Another way to program in parallel

Martin Leadbeater, April 2013

This document forms the notes for a tech-talk about the Clockwork Programming language.

- Clockwork programs are comprised of
    a. definitions of state-machines
    b. a configuration of interconnected state-machine instances

```
LightController MACHINE switch, light {
     RECEIVE switch.on_enter { SET light TO on; }
     RECEIVE switch.off_enter { SET light TO off; }
}
shed_switch INPUT;
shed_light OUTPUT;
shed_light_switch LightController shed_switch, shed_light;
```

Example 1. A Clockwork Program

In Example 1, we have defined a type of machine called a 'LightController' that connects a 'switch' to a 'light'. The LightController simply monitors the switch and turns the light on when the switch is on. The program declares an instance of an 'INPUT' and an 'OUTPUT' machine (not defined in the sample program) and an instance of a LightController, called 'shed_light_switch'. Note a SET statement blocks until the state change is complete.

- Once a machine type is described, it can be used as often as required by creating separate instances

```
porch_sw Input;
porch_light Output;
porch_light_switch LightController porch_sw, porch_light;
```

Example 2. Another Instance of a Machine

- Each instance of a machine executes independently of other machines.
- A machine will generally have its own state
- A machine can only be in a single state at a time
- When a machine enters a state, its 'ENTER' handler, if defined, is executed for that state.

```
LightController MACHINE switch, light {
     on STATE; off STATE;
     RECEIVE switch.on_enter { SET SELF TO on }
```

```
        ENTER switch.off_enter { SET SELF TO off }
        ENTER on { SET light TO on }
        ENTER off { SET light TO off }
}
```

Example 3. Replacement definition of a LightController

In Example 3 we have redefined the LightController Machine to keep its own internal state that follows the state of the switch. The effect is same but the process is slightly different; when the Input switch changes, the LightController sets itself to an equivalent state and then turns the light on or off in response to its own state change. By using its internal state, the light is no longer directly controlled by the physical switch, the light now follows the state of our state machine.

Depending on the hardware devices being used, it may be necessary to debounce a physical switch to avoid flicker. Similarly it may be necessary to ensure a light is on for a minimum time to improve the life of the globe or to make sure the light automatically turns off after a set time.

- Each machine has a timer that automatically tracks the amount of time a machine has been in its current state
- Machines can automatically change state by observing inputs, timers and its own state

```
LightController MACHINE switch, light {
        on WHEN SELF IS on AND TIMER < 10 OR switch IS on;
        off DEFAULT;
        ENTER on { SET light TO on }
        ENTER off { SET light TO off }
}
```

Example 4. Using the timer and state monitoring to debounce a switch

In Example 4, we no longer listen for state changes on the switch, the controller now automatically follows the switch, except that once the controller turns on it will stay on for at least 10ms seconds.

Perhaps the physical switch is a push-button or a motion sensor and the light is expected to simply stay on for a period of time once it has been triggered. In this case, we could implement the LightController to detect the button press and use the timer again to hold the light on.

```
LightController MACHINE switch, light {
     on WHEN SELF IS on AND TIMER < 5000;
     off DEFAULT;
     RECEIVE switch.on_enter { SET SELF TO on }
     ENTER on { SET light TO on }
     ENTER off { SET light TO off }
}
```

Example 5. Holding the light on for a set time of 5 seconds

In the case of a motion sensor, it is useful to reset the timer each time movement is detected.

```
LightController MACHINE sensor, light {
     on WHEN SELF IS on AND TIMER < 5000;
     off DEFAULT;
     RECEIVE sensor.on_enter { SEND reset TO SELF }
     # changes state while executing the reset command
     # this resets the timer
     resetting DURING reset { SET SELF TO on }
     ENTER on { SET light TO on }
     ENTER off { SET light TO off }
}
```

Example 6. Holding the light on for a set time of 5 seconds

Programs can be split so that parts are simpler and more reusable. In the following example, we take the notion of a motion sensor and create a generic ‘delayed off’ machine that we then use to implement the light.

- Clock source can be split into multiple files and included in any order

MyLibrary.cw

```
DelayedOff MACHINE sensor {
      on WHEN SELF IS on AND TIMER < 5000;
      off DEFAULT;
      RECEIVE sensor.on_enter { SEND reset TO SELF }
      resetting DURING reset { SET SELF TO on }
}
```

LightController.cw

```
LightController MACHINE button, light {
      on WHEN button IS on;
      off DEFAULT;
      ENTER on { SET light TO on }

}
```

Shed.cw

```
shed_switch INPUT;
shed_light OUTPUT
dly_off DelayedOff shed_switch;
shed_light_switch LightController dly_off, shed_light;
```

Example 7. Using a generic machine to provide the off-delay logic

Using this technique, a LightController module can now be used with a standard light switch that does not need a debounce filter or it can be used with the debounce feature simply by changing the way the controller is used in shed.cw.

A machine can be instantiated inside another machine to avoid having too many global machine definitions, for example LightController could be implemented as follows:

```
LightController MACHINE switch, light {
      dly_off DelayedOff switch;
      on WHEN dly_off IS on;
      off DEFAULT;
      ENTER on { SET light TO on }
}
```

Machines send and receive messages between themselves and can perform state changes automatically on receipt of the event,

```
Cycle MACHINE {
  a INITIAL;
  b STATE;
  c STATE;

  COMMAND next { }

  TRANSITION a TO b ON next;
  TRANSITION b TO c ON next;
  TRANSITION c TO a ON next;
}
```

Throughout a project, it is useful to build simple machines that provide well-defined functions, for example, here is a machine that latches on when it sees an input turn on:

```
Latch MACHINE input {
      on STATE;
      off INITIAL;
      RECEIVE input.on_enter { SET SELF TO on }
      TRANSITION on TO off ON reset;
}
```

Latches can be useful to measure time elapsed since an input activates or to detect very short pulses.