EE 109 Unit 6

Software State Machines

What is state?

- It's late at night. You see a DPS officer approaching you. Are you happy?
  - It depends on what was happening just a minute ago.
  - Your car broke down.
  - You've been partying a little too hard.
- You press the PAUSE/PLAY button on a video player. What happens?
  - It depends on what was happening previously.
  - We also want to stay in that mode indefinitely after the button is released.
  - This requires maintaining STATE, which helps us remember the necessary information for the system to operate correctly

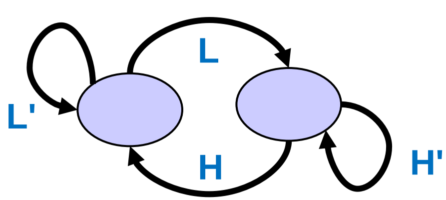
What is state?

- **State**: Everything that must be **remembered** to **interpret the inputs** (think the play/pause button) and/or to produce **outputs at appropriate times**
  - Usually, state is required for **time-dependent** behavior
- As a human:
  - Your "state" determines your interpretation of your senses and thoughts
  - The sum of all your previous experiences is what is known as state
- In a circuit:
  - State refers to all the bits being remembered in **registers** or **memory**
- In software:
  - State refers to all the **variable** values that are being used

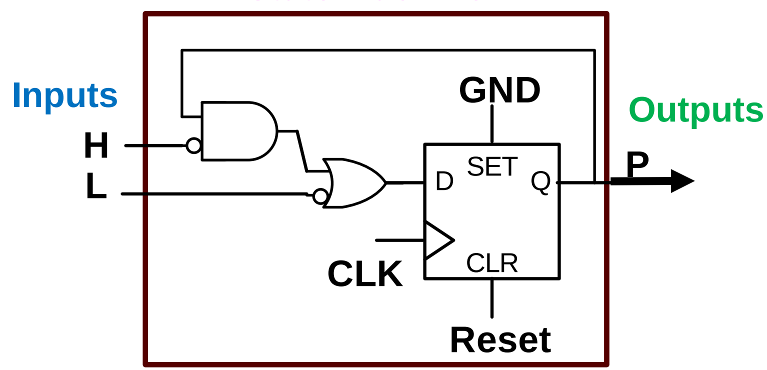
State Machines and State Diagrams

- Hardware and software components that utilize state are referred to as **state machines** (or FSMs = Finite State Machines)
- A state machine is modeled by a **state diagram** (i.e. a flow-chart)
- **FSMs are a very nice problem-solving approach/strategy**
  - If you can model your design with a state diagram, there are **straightforward transformations** to either software (what we'll study today) or hardware (later in the semester)

State Diagram (Abstract representation of FSM operation)

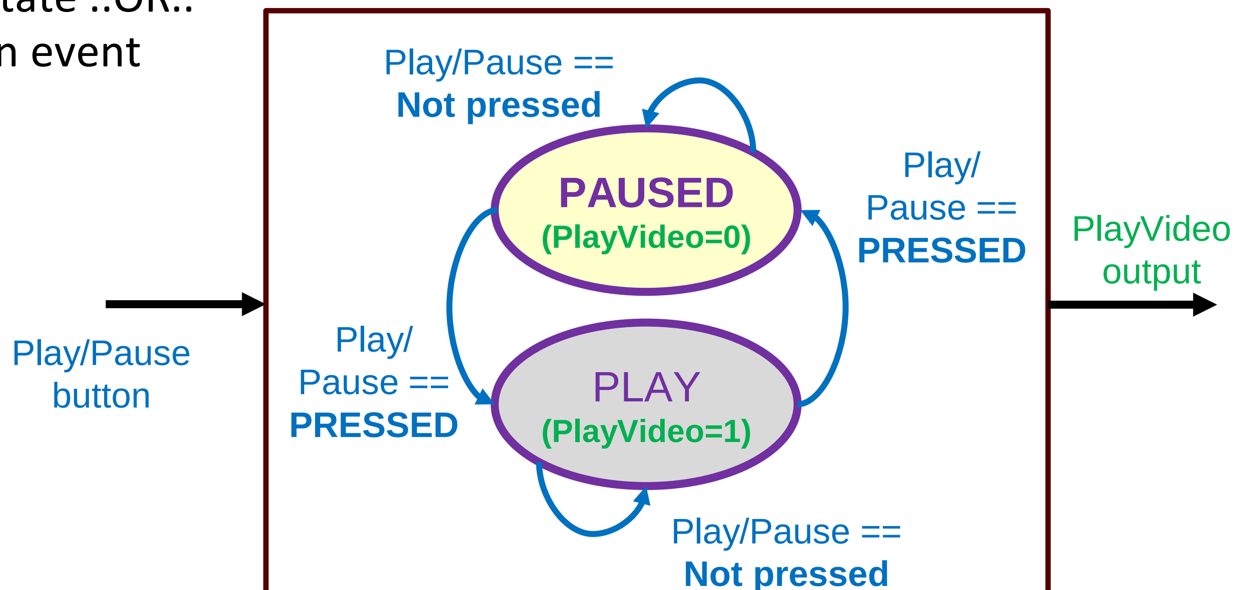


State Machine



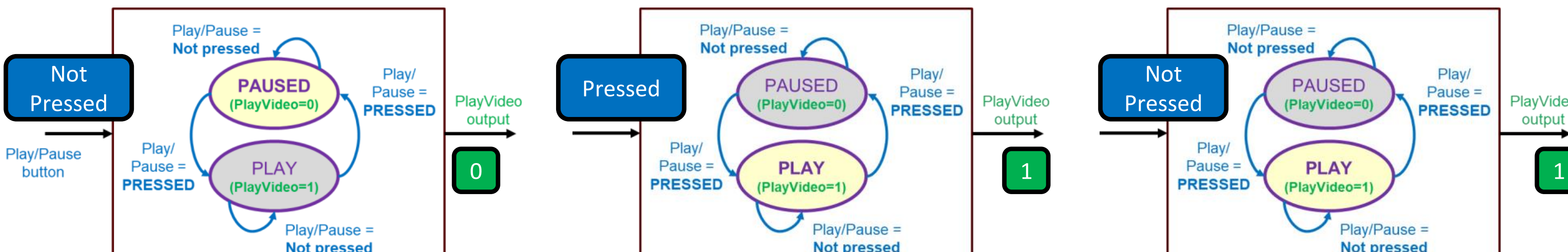
State Diagrams

- The **state diagram** should have 3 parts:
  - The **states** as circles or boxes
  - The **transitions** as arrows labeled by **input** conditions
  - The **outputs**, which can be generated when in a particular state ..OR.. on a specific transition event



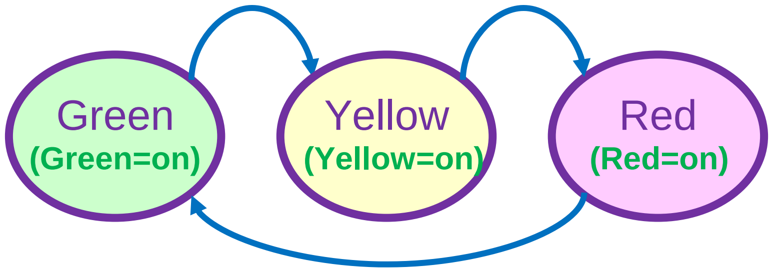
Operation

- State is used to generate the outputs even while the inputs are not activated
- When an input is activated, the state can be updated…
- …and remembered after the input has deactivated



