The next step is to map the FSM graph onto a data structure that can be stored in ROM. Program 10.4 uses an array of structures, where each state is an element of the array, and state transitions are defined as indices to other nodes. The four **Next** parameters define the input-dependent state transitions. The wait times are defined in the software as decimal numbers with units of 10ms, giving a range of 10 ms to about 10 minutes. Using good labels makes the program easier to understand, in other words **goN** is more descriptive than **0**.

The main program begins by specifying the Port E bits 1 and 0 to be inputs and Port B bits 5–0 to be outputs. The initial state is defined as **goN**. The main loop of our controller first outputs the desired light pattern to the six LEDs, waits for the specified amount of time, reads the sensor inputs from Port E, and then switches to the next state depending on the input data. The timer functions were presented earlier as Program 10.2. The function **SysTick_Wait10ms** will wait 10ms times the parameter. Bit-specific addressing will facilitate friendly access to Ports B and E. **SENSOR** accesses PE1–PE0, and **LIGHT** accesses PB5–PB0.


```
#define SENSOR  (*((volatile unsigned long *)0x4002400C))
#define LIGHT   (*((volatile unsigned long *)0x400050FC))
// Linked data structure
struct State {
  unsigned long Out;
  unsigned long Time;
  unsigned long Next[4];};
typedef const struct State STyp;
#define goN   0
#define waitN 1
#define goE   2
#define waitE 3
STyp FSM[4]={
 {0x21,3000,{goN,waitN,goN,waitN}},
 {0x22, 500,{goE,goE,goE,goE}},
 {0x0C,3000,{goE,goE,waitE,waitE}},
 {0x14, 500,{goN,goN,goN,goN}}};
unsigned long S;  // index to the current state
unsigned long Input;
int main(void){ volatile unsigned long delay;
  PLL_Init();       // 80 MHz, Program 10.1
  SysTick_Init();   // Program 10.2
  SYSCTL_RCGC2_R |= 0x12;      // 1) B E
```

```
  delay = SYSCTL_RCGC2_R;      // 2) no need to unlock
  GPIO_PORTE_AMSEL_R &= ~0x03; // 3) disable analog function on PE1-0
  GPIO_PORTE_PCTL_R &= ~0x000000FF; // 4) enable regular GPIO
  GPIO_PORTE_DIR_R &= ~0x03;   // 5) inputs on PE1-0
  GPIO_PORTE_AFSEL_R &= ~0x03; // 6) regular function on PE1-0
  GPIO_PORTE_DEN_R |= 0x03;    // 7) enable digital on PE1-0
  GPIO_PORTB_AMSEL_R &= ~0x3F; // 3) disable analog function on PB5-0
  GPIO_PORTB_PCTL_R &= ~0x00FFFFFF; // 4) enable regular GPIO
  GPIO_PORTB_DIR_R |= 0x3F;    // 5) outputs on PB5-0
  GPIO_PORTB_AFSEL_R &= ~0x3F; // 6) regular function on PB5-0
  GPIO_PORTB_DEN_R |= 0x3F;    // 7) enable digital on PB5-0
  S = goN;
  while(1){
    LIGHT = FSM[S].Out;  // set lights
    SysTick_Wait10ms(FSM[S].Time);
    Input = SENSOR;     // read sensors
    S = FSM[S].Next[Input];
  }
}
```

Program 10.4. Linked data structure implementation of the traffic light controller (C10_TableTrafficLight).

Help

In order to make it easier to understand, which will simplify verification and modification, we have made a 1-to-1 correspondence between the state graph in Figure 10.7 and the **FSM[4]** data structure in Program 10.4. Notice also how this implementation separates the civil engineering policies (the data structure specifies what the machine does), from the computer engineering mechanisms (the executing software specifies how it is done.) Once we have proven the executing software to be operational, we can modify the policies and be confident that the mechanisms will still work. When an accident occurs, we can blame the civil engineer that designed the state graph.

On microcontrollers that have flash, we can place the **FSM** data structure in flash. This allows us to make minor modifications to the finite state machine (add/delete states, change input/output values) by changing the data structure. In this way small modifications/upgrades/options to the finite state machine can be made by reprogramming the flash reusing the hardware external to the microcontroller.

The FSM approach makes it easy to change. To change the wait time for a state, we simply change the value in the data structure. To add more states (e.g., put a red/red state after each yellow state, which will reduce accidents caused by bad drivers running the yellow light), we simply increase the size of the **fsm[]** structure and define the **Out**, **Time**, and **Next** fields for these new states.

To add more output signals (e.g., walk and left turn lights), we simply increase the precision of the **Out** field. To add two more input lines (e.g., wait button, left turn car sensor), we increase the size of the next field to **Next[16]**. Because now there are four input lines, there are 16 possible combinations, where each input possibility requires a **Next** value specifying where to go if this combination occurs. In this simple scheme, the size of the **Next[]** field will be 2 raised to the power of the number of input signals.

CHECKPOINT 10.3

Why is it good to use labels for the states? E.g., why is goN is better than 0.

Hide Answer

To make it easier to understand.

Observation: In order to make the FSM respond quicker, we could implement a time delay function that returns immediately if an alarm condition occurs. If no alarm exists, it waits the specified delay.