The Phase Lock Loop (PLL)

In this section we introduce the phase-lock-loop (PLL), which allows the software to both utilize an accurate crystal and to select how fast the computer executes.

Normally, the execution speed of a microcontroller is determined by an external crystal. The Texas Instruments EK-TM4C123GXL board has a 16 MHz crystal. Most microcontrollers include a phase-lock-loop (PLL) that allows the software to adjust the execution speed of the computer. Typically, the choice of frequency involves the tradeoff between software execution speed and electrical power. In other words, slowing down the bus clock will require less power to operate and generate less heat. Speeding up the bus clock obviously allows for more calculations per second, at the cost of requiring more power to operate and generating more heat.

The default bus speed for the TM4C internal oscillator is 16 MHz ±1%. The internal oscillator is significantly less precise than the crystal, but it requires less power and does not need an external crystal. The TExaS real-board grader has been turning on the PLL, and in this section we will explain how it works. If we wish to have accurate control of time, we will activate the external crystal (called the main oscillator) and use the PLL to select the desired bus speed.

There are two ways to activate the PLL. We could call a library function, or we could access the clock registers directly. In general, using library functions creates a better design because the solution will be more stable (less bugs) and will be more portable (easier to switch microcontrollers). However, the objective of the class being to present microcontroller fundamentals we take the latter approach of showing how to directly access PLL pertinent registers.

An external crystal is attached to the TM4C microcontroller, as shown in Figure 7.1. Table 7.1 shows the clock registers used to define what speed the processor operates. The output of the main oscillator (Main Osc) is a clock at the same frequency as the crystal. By setting the OSCSRC bits to 0, the multiplexer control will select the main oscillator as the clock source.

The main oscillator for the TM4C LaunchPad will be 16 MHz. This means the reference clock (Ref Clk) input to the phase/frequency detector will be 16 MHz. For a 16 MHz crystal, we set the XTAL bits to 10101 (see Table 7.2). In this way, a 400 MHz output of the voltage controlled oscillator (VCO) will yield a 16 MHz clock at the other input of the phase/frequency detector. If the 400 MHz clock is too slow, the **up** signal will add to the charge pump, increasing the input to the VCO, leading to an increase in the 400 MHz frequency. If the 400 MHz clock is too fast, **down** signal will subtract from the charge pump, decreasing the input to the VCO, leading to a decrease in the 400 MHz frequency. Because the reference clock is stable, the feedback loop in the PLL will drive the output to a stable 400 MHz frequency.

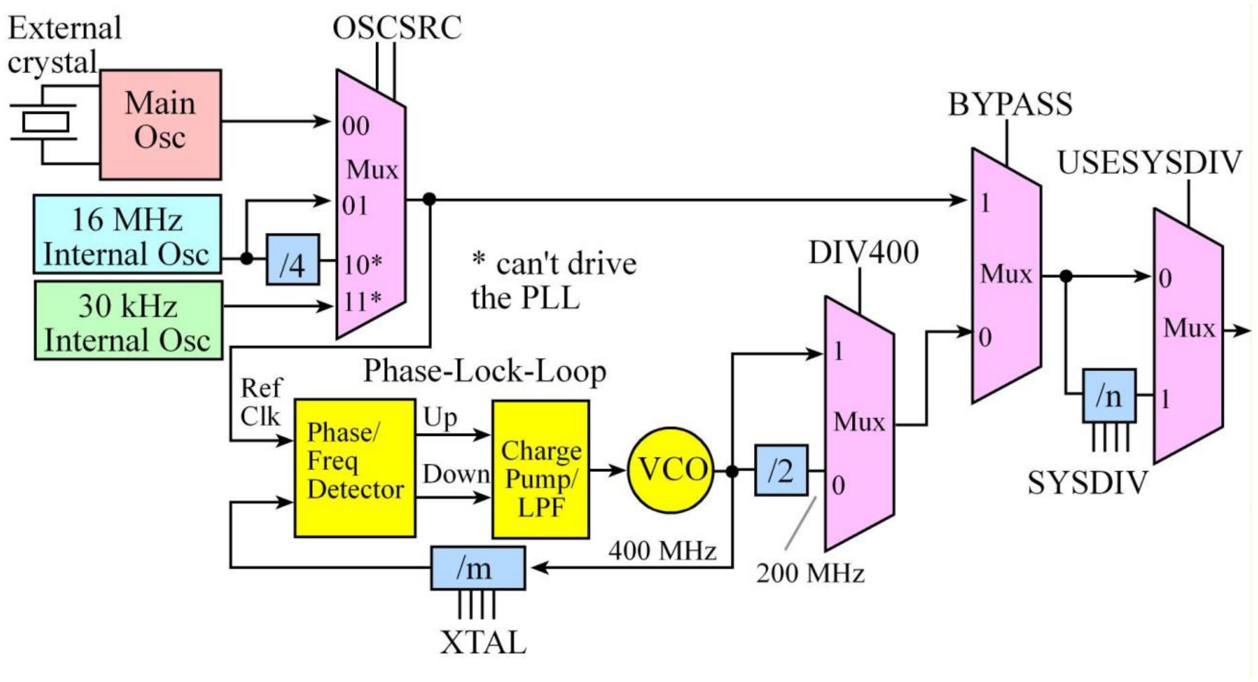


Figure 7.1. Block diagram of the main clock tree on the LM4F/TM4C including the PLL.

Address	26-23	22	13	11	10-6	5-4	Name
$400FE060	SYSDIV	USESYSDIV	PWRDN	BYPASS	XTAL	OSCSRC	SYSCTL_RCC_R
$400FE050					PLLRIS		SYSCTL_RIS_R
	31	30	28-22	13	11	6-4	
$400FE070	USERCC2	DIV400	SYSDIV2	PWRDN2	BYPASS2	OSCSRC2	SYSCTL_RCC2_R

Table 7.1. Main clock registers.

XTAL	*Crystal Freq (MHz)*		*XTAL*	*Crystal Freq (MHz)*
0x0	Reserved		0x10	10.0 MHz
0x1	Reserved		0x11	12.0 MHz
0x2	Reserved		0x12	12.288 MHz
0x3	Reserved		0x13	13.56 MHz
0x4	3.579545 MHz		0x14	14.31818 MHz
0x5	3.6864 MHz		0x15	16.0 MHz
0x6	4 MHz		0x16	16.384 MHz
0x7	4.096 MHz		0x17	18.0 MHz
0x8	4.9152 MHz		0x18	20.0 MHz
0x9	5 MHz		0x19	24.0 MHz
0xA	5.12 MHz		0x1A	25.0 MHz
0xB	6 MHz (reset value)		0x1B	Reserved
0xC	6.144 MHz		0x1C	Reserved
0xD	7.3728 MHz		0x1D	Reserved
0xE	8 MHz		0x1E	Reserved
0xF	8.192 MHz		0x1F	Reserved

Table 7.2. "XTAL" field values of "SYSCTL_RCC_R" register for activating different clock frequencies

Program 7.1 shows the steps 0 to 6 to activate the TM4C123 Launchpad with a 16 MHz main oscillator to run at 80 MHz.

0) Use RCC2 because it provides more options.

1) The first step is to set the BYPASS2 (bit 11) bit. At this point the PLL is bypassed and there is no system clock divider.

2) The second step is to specify the crystal frequency in the four XTAL bits using the code in Table 7.2. The OSCSRC2 bits are cleared to select the main oscillator as the oscillator clock source.

3) The third step is to clear PWRDN2 (bit 13) to activate the PLL.

4) The fourth step is to configure and enable the clock divider using the 7-bit SYSDIV2 field. If the 7-bit SYSDIV2 is **n**, then the clock will be divided by **n**+1. To get the desired 80 MHz from the 400 MHz PLL, we need to divide by 5. So, we place a 4 into the SYSDIV2 field.

5) The fifth step is to wait for the PLL to stabilize by waiting for PLLRIS (bit 6) in the **SYSCTL_RIS_R** to become high.