Another Example: Traffic Light

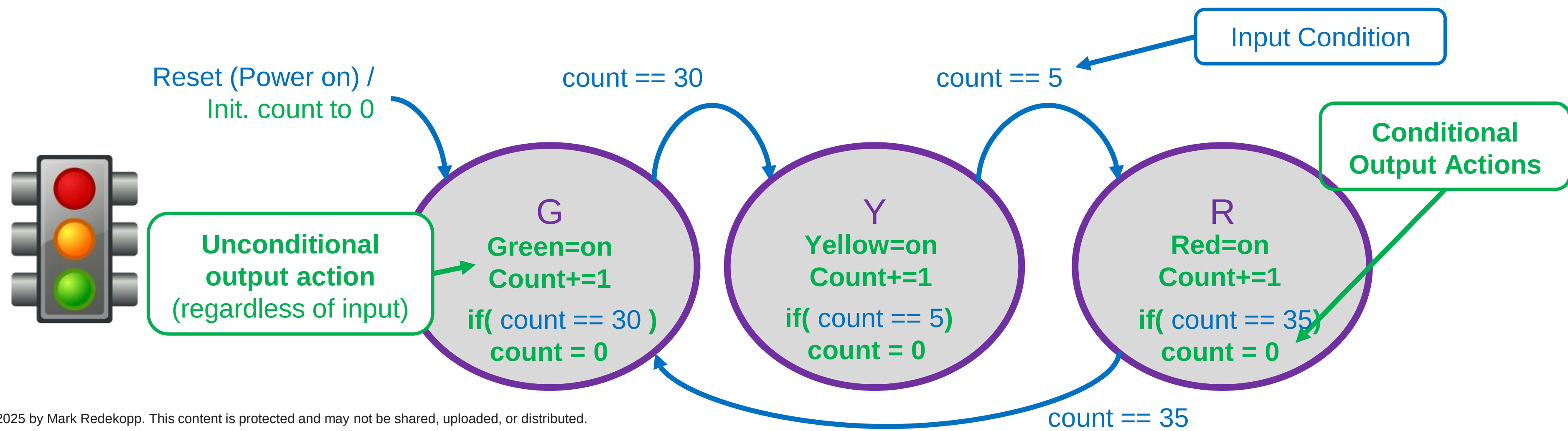
- State machines can be used to trigger **time-dependent** updates
  - Consider a system controlling the traffic lights at an intersection
  - There are no external **inputs** to indicate when the light should change
  - Instead, the **outputs** must change/transition based on time.
  - The **state** helps determine what the next **output** should be.



If a transition does not have a condition, it means it is unconditional. Sometimes we may just label it with **1 (true)**

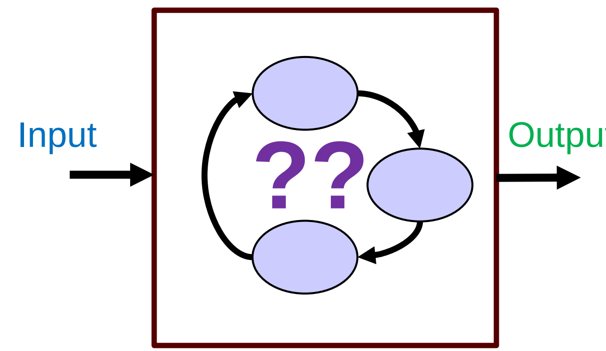
Time-Based Conditions

- Oftentimes we can use some kind of internal time counter to control when we transition states
  - Suppose our internal SW loop cycles every 1 second
- We can generate our **output/actions**
  - On each iteration, based on **state** (Green, Yellow, Red lights; increment counter)
  - On specific iterations based on other **conditions** (if count is 30, reset it to 0)



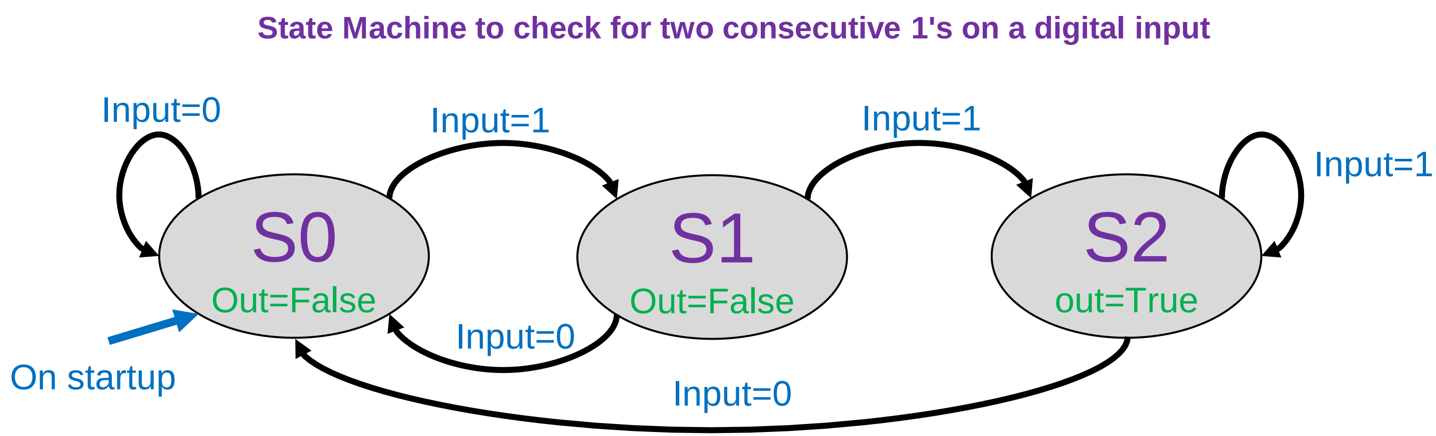
FSM Example 1-1

- Consider a system with one digital input and one output.
- The output should be true whenever the input is 1 for two consecutive time units
  - `Input:  0 1 0 1 1 0 1 1 1 0`
  - `Output: 0 0 0 0 0 1 0 0 1 1`
- Does this system need state?
- To help answer the question:
  - "The input is a 1 right now, should the output be true?"
  - Need to know whether the input was true last time unit as well? We need state!



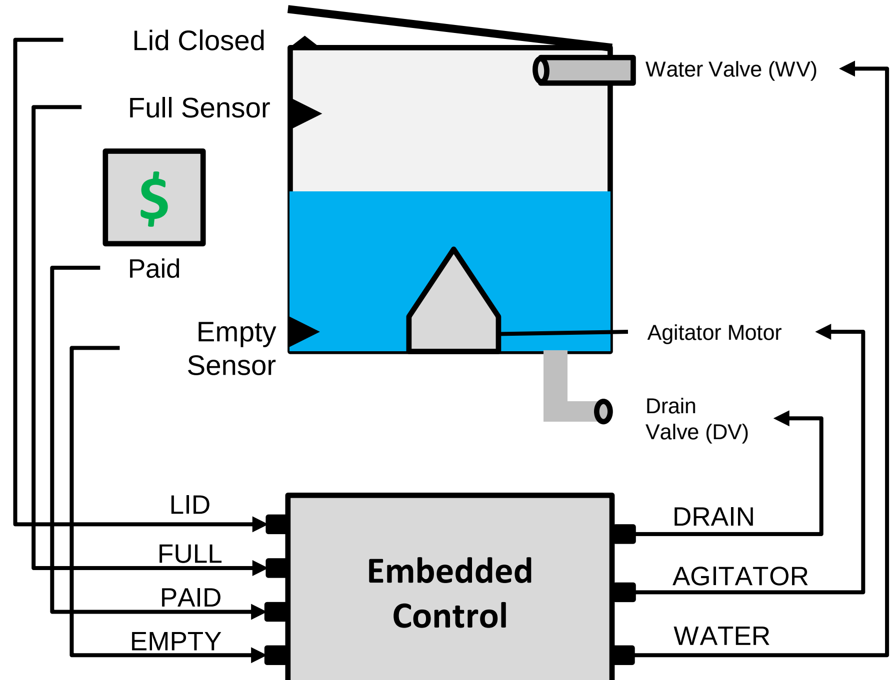
FSM Example 1-2

- Draw the state diagram for the system that outputs true (1) whenever the input has been 1 for two consecutive time periods



