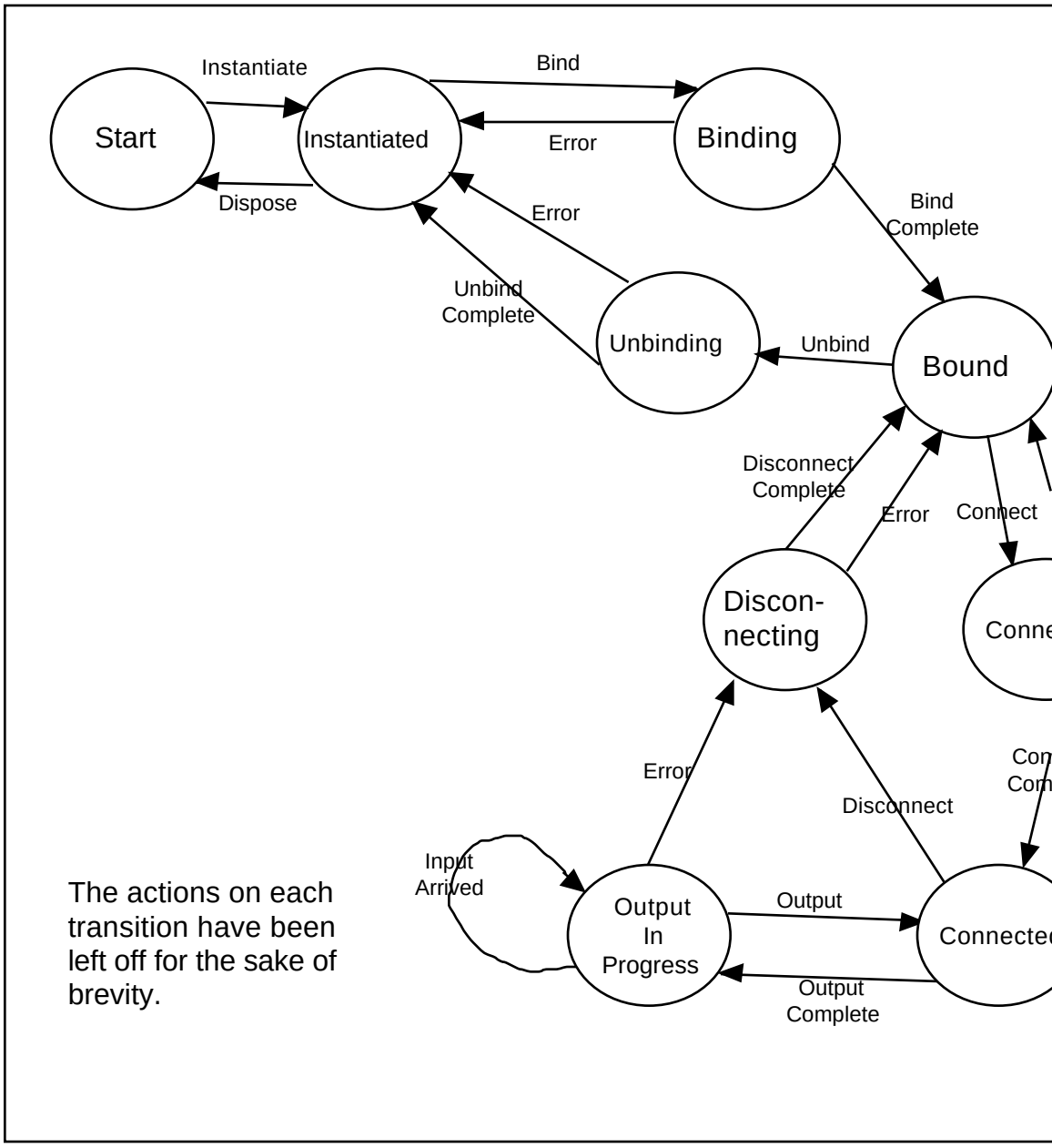
An Example of a state machine

To help illustrate all this, let's have a look at a state machine for doing simple endpoint setup and tear down. The endpoint will establish a serial connection, send and receive simple items, and do a disconnect in response to a user action, or whenever an error occurs. First we need to draw a State Diagram to show the behavior of the endpoint state machine:

---

[2] The word 'task' should not be confused with the idea of any sort of multi-tasking. Though Newton 2.0 OS is a multi-tasking (or more properly *multiprogramming*) operating system, this is not available to NewtonScript-based applications.