6) The last step is to connect/enable the PLL by clearing the BYPASS2 bit. To modify this program to operate on other microcontrollers, you will need to change the crystal frequency and the system clock divider

```
void PLL_Init(void){
  // 0) Use RCC2
  SYSCTL_RCC2_R |=  0x80000000;  // USERCC2
  // 1) bypass PLL while initializing
  SYSCTL_RCC2_R |=  0x00000800;  // BYPASS2, PLL bypass
  // 2) select the crystal value and oscillator source
  SYSCTL_RCC_R = (SYSCTL_RCC_R &~0x000007C0)   // clear XTAL field, bits 10-6
                 + 0x00000540;   // 10101, configure for 16 MHz crystal
  SYSCTL_RCC2_R &= ~0x00000070;  // configure for main oscillator source
  // 3) activate PLL by clearing PWRDN
  SYSCTL_RCC2_R &= ~0x00002000;
  // 4) set the desired system divider
  SYSCTL_RCC2_R |= 0x40000000;   // use 400 MHz PLL
  SYSCTL_RCC2_R = (SYSCTL_RCC2_R&~ 0x1FC00000)  // clear system clock divider
                  + (4<<22);      // configure for 80 MHz clock
  // 5) wait for the PLL to lock by polling PLLLRIS
  while((SYSCTL_RIS_R&0x00000040)==0){};  // wait for PLLRIS bit
  // 6) enable use of PLL by clearing BYPASS
  SYSCTL_RCC2_R &= ~0x00000800;
}
```

Program 7.1. Activate the LM4F/TM4C with a 16 MHz crystal to run at 80 MHz

Accurate Time Delays Using SysTick

Having seen how to get an accurate clock setup on our microcontroller lets revisit the SysTick timer, to see how it can be used to create time delays. Recall that the SysTick is a 24-bit timer such that CURRENT counts down every bus cycle. After CURRENT counts to 0, it is automatically reloaded with the RELOAD value and continues to count.

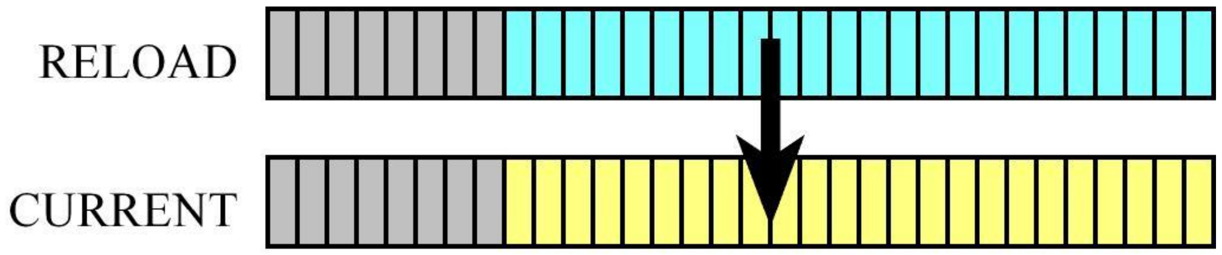


Figure 7.2. SysTick configuration registers

The accuracy of SysTick depends on the accuracy of the clock. When we use the PLL to derive a bus clock based on the 16 MHz crystal, the time measured or generated using SysTick will be very accurate. More specifically, the accuracy of the NX5032GA crystal on the LaunchPad board is ±50 parts per million (PPM), which translates to 0.005%, which is about ±5 seconds per day. One could spend more money on the crystal and improve the accuracy by a factor of 10. Not only are crystals accurate, they are stable. The NX5032GA crystal will vary only ±150 PPM as temperature varies from -40 to +150 ºC. Crystals are more stable than they are accurate, typically varying by less than 5 PPM per year.

Program 7.2 shows a simple function to implement time delays based on SysTick. The RELOAD register is set to the number of bus cycles one wishes to wait. If the PLL function of Program 7.1 has been executed, then the units of this delay will be 12.5 ns (which corresponds to a 80MHz clock). Writing to CURRENT will clear the counter and will clear the count flag (bit 16) of the CTRL register. After SysTick has been decremented delaytimes, the count flag will be set and the while loop will terminate. Since SysTick is only 24 bits, the maximum time one can wait with SysTick_Wait is 2^{24}*12.5ns, which is about 200 ms. To provide longer delays, the function SysTick_Wait10ms calls the function SysTick_Wait repeatedly. Notice that 800,000*12.5ns is 10ms.

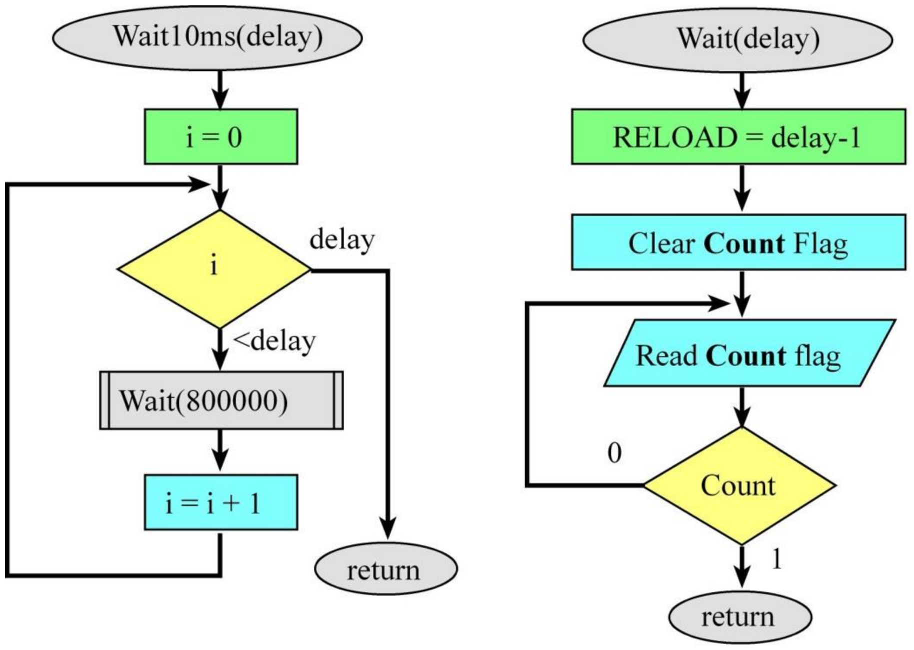


Figure 7.3. Flowchart of program 7.2

```
#define NVIC_ST_CTRL_R      (*((volatile unsigned long *)0xE000E010))
#define NVIC_ST_RELOAD_R    (*((volatile unsigned long *)0xE000E014))
#define NVIC_ST_CURRENT_R   (*((volatile unsigned long *)0xE000E018))
void SysTick_Init(void){
  NVIC_ST_CTRL_R = 0;               // disable SysTick during setup
  NVIC_ST_CTRL_R = 0x00000005;      // enable SysTick with core clock
}
// The delay parameter is in units of the 80 MHz core clock. (12.5 ns)
void SysTick_Wait(unsigned long delay){
  NVIC_ST_RELOAD_R = delay-1;            // number of counts to wait
  NVIC_ST_CURRENT_R = 0;                 // any value written to CURRENT clears
  while((NVIC_ST_CTRL_R&0x00010000)==0){ // wait for count flag
  }
}
// 800000*12.5ns equals 10ms
void SysTick_Wait10ms(unsigned long delay){
  unsigned long i;
  for(i=0; i<delay; i++){
    SysTick_Wait(800000);  // wait 10ms
  }
}
```

Program 7.2. Use of SysTick to delay for a specified amount of time (C10_SysTick_Wait).

Introduction to Structures

A **structure** has elements with different data types and/or precisions. In C, we use **struct** to define a structure. The **const** modifier causes the structure to be allocated in ROM. Without the **const**, the C compiler will place the structure in RAM, allowing it to be dynamically changed. If the structure were to contain an ASCII string of variable length, then we must allocate space to handle its maximum size. In this first example, the structure will be allocated in RAM so no **const** is included. **Structure** is used to implement designed Finite State Machine (FSM) in C code.

In programming an FSM we will use a "structure" to encapsulate the behavior corresponding to each state. We will see that the behavior of a state will involve, the output generated in that state, the time to dwell in that state and the states to transition to on each input. These attributes that encapsulate of the behavior may be of different *data types*, so we need a mechanism to store them in a composite structure. Since, an **array** only allows us to store multiple values of the "same" *data type*, we will need something different than an array. In C, such a composite store is called a **struct**, which allows one to combine multiple elements of different data types into one entity. We will also see how users can create new data types with **typedef**.