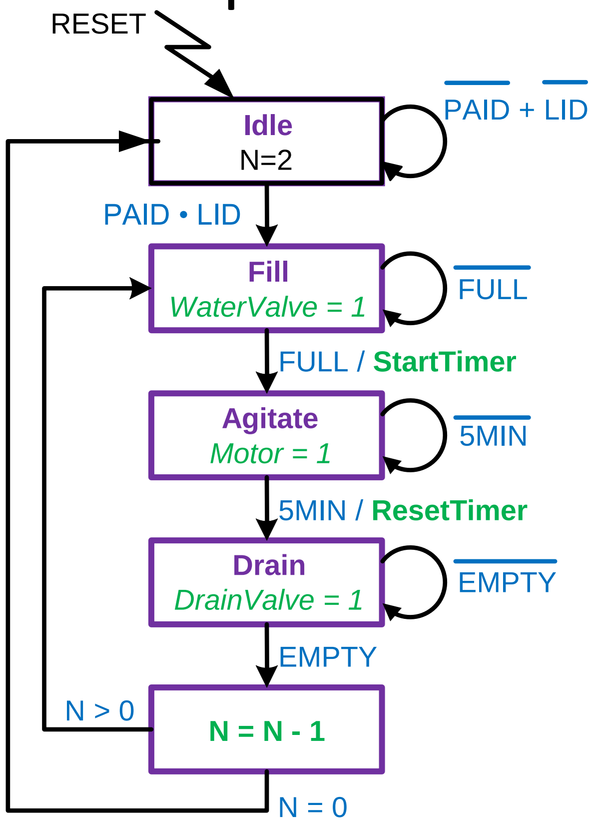
FSM Example 2: Washing Machine

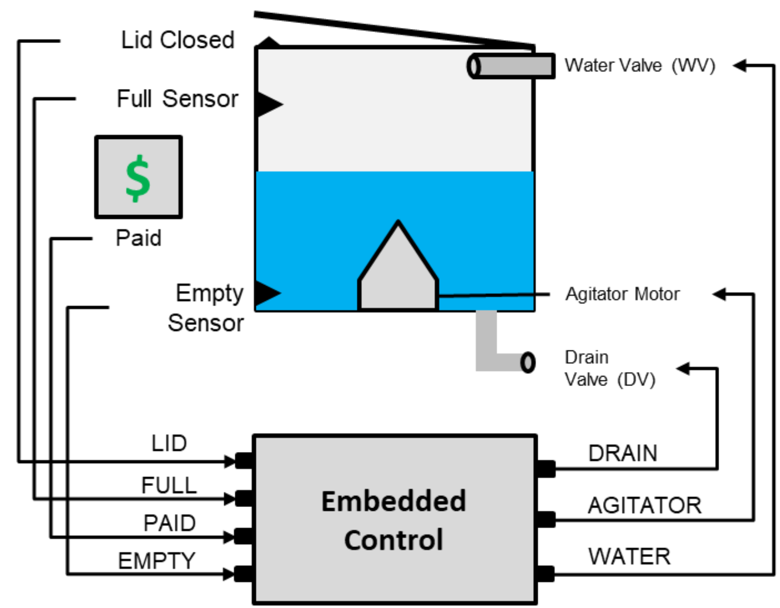
- Consider the design of an embedded controller for a coin/card-operated laundry machine.
- Consider the inputs and outputs



Washing Machine State Diagram

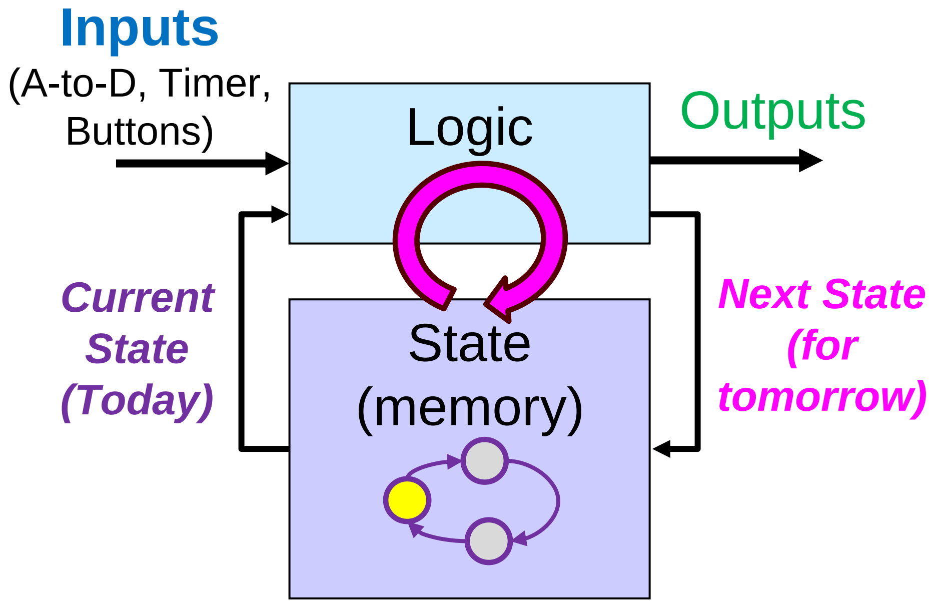
- Examine a potential state machine for this design.





A Day in the Life of a State Machine

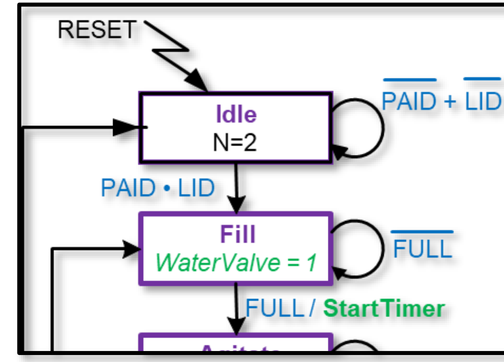
- State machines operate time step by time step
  - **Human analogy: day-by-day (see inset)**
- Each time step, the state machine use current (today's) state to:
  - Determine which inputs to examine to determine the **next (tomorrow's) state**
  - Determine any **outputs** and **actions** to take (sometimes based on the inputs)