The next step is to construct a State Transition Table from this diagram. Again, for the sake of readability, the table is being presented more as an outline. This helps match the form the state machine will take in NTK.



· Start State:
  · Instantiate Event:
    · NextState: Instantiated
    · Action: Create an endpoint frame in vars, call ep:Instantiate()
· Instantiated State:
  · Bind Event:
    · NextState: Binding
    · Action: call ep:Bind asynchronously, the completionScript will post either Error, or BindComplete
  · Dispose Event:
    · NextState: Start
    · Action: call ep:Dispose() and throw away the endpoint frame.

- Binding State:
    - BindComplete Event:
        - NextState: Bound
        - Action: none
    - Error Event:
        - NextState: Instantiated
        - Action: Post a Notify that an error occurred. Could also post a Dispose event from here!
- Bound State:
    - Connect Event:
        - NextState: Connecting
        - Action: call ep:Connect asynchronously, the completionScript will post either Error or ConnectComplete
    - Unbind Event:
        - NextState: Unbinding
        - Action: call ep:Unbind asynchronously, the completionScript will post either Error or UnbindComplete
- Connecting State:
    - ConnectComplete Event:
        - NextState: Connected
        - Action: Setup the inputSpec. The inputScript will post an InputArrived event, the completionScript will post an InputError event if an error occurs.
    - Error Event:
        - NextState: Bound
        - Action: Post a Notify that an error occurred. Could also post an Unbind event from here!
- Connected State:
    - InputArrived Event:
        - NextState: Connected.
        - Action: Handle the input somehow.
    - Output Event:
        - NextState: OutputInProgress
        - Action: Call ep:Output asynchronously. The completionScript will post either an OutputComplete or an Error event.
    - Disconnect Event:
        - NextState: Disconnecting
        - Action: Call ep:Disconnect with the cancel option selected. The completionScript will post either Error or DisconnectComplete.

- OutputInProgress State:
    - OutputComplete Event:
        - NextState: Connected
        - Action: none
    - InputArrived Event:
        - NextState: OutputInProgress
        - Action: Handle the input somehow
    - Error event:
        - NextState: Disconnecting
        - Action: Post a Notify that an error occurred. Call ep:Disconnect with the cancel option selected. The completionScript will post either Error or DisconnectComplete.
- Disconnecting State:
    - DisconnectComplete Event:
        - NextState: Bound
        - Action: none
    - Error Event:
        - NextState: Bound
        - Action: Post a Notify that an error occurred. Could also post an Unbind event from here!
- Unbinding State:
    - UnbindComplete Event:
        - NextState: Instantiated
        - Action: none
    - Error Event:
        - NextState: Instantiated
        - Action: Post a Notify that an error occurred. Could also post a Dispose event from here!

From this table it is easy to lay out the elements of the state machine in NTK. The final thing worth noting is that you can pass parameters to the action procedure along with the event. For example, the Output event could take an additional parameter: the item to be output. The error event could take an additional parameter: an exception frame, etc. In general it is best to make the actions as simple as reasonably possible, and to try to capture as much of the behavior in the states as possible.

Summary

Finite State Machines are a simple and elegant way to model the behavior of applications. Referring to a State Diagram it is easy to see if all contingencies have been handled. ProtoFSM provides a nice clean way to

specify state machines for Newton applications. Communications protocols are often specified in terms of a state machine. Being able to easily transcribe a protocol definition into a finite state machine conveniently will help reduce the development time, and reduce the complexity of the resulting communications code.

Communications on the Newton platform is complex enough, using state machines is a clean way of reducing that complexity to a more manageable level. Because state machines lend themselves better to performing communications asynchronously, applications will gain a performance advantage by using state machines. Synchronous communications calls involve a fair amount of overhead.