In the following example, we will illustrate the code a game designer may use to write a game with objects (called sprites) that move on the screen. A player (could be an enemy ship or a user ship for instance) can be defined by a structure with three attributes. We give separate names to each attribute. In this example the attributes are **Xpos Ypos Score**. The **typedef** command creates a new data type based on the structure, but no memory is allocated.

```c
struct player{
  unsigned char Xpos;      // first element
  unsigned char Ypos;      // second element
  unsigned short Score;    // third element
};
typedef struct player playerType;
```

We can allocate a variable called **Sprite** of this type, which will occupy four bytes (one each for Xpos and Ypos and two for score) in RAM:

```c
playerType Sprite;
```

We can access the individual attributes of this variable using the syntax **name.element**. After the following three lines are executed we have the data structure as shown in Figure 7.4 (assuming the variable occupies the four bytes starting at 0x2000.0250.

```
    Sprite.Xpos = 10;
    Sprite.Ypos = 20;
    Sprite.Score = 12000;
```

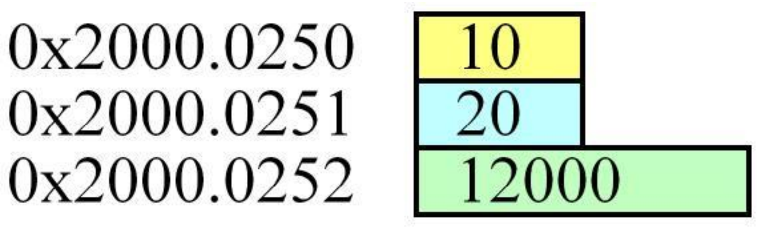


Figure 7.4. A structure collects elements of different sizes and/or types into one object.

We can also have an array of structures. We define structure array in a similar way as other arrays

```
playerType Ships[10];
unsigned long i;
```

While we are accessing an array, we must make sure the index is valid. In this case, the variable **i** must be between 0 and 9. We can read and write the individual fields using the syntax combining array access and structure access.

```
    Ships[i].Xpos = 10;
    Ships[i].Ypos = 20;
    Ships[i].Score = 12000;
```

The C function in Program 7.3 takes a player, moves it to location 50,50 and adds one point. For example, we execute **MoveCenter(6);** to move the 7th ship to location 50,50 and increase its score.

The C function in Program 7.3 takes a player, moves it to location 50,50 and adds one point. For example, we execute **MoveCenter(6);** to move the 7th ship to location 50,50 and increase its score.

```
// move to center and add to score
void MoveCenter(unsigned long i){
  Ships[i].Xpos = 50;
  Ships[i].Ypos = 50;
  if(Ships[i].Score < 65535){
    Ships[i].Score++;
  }
}
```

Program 7.3. A function that accesses a structure.

Observation: Most C compilers will align 16-bit elements within structures to an even address and will align 32-bit elements to a word-aligned address.

Without the **const**, the C compiler will place the structure in RAM, allowing it to be dynamically changed. If the structure resides in RAM, then the system will have to initialize the data structure explicitly by executing software. If the structure is in ROM, we must initialize it at compile time. The next section shows examples of ROM-based structures.

Software **abstraction** allows us to define a complex problem with a set of basic abstract principles. If we can construct our software system using these abstract building blocks, then we have a better understanding of both the problem and its solution. This is because we can separate what we are doing (policies) from the details of how we are getting it done (mechanisms). This separation also makes it easier to optimize. Abstraction provides a proof of correct function and simplifies both extensions and customization. The abstraction presented in this section is the **F**inite **S**tate **M**achine (FSM).

Finite State Machine (FSM)

An FSM can be described by five things that capture the behavior of the system we want to design:

1. Set of inputs the system takes in *Input* $\in$ *{In0, In1, ...}*
2. Set of outputs the system produces *Output* $\in$ *{Out0, Out1...}*
3. Set of states one can find the system in *CurrentState* $\in$ *{S0, S1,...}*
4. The State Transition Graph (STG), which tells us what causes the system to move state to another, *NextState= f(CurrentState ,Input)*
5. Output determination, which tells us what is the corresponding in each state, *Output = g(CurrentState)*

The abstract principles of FSM development are the inputs, outputs, states, and state transitions. The FSM state graph defines the time-dependent relationship between its inputs and outputs. If we can take a complex problem and map it into a FSM model, then we can solve it with simple FSM software tools. Further, our FSM software implementation will be easy to understand, debug, and modify. Other examples of software abstraction include Proportional Integral Derivative digital controllers, fuzzy logic digital controllers, neural networks, and linear systems of differential equations. In each case, the problem is mapped into a well-defined model with a set of abstract yet powerful rules. Then, the software solution is a matter of implementing the rules of the model. In our case, once we prove our software correctly solves one FSM, then we can make changes to the state graph and be confident that our software solution correctly implements the new FSM.

A Finite State Machine (FSM) is an abstraction that describes the solution to a problem very much like an algorithm. Unlike an algorithm which gives a sequence of steps that need to be followed to realize the solution to a problem, an FSM describes the system (the solution being a realization of the system's behavior) as a machine that changes states in reaction to inputs and produces appropriate outputs. Many systems in engineering can be described using an FSM. First let's define what are the essential elements that constitute an FSM. A Finite Statement Machine can be described by these five essential elements:

1. A finite set of states that you can find the system in. One of these states has to be identified as the initial state
2. A finite set of external inputs to the system
3. A finite set of external outputs that the system generates
4. An explicit specification of all state transitions. That is, for every state, what happens (as in, which state will the system transition to) when you are in that state and a specific input occurs?
5. An explicit specification of how the outputs are determined. That is, when does a specific output get generated?

A representation of a system's behavior involves describing all five of these essential elements. Elements 4 and 5 are visually described using a State Transition Graph. We can also state 4 and 5 mathematically as follows:

- Element 4: The next state that the system goes into is a function of the input received and the current state. i.e., *NextState = f(Input, CurrentState)*
- Element 5: The output that the system generates is a function of only the current state. i.e., *Output = g(CurrentState)*

A State Transition Graph (STG) has nodes and edges, where the nodes relate to the states of the FSM and the edges represent the transitions from one state to another when a particular input is received. Edges are accordingly labeled with the input that caused the transition. The output can also be captured in the Graph. Note that a FSM where the output is only dependent on the current state and not the input is called a **Moore FSM**. FSMs where the output is dependent on both the current state and the input are called **Mealy FSMs** i.e.,

Output = h(Input, CurrentState)

We note that some systems lend themselves better to a Mealy description while others are more naturally expressed as Moore machines. However, both machine descriptions are equivalent in that any system that can be described using a Mealy machine can also be expressed equivalently as a Moore machine and vice versa. See Figure 7.5.

The FSM controller employs a well-defined model or framework with which we solve our problem. STG will be specified using either a linked or table data structure. An important aspect of this method is to create a 1-1 mapping from the STG into the data structure. The three advantages of this abstraction are 1) it can be faster to develop because many of the building blocks preexist; 2) it is easier to debug (prove correct) because it separates conceptual issues from implementation; and 3) it is easier to change.

When designing an FSM, we begin by defining what constitutes a state. In a simple system like a single intersection traffic light, a state might be defined as the pattern of lights (i.e., which lights are on and which are off). In a more sophisticated traffic controller, what it means to be in a state might also include information about traffic volume at this and other adjacent intersections. The next step is to make a list of the various states in which the system might exist. As in all designs, we add outputs so the system can affect the external environment, and inputs so the system can collect information about its environment or receive commands as needed. The execution of a Moore FSM repeats this sequence over and over:

1. Perform output, which depends on the current state
2. Wait a prescribed amount of time (optional state dwell time)
3. Input
4. Go to next state, which depends on the input and the current state

The execution of a Mealy FSM repeats this sequence over and over

1. Wait a prescribed amount of time (optional)
2. Input
3. Perform output, which depends on the input and the current state

4. Go to next state, which depends on the input and the current state

There are other possible execution sequences. Therefore, it is important to document the sequence before the state graph is drawn. The high-level behavior of the system is defined by the state graph. The states are drawn as circles. Descriptive states names help explain what the machine is doing. Arrows are drawn from one state to another, and labeled with the input value causing that state transition.