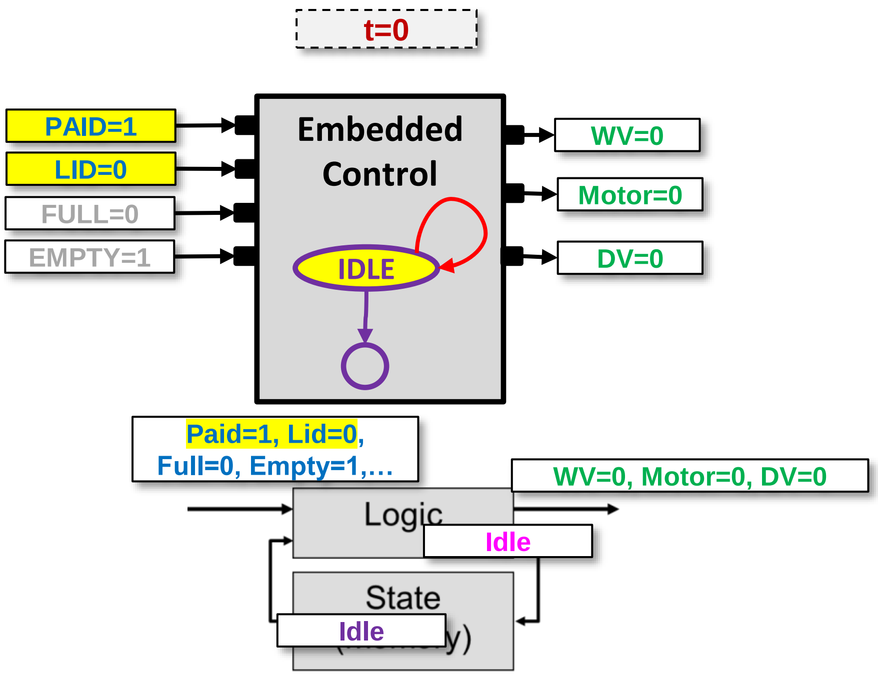
Human analogy: day-by-day

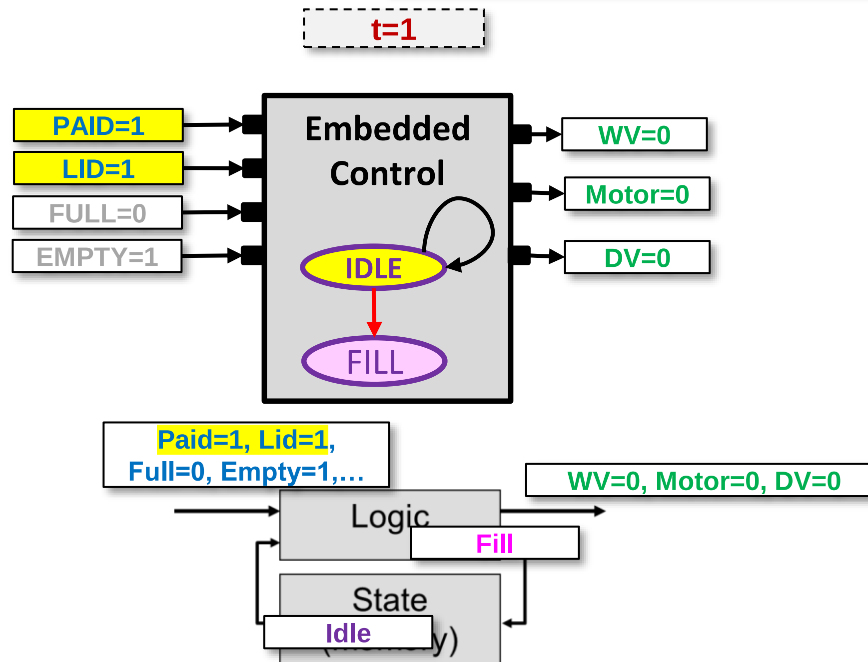
- Wake up with only a **memory** of the **current state**
- Use **current state** (and inputs) to determine **outputs** and **actions** for today
- Use **current state** and **inputs** to update state (i.e. determine state for tomorrow)
- Go to sleep and repeat same process tomorrow

State Machine Operation (1)

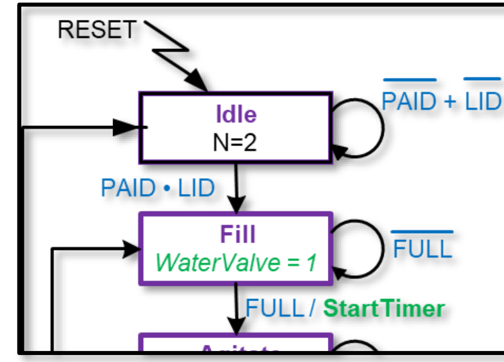


- Notice how the current state helps identify which inputs "matter" at specific times

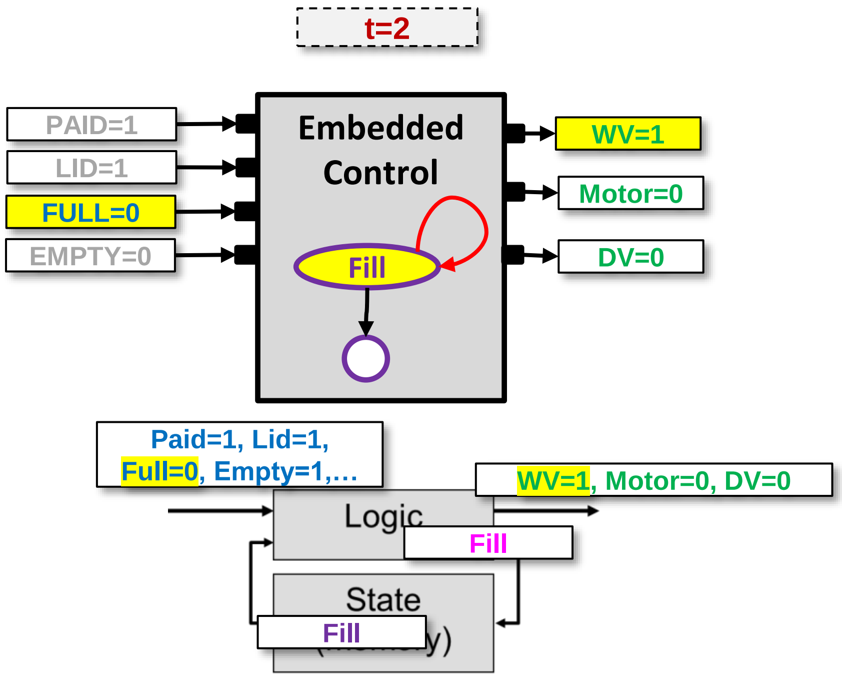


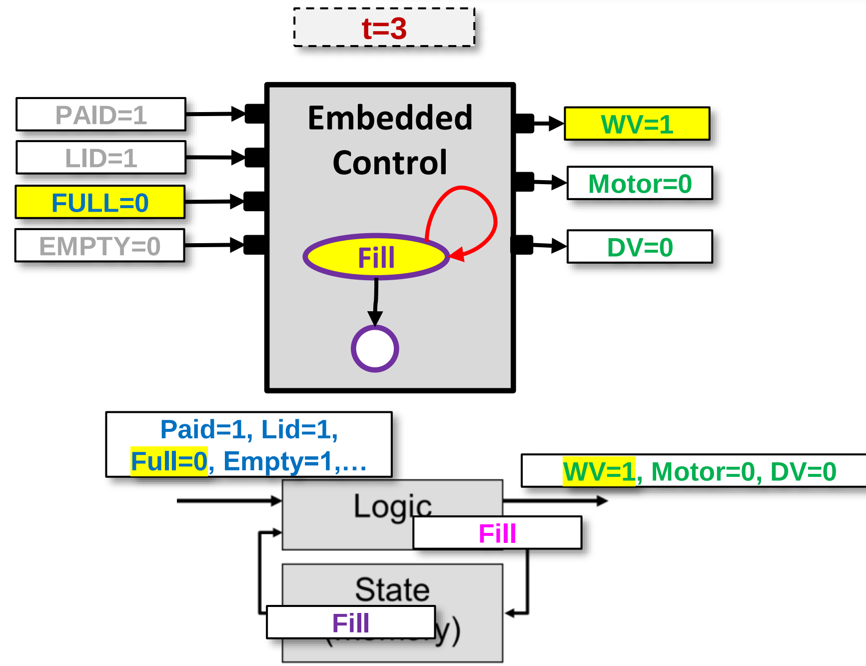


State Machine Operation (2)