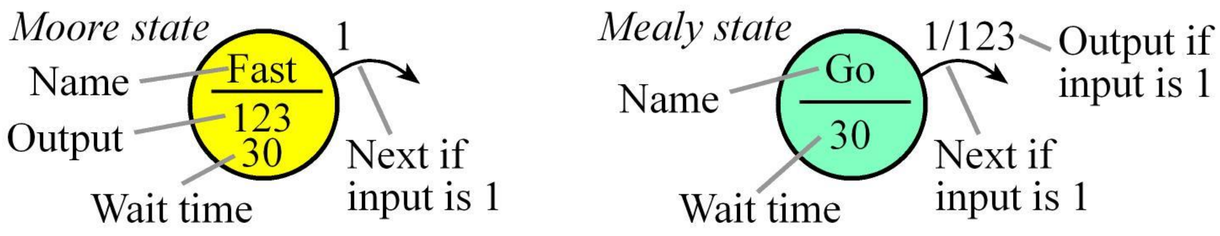
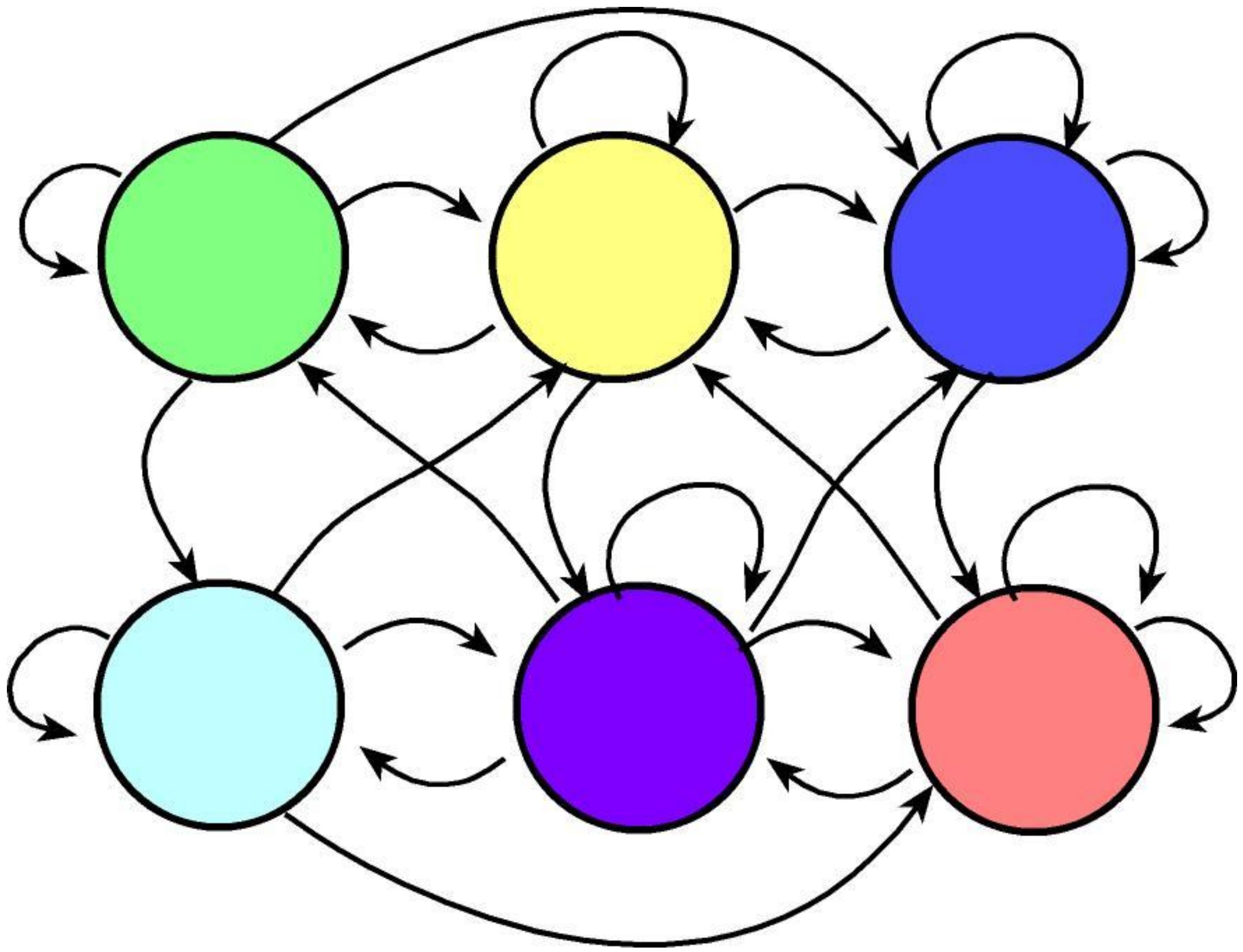


Figure 7.5. The output in a Moore depends just on the state. In a Mealy the output depends on state and input.

Observation: If the machine is such that a specific output value is necessary “to be a state”, then a Moore implementation will be more appropriate.

Observation: If the machine is such that no specific output value is necessary “to be a state”, but rather the output is required to transition the machine from one state to the next, then a Mealy implementation will be more appropriate.

A **linked structure** consists of multiple identically-structured nodes. Each node of the linked structure defines one state. One or more of the entries in the node is a link to other nodes. In an embedded system, we usually use statically-allocated fixed-size linked structures, which are defined at compile time and exist throughout the life of the software. In a simple embedded system the state graph is fixed, so we can store the linked data structure in nonvolatile memory. For complex systems where the control functions change dynamically (e.g., the state graph itself varies over time), we could implement dynamically-allocated linked structures, which are constructed at run time and number of nodes can grow and shrink in time. The FSMs we discuss here will be static. We will use a table structure to define the state graph, which consists of contiguous multiple identically-structured elements. Each element of the table defines one state. One or more of the entries is an index to other elements. The index is essentially a link to another state. An important factor when implementing FSMs is that there should be a clear and one-to-one mapping between the FSM state graph and the linked data structure in software. That is, there should be one element of the structure for each state. If each state has four arrows, then each node of the linked structure should have four links.



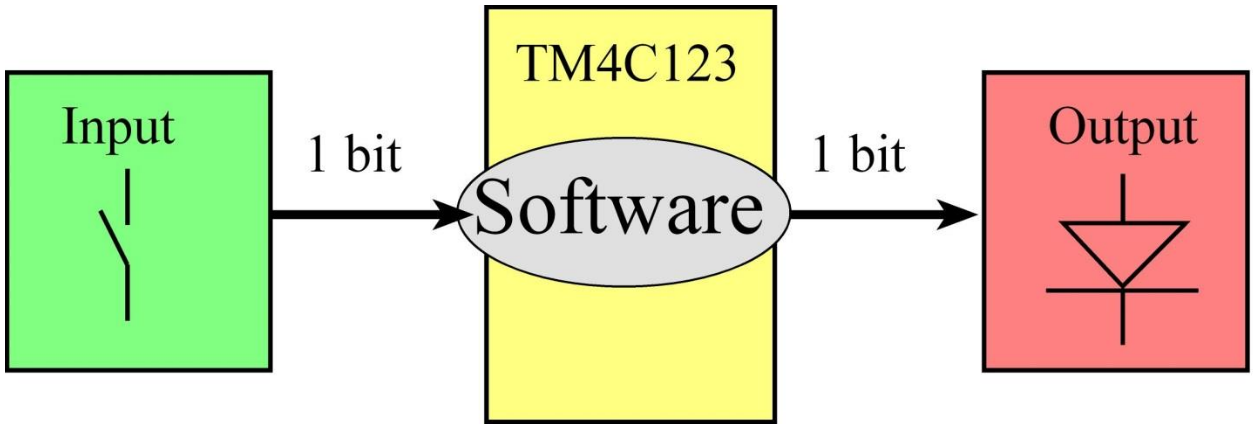
Linked structure with 6 nodes and each node has 4 links

The outputs of a Moore FSM are only a function of the current state. In contrast, the outputs are a function of both the input and the current state in a Mealy FSM. Often, in a Moore FSM, the specific output pattern defines what it means to be in the current state. In the following videos we take a simplistic system where we wish to detect if an input stream has an odd number of 1s so far. It does not lend itself to an easy implementation because we do not have a notion of how long a bit has to persist for it to be inferred as the same bit as opposed to a new bit. It serves though as a simple example. The later examples are more rigorously specified and therefore lend themselves to proper implementations.

Example 7.0. Design a system with one binary input and one binary output. The system should read the input every second. The output should be 1 if the number of received input bits has been odd, and should be 0 if the number of input bits has been even.

Solution: We begin this example by translating the FSM state graph into a **structure**. We define as a requirement that this FSM should run every 1 second. The steps will be output, wait 1 second, input, and go to next state.

Solution: We begin this example by translating the FSM state graph into a **structure**. We define as a requirement that this FSM should run every 1 second. The steps will be output, wait 1 second, input, and go to next state.



Next, we write the engine to execute the FSM, called the **FSM controller**. The FSM controller has 4 steps
1) Output, which depends only on state
2) Wait, which depends only on state
3) Input
4) Go to next state, which depends on input and current state

The solution can be found as part of the TExaS installation at **\Keil\ EECS3100ware\C10_Odd1sDetector**

Example 7.1. Design a traffic light controller for the intersection of two equally busy one-way streets. The goal is to maximize traffic flow, minimize waiting time at a red light, and avoid accidents.

Solution: The intersection has two one-ways roads with the same amount of traffic: North and East, as shown in Figure 7.6. Controlling traffic is a good example because we all know what is supposed to happen at the intersection of two busy one-way streets. We begin the design defining what constitutes a state. In this system, a state describes which road has authority to cross the intersection. The basic idea, of course, is to prevent southbound cars to enter the intersection at the same time as westbound cars. In this system, the light pattern defines which road has right of way over the other. Since an output pattern to the lights is necessary to remain in a state, we will solve this system with a Moore FSM. It will have two inputs (car sensors on North and East roads) and six outputs (one for each light in the traffic signal.) The six traffic lights are interfaced to Port B bits 5–0, and the two sensors are connected to Port E bits 1–0,

PE1=0, PE0=0 means no cars exist on either road
PE1=0, PE0=1 means there are cars on the East road
PE1=1, PE0=0 means there are cars on the North road
PE1=1, PE0=1 means there are cars on both roads

The next step in designing the FSM is to create some states. Again, the Moore implementation was chosen because the output pattern (which lights are on) defines which state we are in. Each state is given a symbolic name:

goN,	PB5-0 = 100001 makes it green on North and red on East
waitN,	PB5-0 = 100010 makes it yellow on North and red on East
goE,	PB5-0 = 001100 makes it red on North and green on East
waitE,	PB5-0 = 010100 makes it red on North and yellow on East

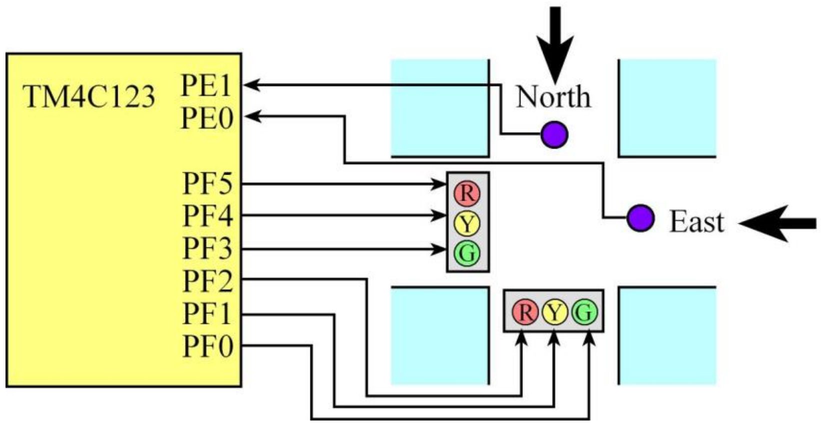


Figure 7.6. Traffic light interface with two sensors and 6 lights.

The output pattern for each state is drawn inside the state circle. The time to wait for each state is also included. How the machine operates will be dictated by the input-dependent state transitions. We create decision rules defining what to do for each possible input and for each state. For this design we can list heuristics describing how the traffic light is to operate:

If no cars are coming, stay in a green state, but which one doesn't matter. To change from green to red, implement a yellow light of exactly 5 seconds. Green lights will last at least 30 seconds. If cars are only coming in one direction, move to and stay green in that direction.
If cars are coming in both directions, cycle through all four states.

Before we draw the state graph, we need to decide on the sequence of operations.

1. Initialize timer and direction registers
2. Specify initial state
3. Perform FSM controller
    a) Output to traffic lights, which depends on the state
    b) Delay, which depends on the state
    c) Input from sensors
    d) Change states, which depends on the state and the input

We implement the heuristics by defining the state transitions, as illustrated in Figure 7.7. Instead of using a graph to define the finite state machine, we could have used a table, as shown in Table 7.2.

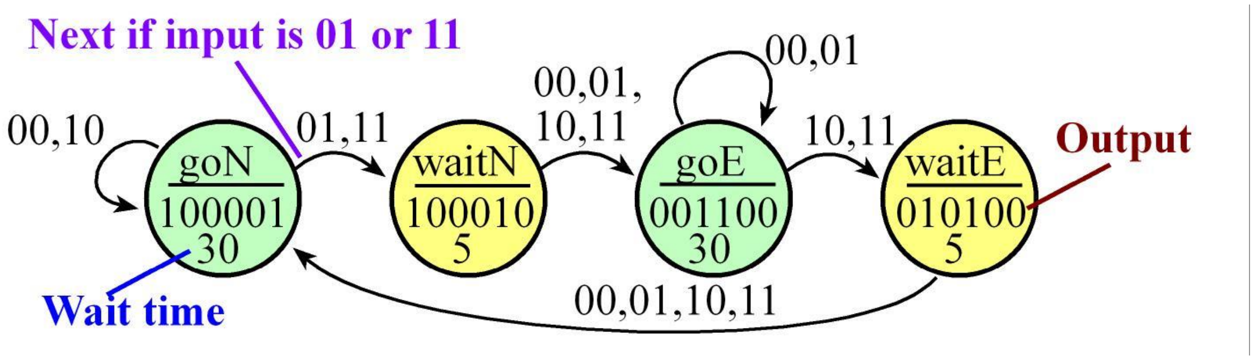


Figure 7.7. State Transition Graph (STG) of a Moore FSM that implements a traffic light.

A **state transition table** has exactly the same information as the state transition graph, but in tabular form. The first column specifies the state number, which we will number sequentially from 0. Each state has a descriptive name. The "Lights" column defines the output patterns for six traffic lights. The "Time" column is the time to wait with this output. The last four columns will be the next states for each possible input pattern.

Num	Name	Lights	Time	In=0	In=1	In=2	In=3
0	goN	100001	30	goN	waitN	goN	waitN
1	waitN	100010	5	goE	goE	goE	goE
2	goE	001100	30	goE	goE	waitE	waitE
3	waitE	010100	5	goN	goN	goN	goN

Table 7.2. Tabular form of a Moore FSM that implements a traffic light.