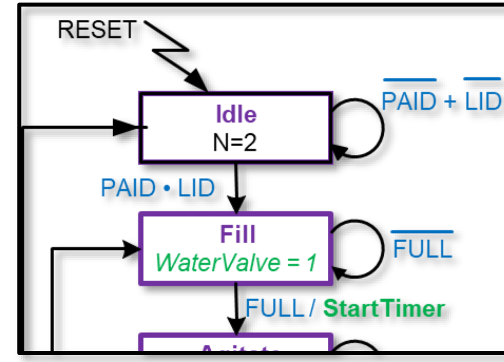
- When the state changes, we produce new output values and may look at a new set of inputs

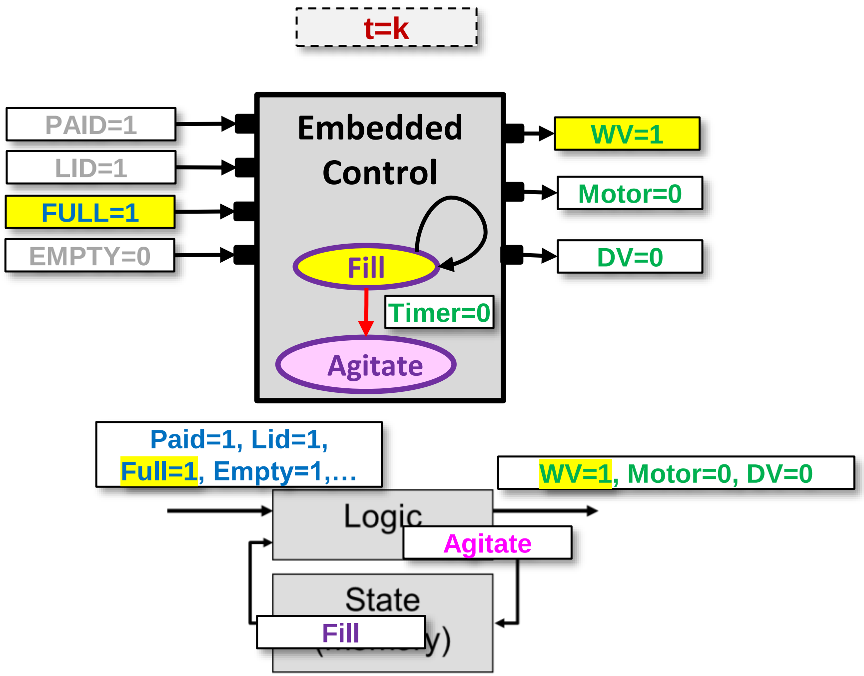


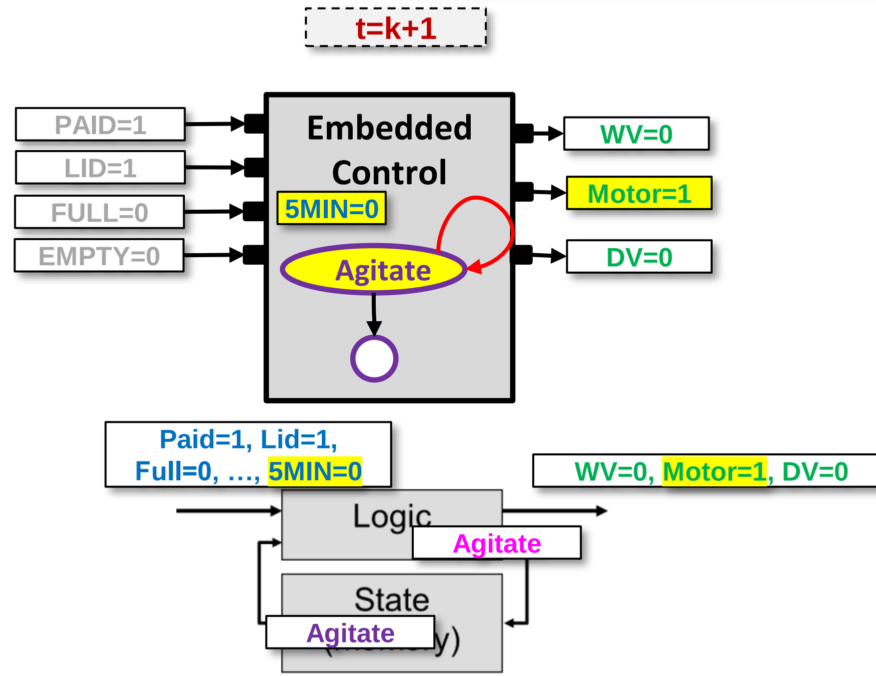


State Machine Operation (3)

- We can use internal "time" inputs to control when we change states.



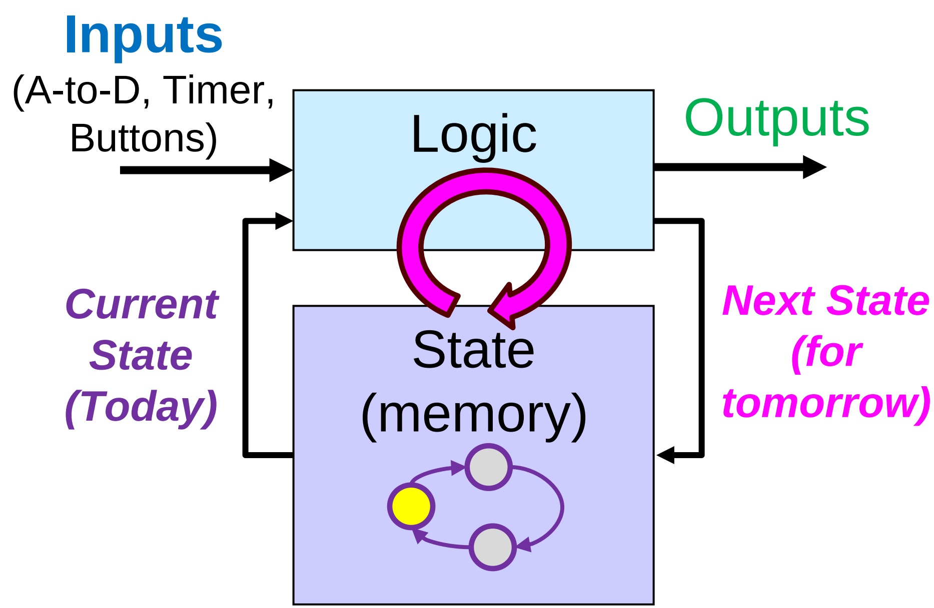




IMPLEMENTING STATE MACHINES IN SOFTWARE

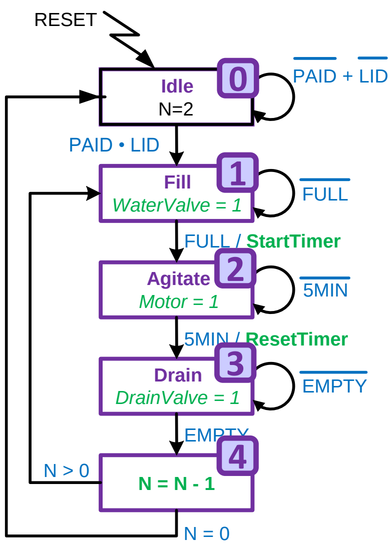
Software and Hardware Implementation

- Software Implementation
  - **Current State = just a variable(s)**
  - **Input**/**output** Logic = **if** statements to update the next state or produce outputs
    - ```
      if(state == 0 && input == 1)
          { state = 1; output = 0; }
      ```
  - Transitions triggered by input or timers
  - **We'll start by implementing state machines in SW**
- Later in the semester we'll see how to implement state machines in hardware