The next step is to map the FSM graph onto a data structure that can be stored in ROM. Program 7.4 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 7.2. The function **SysTick_Wait10ms** will wait 10 ms times the parameter. Bit-specific addressing will facilitate friendly access to Ports B and E. **SENSOR** accesses PE1–PE0, and **LIGHT** accesses PB5–PB0.

```c
#define SENSOR  (*((volatile unsigned long *)0x4002400C))
#define LIGHT   (*((volatile unsigned long *)0x400050FC))
// Linked data structure
struct State {
  unsigned long Out;  // 6-bit pattern to output
  unsigned long Time; // delay in 10ms units
  unsigned long Next[4];}; // next state for inputs 0,1,2,3
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 7.4. Linked data structure implementation of the traffic light controller (Code Location: \Keil\ EECS3100ware\C10_TableTrafficLight).

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 7.7 and the **FSM[4]** data structure in Program 7.4. 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.

We store our code into the **flash EEPROM** on the TM4C123, which for this example includes both the machine instructions of the program and the **FSM** data structure. However, some microcontrollers have both **permanent ROM** (which can be written once at the factory) and flash EEPROM (which can be erased and reprogrammed over and over). On microcontrollers that have ROM/EEROM combination, we can place the program in ROM and the **FSM** data structure in flash EEPROM. 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.

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.

OPTIONAL READING

The following sections are advanced examples, and utilize pointers.

Example 7.2. Design vending machine with two outputs (soda, change) and two inputs (dime, nickel).

Solution: This vending machine example illustrates additional flexibility that we can build into our FSM implementations. In particular, rather than simple digital inputs, we will create an input function that returns the current values of the inputs. Similarly, rather than simple digital outputs, we will implement general functions for each state. We could have solved this particular vending machine using the approach in the previous example, but this approach provides an alternative mechanism when the input and/or output operations become complex. Our simple vending machine has two coin sensors, one for dimes and one for nickels, see Figure 7.8. When a coin falls through a slot in the front of the machine, light from the QEB1134 sensor reflects off the coin and is recognized back at the sensor. An op amp (OPA2350) creates a digital high at the Port E input whenever a coin is reflecting light. So as the coin passes the sensor, a pulse (V2) is created. The two coin sensors will be inputs to the FSM. If the digital input is high (1), this means there is a coin currently falling through the slot. When a coin is inserted into the machine, the sensor goes high, then low. Because of the nature of vending machines we will assume there cannot be both a nickel and a dime at the same time. This means the FSM input can be 0, 1, or 2. To implement the soda and change dispensers, we will interface two solenoids to Port B. The coil current of the solenoids is less than 40 mA, so we can use the 7406 open collector driver. If the software makes PB0 high, waits 10ms, then makes PB0 low, one soda will be dispensed. If the software makes PB1 high, waits 10ms, then makes PB1 low, one nickel will be returned.

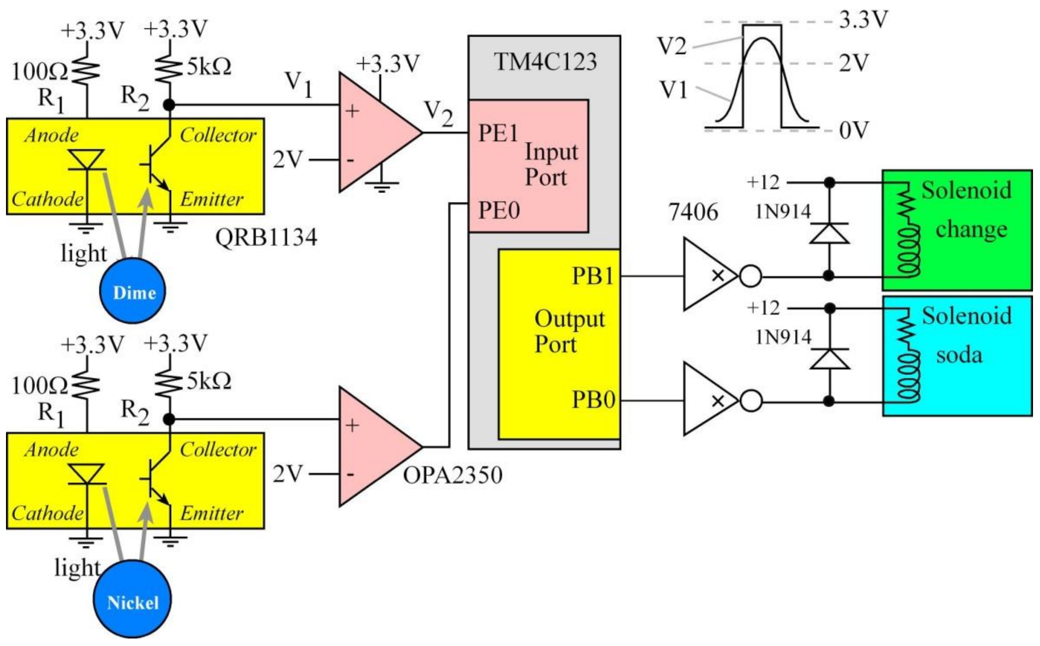


Figure 7.8. A vending machine interfaced to a microcontroller.

We need to decide on the sequence of operations before we draw the state graph.

1) Initialize timer and directions registers
2) Specify initial state
3) Perform FSM controller
    a) Call an output function, which depends on the state
    b) Delay, which depends on the state
    c) Call an input function to get the status of the coin sensors
    d) Change states, which depends on the state and the input.

Figure 7.9 shows the Moore FSM that implements the vending machine. A soda costs 15 cents, and the machine accepts nickels (5 cents) and dimes (10 cents). We have an input sensor to detect nickels (bit 0) and an input sensor to detect dimes (bit 1.) We choose the wait time in each state to be 20ms, which smaller than the time it takes the coin to pass by the sensor. Waiting in each state will debounce the sensor, preventing multiple counting of a single event. Notice that we wait in all states, because the sensor may bounce both on touch and release. Each state also has a function to execute. The function **Soda** will trigger the Port B output so that a soda is dispensed. Similarly, the function **Change** will trigger the Port B output so that a nickel is returned. The **M** states refer to the amount of collected money. When we are in a **W** state, we have collected that much money, but we're still waiting for the last coin to pass the sensor. For example, we start with no money in state **M0**. If we insert a dime, the input will go 10_2, and our state machine will jump to state **W10**. We will stay in state **W10** until the dime passes by the coin sensor. In particular when the input goes to 00, then we go to state **M10**. If we insert a second dime, the input will go 10_2, and our state machine will jump to state **W20**. Again, we will stay in state **W20** until this dime passes. When the input goes to 00, then we go to state **M20**. Now we call the function **change** and jump to state **M15**. Lastly, we call the function **Soda** and jump back to state **M0**.

Since this is a layered system, we will begin by designing the low-level input/output functions that handle the operation of the sensors and solenoid, see Program 7.5. The bit-specific addressing **COINS** provides friendly access to PE1 and PE0, **CHANGE** provides friendly access to PB1, and **SODA** provides friendly access to PB0. The initialization specifies Port E bits 1 and 0 to be input and the Port B bits 1 and 0 to be outputs. The PLL and SysTick are also initialized.

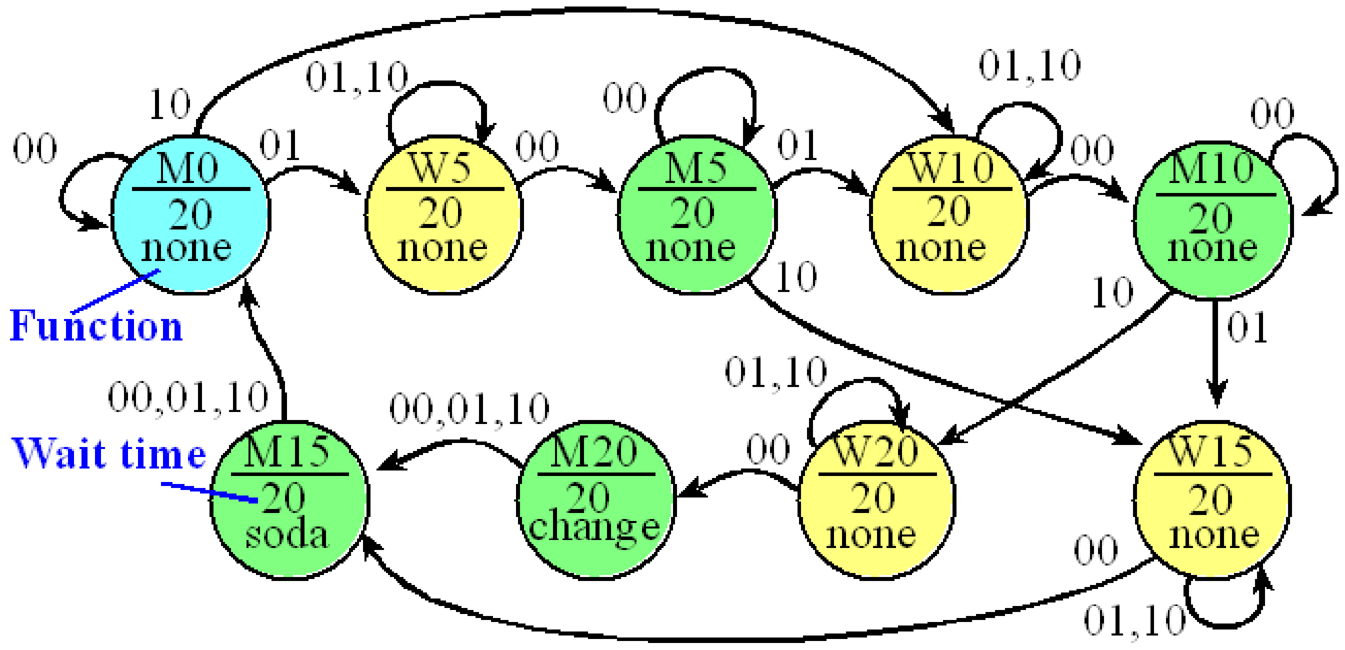


Figure 7.9. This Moore FSM implements a vending machine.