      ```

Coding State Machines 1

- Setup (declare and initialize) your state variable
  - Choose some numeric code for each state: 0=Idle, 1=Fill, 2 = Agitate, etc.
- Use **one while** loop and a single delay (or timer) to repeat the "day-in-the-life" routine of a state machine




```
int main(){
  char currst = 0, n = 2; int timer;
  // other initialization
  while(1) {















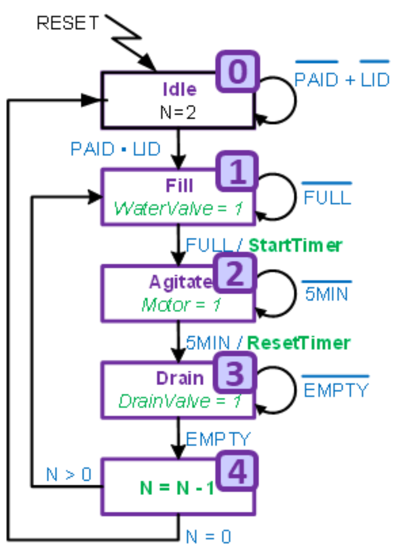





     _delay_ms(100);
  }
  return 0;
}
```

Coding State Machines 2

- In the while loop, setup a series of if..else if..else statements to determine what state you are in "today"

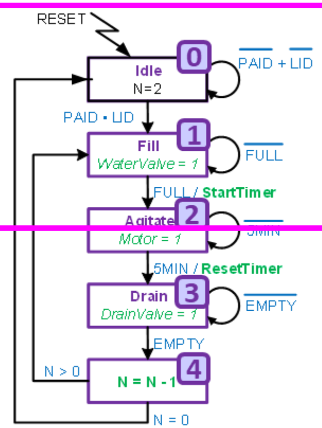



```
int main(){
  char currst = 0, n = 2; int timer;
  // other initialization
  while(1) {

    if( currst == 0 ){ // Idle
      // code pertinent to Idle
    }
    else if( currst == 1 ){ // Fill
      // code pertinent to Fill
    }
    else if( currst == 2 ){ // Agitate
      // code pertinent to Agitate
    }
    else if( currst == 3 ){ // Drain
      // code pertinent to Drain
    }
    else { // Decrement
      // code pertinent to last state
    }
    _delay_ms(100);
  }
  return 0;
}
```

Coding State Machines 3a

- Sample the inputs at the start of each iteration (each day)
- In each if statement for the current state, use a **nested if statement** for the **input conditions** to determine **next state** and **output** actions

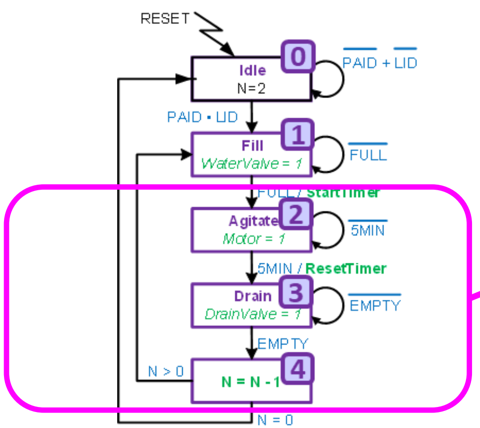



```
int main(){
  char currst = 0, n = 2; int timer;
  // other initialization
  while(1) {
    char paid = PIND & (1 << PD0);
    char lid = PIND & (1 << PD1);
    char full = PIND & (1 << PD2);
    if( currst == 0 ){ // Idle
      if(paid && lid)
       { currst = 1; /* Goto Fill */ }
    }
    else if( currst == 1 ){ // Fill
      PORTC |= (1 << PC0); // WV=1
      if(full)
        { currst = 2; timer = 0; }
    }
    else if( currst == 2 ){ // Agitate
      PORTC &= ~(1 << PC0); // WV=0
      PORTC |= (1 << PC1);  // Motor=1
      ...
    }
    ...
    _delay_ms(100);
  }
  return 0;
}
```

Notice the nested IF statement structure used for state machines.

Coding State Machines 3b

- Sample the inputs at the start of each iteration (each day)
- In each if statement for the current state, use a **nested if statement** for the **input conditions** to determine **next state** and **output** actions




```
int main(){
  char currst = 0, n = 2; int timer;
  while(1) {
    ...
    char full = PIND & (1 << PD2);
    char empty = PIND & (1 << PD3);
    ...
    else if( currst == 2 ){ // Agitate
      timer++;
      PORTC &= ~(1 << PC0); // WV=0
      PORTC |= (1 << 1); // Motor=1
      if(timer=3000) // 5 min
        { currst = 3; timer = 0; }
    }
    else if( currst == 3 ){ // Drain
      PORTC &= ~(1 << PC1); // Motor=0
      PORTC |= (1 << PC2); // DV = 1
      if(empty) { currst = 4; }
    }
    else { if(--n > 0) currst = 1;
           else        currst = 0;  }
    _delay_ms(100);
  }
  return 0;
}
```

The final "else" is equivalent to "else if (currst == 4)".

Notice the nested IF statement structure used for state machines.

Enumerations

- It would be nice to use **symbolic names** for states, rather than numbers
- In C/C++, enumerations associate an integer code (number) with a symbolic name
- Syntax:
  `enum [optional_collection_name] {SymName1, SymName2, … SymNameN}`
  - SymName1 = 0
  - SymName2 = 1
  - …
  - SymNameN = N-1
- Use symbolic item names in your code and compiler will replace the symbolic names with corresponding integer values…**makes the code much more readable!**

```
const int IDLE=0;
const int FILL=1;
const int AGITATE=2;
...
char state = IDLE;
...
if(state == FILL && full == true) {
  state = AGITATE;
}
```

Option 1: Hard coding symbolic state names with given codes. Better than nothing, but enumerations (below) are often preferred.

```
// First enum item is associated with code 0
enum States {IDLE, FILL, AGITATE, DRAIN, DEC};
// auto-assign 0     1      2        3    4
char state = IDLE;    // same as state = 0;
...
if(state == FILL && full == true) {
   state = AGITATE;   // same as state = 2;
}
```

Option 2: Using enumeration to simplify state coding and make the code more readable!