```
#define T10ms 800000
#define T20ms 1600000
#define COINS   (*((volatile unsigned long *)0x4002400C))  // PE1,PE0
#define SODA    (*((volatile unsigned long *)0x40005004))  // PB0
#define CHANGE  (*((volatile unsigned long *)0x40005008))  // PB1
void FSM_Init(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 &= ~0x000000FF; // 4) enable regular GPIO
  GPIO_PORTB_DIR_R |= 0x03;    // 5) outputs on PB1-0
  GPIO_PORTB_AFSEL_R &= ~0x03; // 6) regular function on PB1-0
  GPIO_PORTB_DEN_R |= 0x03;    // 7) enable digital on PB1-0
  SODA = 0; CHANGE = 0;
}
unsigned long Coin_Input(void){
  return COINS;  // PE1,0 can be 0, 1, or 2
}
void Solenoid_None(void){
};
void Solenoid_Soda(void){
  SODA = 0x01;           // activate solenoid, PB0
  SysTick_Wait(T10ms);   // 10 msec, dispenses a delicious soda
  SODA = 0x00;           // deactivate
}
void Solenoid_Change(void){
  CHANGE = 0x02;         // activate solenoid, PB1
  SysTick_Wait(T10ms);   // 10 msec, return 5 cents
  CHANGE = 0x00;         // deactivate
}
```

Program 7.5. Low-level input/output functions for the vending machine.

The initial state is defined as **M0**. Our controller software first calls the function for this state, waits for the specified amount of time, reads the sensor inputs from Port B, then switches to the next state depending on the input data. Notice again the 1-to-1 correspondence between the state graph in Figure 7.9 and the data structure in Program 7.6.

```
struct State {
  void (*CmdPt)(void);   // output function
  unsigned long Time;    // wait time, 12.5ns units
  unsigned long Next[3];};
typedef const struct State StateType;
#define M0  0
#define W5  1
#define M5  2
#define W10 3
#define M10 4
#define W15 5
#define M15 6
#define W20 7
#define M20 8
StateType FSM[9]={
  {&Solenoid_None,  T20ms,{M0,W5,W10}},      // M0, no money
  {&Solenoid_None,  T20ms,{M5,W5,W5}},       // W5, seeing a nickel
  {&Solenoid_None,  T20ms,{M5,W10,W15}},     // M5, have 5 cents
  {&Solenoid_None,  T20ms,{M10,W10,W10}},    // W10, seeing a dime
  {&Solenoid_None,  T20ms,{M10,W15,W20}},    // M10, have 10 cents
  {&Solenoid_None,  T20ms,{M15,W15,W15}},    // W15, seeing something
  {&Solenoid_Soda,  T20ms,{M0,M0,M0}},       // M15, have 15 cents
  {&Solenoid_None,  T20ms,{M20,W20,W20}},    // W20, seeing dime
  {&Solenoid_Change,T20ms,{M15,M15,M15}}};   // M20, have 20 cents
unsigned long S; // index into current state
unsigned long Input;
int main(void){
  FSM_Init();
  S = M0;         // Initial State
  while(1){
    (FSM[S].CmdPt)();           // call output function
    SysTick_Wait(FSM[S].Time);  // wait Program 10.2
    Input = Coin_Input();       // input can be 0,1,2
    S = FSM[S].Next[Input];     // next
  }
}
```

Program 7.6. Vending machine controller.

Alternate source for the following interactive http://users.ece.utexas.edu/~valvano/Volume1/E-Book/C10_Interactives.htm#ITool10.1

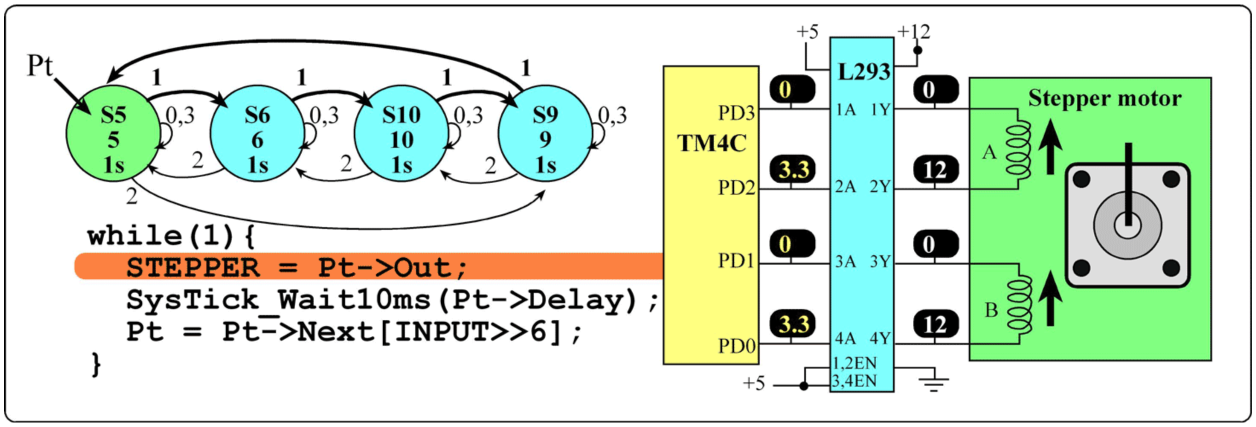
Click on the Deposit 5¢ button to add 5 cents. Click on the Deposit 10¢ button to add 10 cents. Sodas cost 15¢.

Stepper Motor

A motor can be evaluated in terms of its maximum speed (RPM), its torque (N-m), and the efficiency in which it translates electrical power into mechanical power. Sometimes however, we wish to use a motor to control the rotational position (θ=motor shaft angle) rather than to control the rotational speed ($\omega=d\theta/dt$). Stepper motorsare used in applications where precise positioning is more important than high RPM, high torque, or high efficiency. Stepper motors are very popular for microcontroller-based embedded systems because of their inherent digital interface. This next image shows a stepper motor. One input makes it spin clockwise, the second input makes it spin counter clockwise.

In this example there is a 2-input/4-output FSM controlling a stepper motor. When the input is 1, the motor will spin clockwise at 7.5 RPM. To spin clockwise the output pattern will repeat 5, 6, 10, 9, 5, 6, 10, 9, ... For a stepper motor each time a new output is sent, the motor makes one step. For this motor each step will move the motor 45 degrees. It will take 8 outputs to rotate this motor once. If there is a 1 second delay in each state, the motor spins once every 8 seconds or 7.5 RPM.

When the input is 2, the motor will spin counter-clockwise at 7.5 RPM. To spin counter-clockwise the output pattern will repeat 5, 9, 10, 6, 5, 9, 10, 6, ... If the input is 0 or 3, the motor will stop. We stop a stepper by leaving the output at any of the valid states 5, 6, 10 or 9.



```
while(1){
  STEPPER = Pt->Out;
  SysTick_Wait10ms(Pt->Delay);
  Pt = Pt->Next[INPUT>>6];
}
```

In order to animate the picture, visit the following link - https://d37djvu3ytnwxt.cloudfront.net/asset-v1:UTAustinX+UT.6.03x+1T2016+type@asset+block/stepper2.gif

```c
// PD7,PD6 are inputs from switches
// PD3-PD0 are outputs to the stepper motor
#define T1sec 100
struct State{
  uint8_t Out;
  uint32_t Delay;
  const struct State *Next[4];
};
typedef const struct State StateType;
#define S5  &fsm[0]
#define S6  &fsm[1]
#define S10 &fsm[2]
#define S9  &fsm[3]
StateType fsm[4]={
  {5, T1sec, S5, S6, S9, S5},
  {6, T1sec, S6,S10, S5, S6},
  {10,T1sec,S10, S9, S6,S10},
  {9, T1sec, S9, S5,S10, S9}
};
const struct State *Pt;  // Current State
#define STEPPER  (*((volatile uint32_t *)0x4000703C))
#define INPUT  (*((volatile uint32_t *)0x40007300))
int main(void){
  PLL_Init();               // Program 4.6
  SysTick_Init();           // Program 4.7
  SYSCTL_RCGC2_R |= 0x08;   // 1) port D clock enabled
  Pt = &fsm[0];
  GPIO_PORTD_LOCK_R = 0x4C4F434B;  // unlock GPIO Port D
  GPIO_PORTD_CR_R = 0xFF;          // allow changes to PD7-0
  GPIO_PORTD_AMSEL_R &= ~0xCF;      // 3) disable analog function
  GPIO_PORTD_PCTL_R &= ~0xFF00FFFF; // 4) GPIO
  GPIO_PORTD_DIR_R |= 0x0F;   // 5) make PD3-0 out
  GPIO_PORTD_DIR_R &= ~0xC0;  //    make PD7-6 input
  GPIO_PORTD_AFSEL_R &= ~0xCF;// 6) disable alt func on PD7-6,3-0
  GPIO_PORTD_DR8R_R |= 0x0F;  // enable 8 mA drive on PD7-6,3-0
  GPIO_PORTD_DEN_R |= 0xCF;   // 7) enable digital I/O on PD7-6,3-0
  while(1){
    STEPPER = Pt->Out;          // Output
    SysTick_Wait10ms(Pt->Delay); // Wait
    Pt = Pt->Next[INPUT>>6];     // Next
  }
}
```

Larger motors provide more torque, but require more current. It is easy for a computer to control both the position and velocity of a stepper motor in an open-loop fashion. Although the cost of a stepper motor is typically higher than an equivalent DC permanent magnetic field motor, the overall system cost is reduced because stepper motors may not require feedback sensors. They are used in printers to move paper and print heads, tapes/disks to position read/write heads, and high-precision robots.

A bipolar stepper motor has two coils on the stator (the frame of the motor), labeled **A** and **B** in Figure 7.10. Typically, there is always current flowing through both coils. When current flows through both coils, the motor does not spin (it remains locked at that shaft angle). Stepper motors are rated in their holding torque, which is their ability to hold stationary against a rotational force (torque) when current is constantly flowing through both coils. To move a bipolar stepper, we reverse the direction of current through one (not both) of the coils, see Figure 7.10. To move it again we reverse the direction of current in the other coil. Remember, current is always flowing through both coils. Let the direction of the current be signified by up and down arrows in Figure 7.10. To make the current go up, the microcontroller outputs a binary 01 to the interface. To make the current go down, it outputs a binary 10. Since there are 2 coils, four outputs will be required (e.g., 0101_2 means up/up). To spin the motor, we output the sequence 0101_2, 0110_2, 1010_2, 1001_2… over and over. Each output causes the motor to rotate a fixed angle. To rotate the other direction, we reverse the sequence (0101_2, 1001_2, 1010_2, 0110_2…).

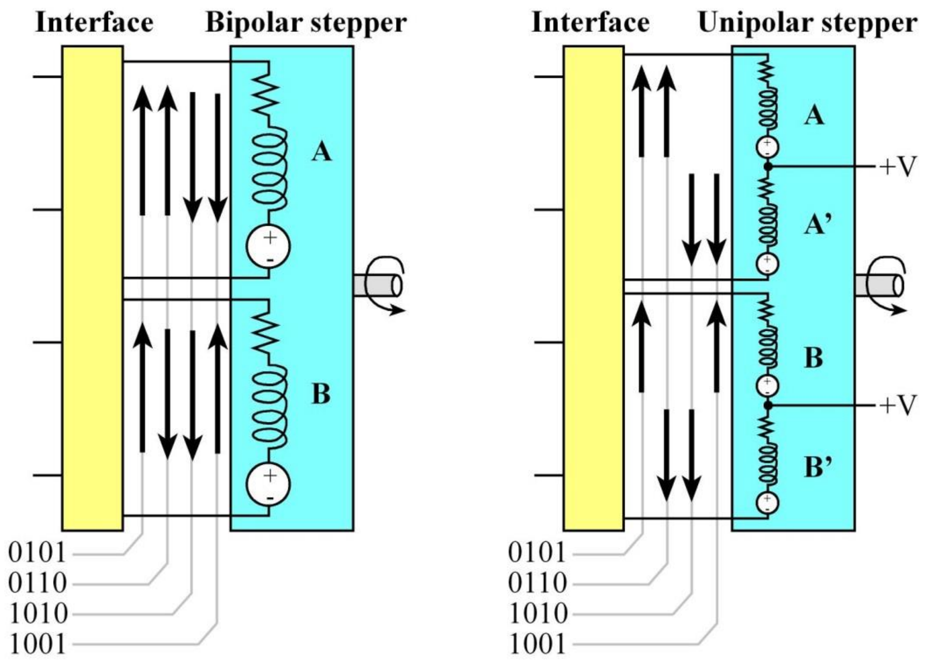


Figure 7.10. A bipolar stepper has 2 coils, but a unipolar stepper divides those two coils into four parts.

The unipolar stepper motor provides for bi-directional currents by using a center tap, dividing each coil into two parts. In particular, coil **A** is split into coil **A** and **A'**, and coil **B** is split into coil **B** and **B'**. The center tap is connected to the +V power source and the four ends of the coils can be controlled with open collector drivers. Because only half of the electro-magnets are energized at one time, a unipolar stepper has less torque than an equivalent-sized bipolar stepper. However, unipolar steppers are easier to interface. For more information on interfacing stepper motors see Volume 2, Embedded Systems: Real-Time Interfacing to ARM® Cortex™-M Microcontrollers.

There is a North and a South permanent magnet on the rotor (the part that spins). The amount of rotation caused by each current reversal is a fixed angle depending on the number of teeth on the permanent magnets. For example, the rotor in Figure 7.11 is drawn with one North tooth and one South tooth. If there are *n* teeth on the South magnet (also *n* teeth on the North magnet), then the stepper will move at *90/n* degrees. This means there will be *4n* steps per rotation. Because moving the motor involves accelerating a mass (rotational inertia) against a load friction, after we output a value, we must wait an amount of time before we can output again. If we output too fast, the motor does not have time to respond. The speed of the motor is related to the number of steps per rotation and the time in between outputs.

Click 'CW' or 'CCW' to step the motor clockwise and counterclockwise, respectively. Observe how different microcontroller outputs to the motor coils produce different responses from the motor. Assume that an output value of '1' to a motor coil will cause current to flow in that motor, resulting in a 'N' oriented magnetic field at that coil. Click here to see interactive http://users.ece.utexas.edu/~valvano/Volume1/E-Book/C10_Interactives.htm#ITool10.2

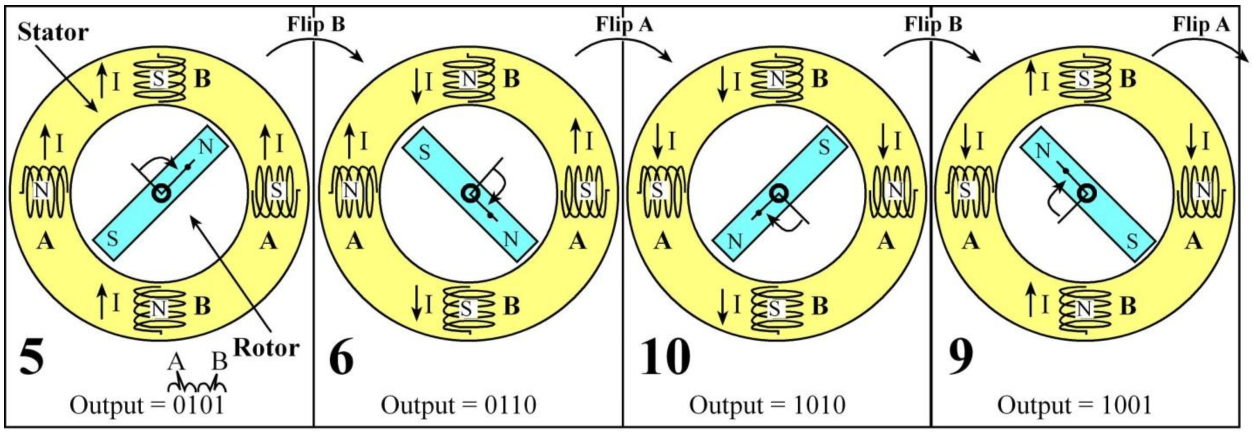


Figure 7.11. To make a stepper motor move by one step, the interface flips the direction of one of the currents.

Figure 7.12 shows a circular linked graph containing the output commands to control a stepper motor. This simple FSM has no inputs, four output bits and four states. There is one state for each output pattern in the usual stepper sequence 5,6,10,9... The circular FSM is used to spin the motor is a clockwise direction. This FSM has no inputs, but could be used to spin a stepper motor at constant speed.

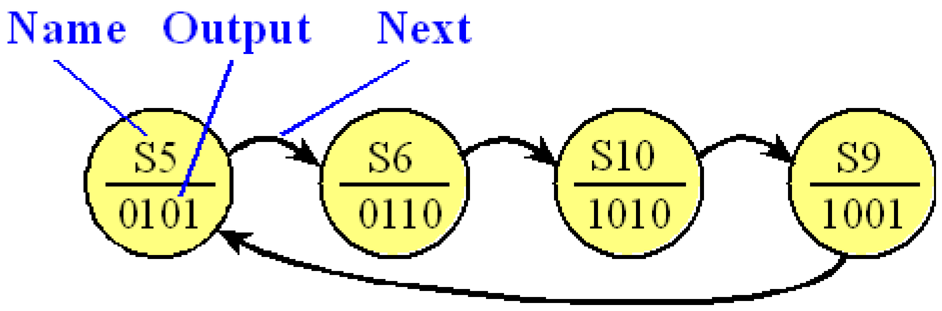


Figure 7.12. This stepper motor FSM has four states. The 4-bit outputs are given in binary.