Another Example: 2 Consecutive 1's FSM

- How would we begin to code the implementation of this state machine?
  - Start with an enum to list the states
  - Declare and initialize your state variable
  - Choose or determine the rate / delay at which transitions in state should be made or output actions must occur.
    - 1 iteration of the loop handles 1 time step (a "day")

```
enum { S0, S1, S2 };
// input = PD0,  output = PD7
int main()
{ // be sure to init. state
  unsigned char state=S0, input;
  while(1)
  {





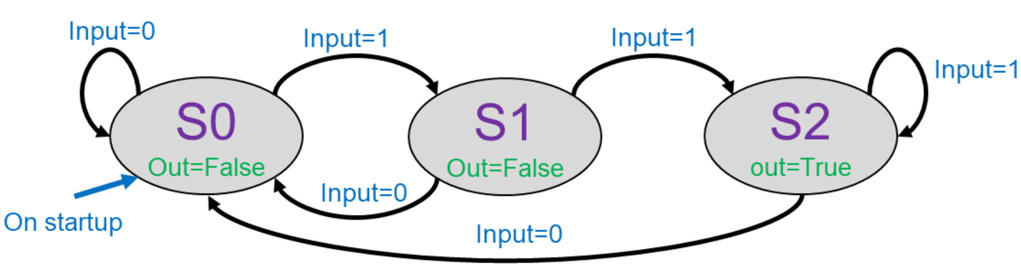







    _delay_ms(10); // use approp. Time
  } return 0;
}
```

State Machine to check for two consecutive 1's on a digital input



Consecutive 1's FSM – State

- Again, notice the structure:
  - The purple **'if'** statements determine which **state** we are in

```
enum { S0, S1, S2 };
// input = PD0,  output = PD7
int main()
{ // be sure to init. state
  unsigned char state=S0, input;
  while(1)
  {

    if(state == S0){

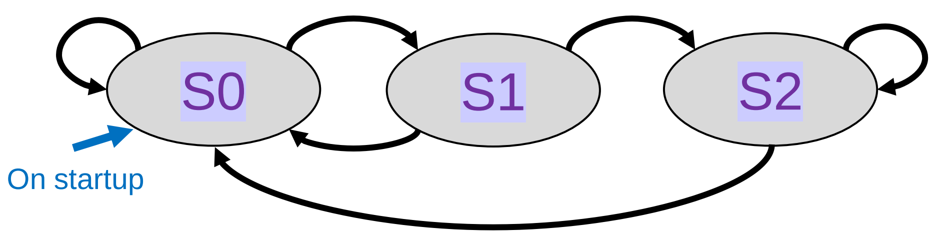

    }
    else if(state == S1){


    }
    else { // state == S2


    }
    _delay_ms(10); // use approp. Time
  } return 0;
}
```

Select current state

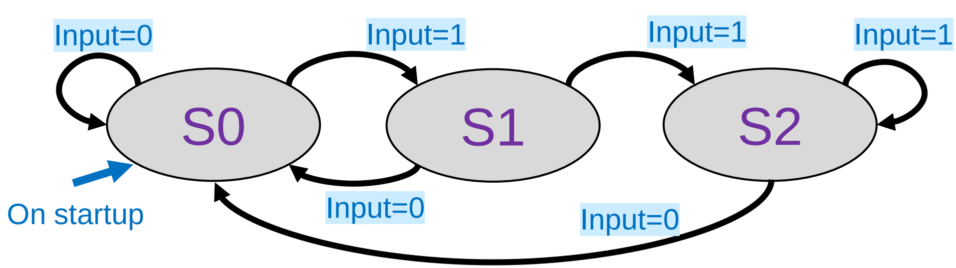
State Machine to check for two consecutive 1's on a digital input



Consecutive 1's FSM – Transitions

- Again, notice the structure:
  - The nested orange 'if' statements determine which **input conditions** are true to determine how we **update the state**

State Machine to check for two consecutive 1's on a digital input



Select current state

Select input val.

```
enum { S0, S1, S2 };
// input = PD0,  output = PD7
int main()
{ // be sure to init. state
  unsigned char state=S0, input;
  while(1)
  {
    input = PIND & (1 << PD0);
    if(state == S0){

      if( input ){ state = S1; }
    }
    else if(state == S1){

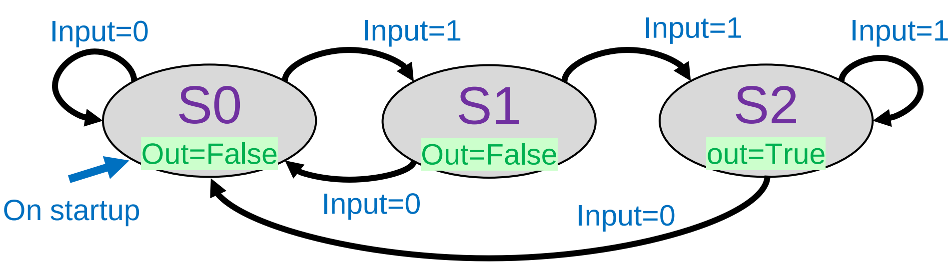
      if( input ){ state = S2; }
      else { state = S0; }
    }
    else { // state == S2

      if( !input ) { state = S0; }
    }
    _delay_ms(10); // use approp. Time
  } return 0;
}
```

Consecutive 1's FSM – Output Actions

- Again, notice the structure:
  - We can add appropriate **output** actions

State Machine to check for two consecutive 1's on a digital input




```
enum { S0, S1, S2 };
// input = PD0,  output = PD7
int main()
{ // be sure to init. state
  unsigned char state=S0, input;
  while(1)
  {
    input = PIND & (1 << PD0);
    if(state == S0){
      PORTD &= ~(1 << PD7);
      if( input ){ state = S1; }
    }
    else if(state == S1){
      PORTD &= ~(1 << PD7);
      if( input ){ state = S2; }
      else { state = S0; }
    }
    else { // state == S2
      PORTD |= (1 << PD7);
      if( !input ) { state = S0; }
    }
    _delay_ms(10); // use approp. Time
  } return 0;
}
```

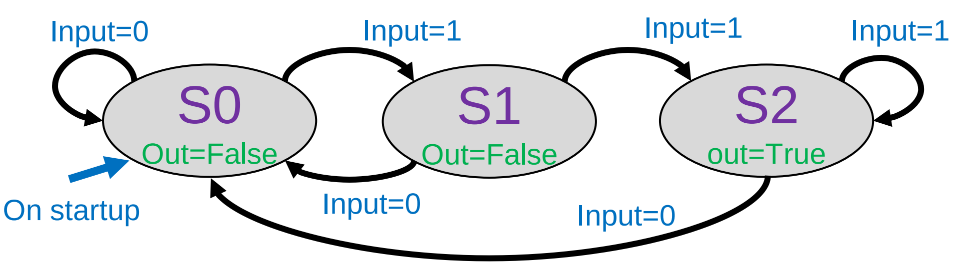
Select current state

Select input val.

Consecutive 1's FSM – Summary

- Again, notice the structure:
  - 1 iteration of the loop handles 1 time step (a "day")
  - The purple 'if' statements determine which **state** we are in and the nested orange 'if' statements determine which **input conditions** are true to determine how we **update the state** and what **output actions** we take
  - Some delay before the next iteration begins

State Machine to check for two consecutive 1's on a digital input



Select current state

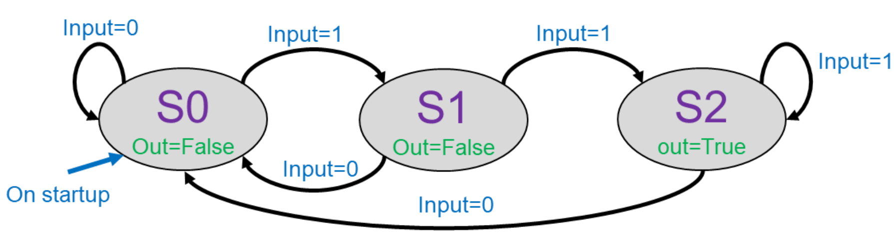
Select input val.

```
enum { S0, S1, S2 };
// input = PD0,  output = PD7
int main()
{ // be sure to init. state
  unsigned char state=S0, input;
  while(1)
  {
    input = PIND & (1 << PD0);
    if(state == S0){
      PORTD &= ~(1 << PD7);
      if( input ){ state = S1; }
    }
    else if(state == S1){
      PORTD &= ~(1 << PD7);
      if( input ){ state = S2; }
      else { state = S0; }
    }
    else { // state == S2
      PORTD |= (1 << PD7);
      if( !input ) { state = S0; }
    }
    _delay_ms(10); // use approp. Time
  } return 0;
}
```

A Potential Alternate Structure

- Sometimes, it may be easiest to **separate** :
  - the **state transition code** and
  - the **output action** code
- We can use separate **'if'** sequences.

State Machine to check for two consecutive 1's on a digital input



State transitions

Outputs

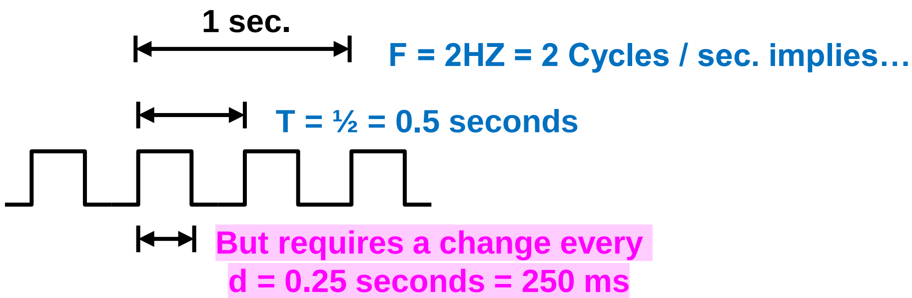
```
enum { S0, S1, S2 };
int main() {
  unsigned char state=S0, input;
  while(1) {
    // state transitions
    input = PIND & (1 << PD0);
    if(state == S0){
      if( input ){ state = S1; }
    }
    else if(state == S1){
      if( input ){ state = S2; }
      else { state = S0; }
    }
    else { // state == S2
      if( !input ) { state = S0; }
    }
    // output actions
    if( state == S2)
      PORTD |= (1 << PD7);
    else
      PORTD &= ~(1 << PD7);
    _delay_ms(10); // use approp. Time
  } return 0;
}
```

State Machines as a Problem-Solving Technique

- Modeling a problem as a state machine is a powerful problem-solving tool
- When you need to write a program, design HW, or solve a more abstract problem at least consider if it can be modeled with a state machine
  - Ask questions like:
    - What do I need to remember to interpret my inputs or produce my outputs? [e.g. Checking for two consecutive 1's]
    - Is there a distinct sequence of "steps" or "modes" that are used (each step/mode is a state) [e.g. Washing machine, etc.]

A Note About Timing

- Write a program to blink an LED at **2HZ**
- What delays should you use?



- If all we are doing is blinking, we can simplify to use an XOR to flip the output bit

```
int main()
{
  // Initialization
  while(1)
  {
     PORTD |= (1 << 7);  // LED on PD7
     _delay_ms(250);
     PORTD &= ~(1 << 7);
     _delay_ms(250);
  }
  return 0;
}
```

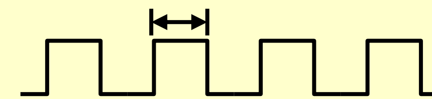
```
int main()
{
  // Initialization
  while(1)
  {
     PORTD ^= (1 << 7); // LED on PD7
     _delay_ms(250);
  }
  return 0;
}
```

Tunnel Vision (1)

- Consider a program that constantly monitors several inputs and takes appropriate actions:
  - If button1 is pressed it should blink an LED 10 times at a rate of 2 HZ
  - If button2 is pressed it should output something to the LCD screen
  - If button3 is pressed it should enable a motor
  - And even more tasks…
- To do something 10 times, it would be easiest to use a `for` loop, RIGHT?!?

```
// Ad-hoc implementation
int main()
{
  while(1)
  {
     int i;
     if(checkInput(1) == 0) {
       for(i=0; i < 10; i++) {
          blink(250); // on for 250, off for 250
          // delays are in the blink() functions
       }
     }
     if(checkInput(2) == 0) {
       // output to LCD
     }
     if(checkInput(3) == 0) {
       // enable motor
     }
     if(...) {
       // even more tasks
     }
  }
  return 0;
}
```

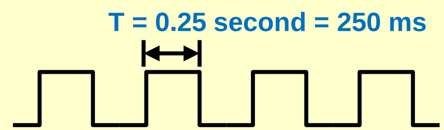
T = 0.25 second = 250 ms



Tunnel Vision (2)

- Consider a program that constantly monitors several inputs and takes appropriate actions:
  - If button1 is pressed it should blink an LED 10 times at a rate of 2 HZ
  - If button2 is pressed it should output something to the LCD screen
  - If button3 is pressed it should enable a motor
  - And even more tasks…
- To do something 10 times, it would be easiest to use a `for` loop, RIGHT?!?
- No! When we are in the `for` loop, we would not be performing our other tasks and miss actions.

```
// Ad-hoc implementation
int main()
{
  while(1)
  {
     int i;
     if(checkInput(1) == 0) {
       for(i=0; i < 10; i++) {
          blink(250); // on for 250, off for 250
          // what if button 2, 3, ... are pressed?
       }
     }
     if(checkInput(2) == 0) {
       // output to LCD
     }
     if(checkInput(3) == 0) {
       // enable motor
     }
     if(...) {
       // even more tasks
     }
  }
  return 0;
}
```



A Better Approach

To keep many things going at once, cycle through all the tasks doing only a short / small amount of the task at a time!

A Better Approach

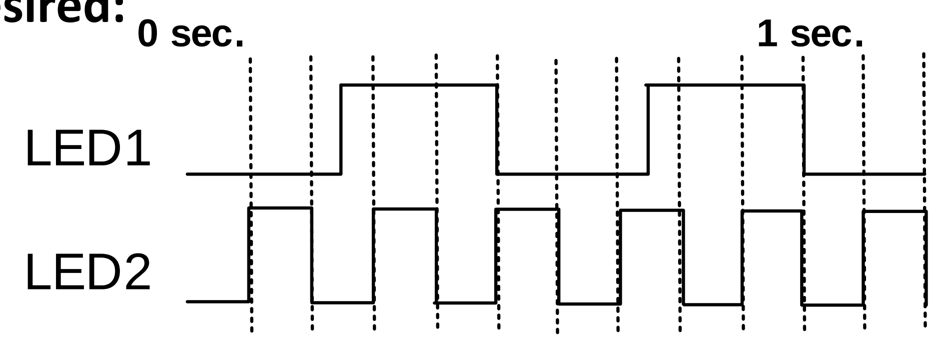
- Instead, perform 1 blink **per** iteration, tracking your count!
- This allows other checks and actions to be performed after each single blink
- You can use your count as a "state" variable:
  - cnt: 0-9 tracks how many blinks
  - cnt: 10 DONE/OFF
- ...or use a separate state variable (**s=1**: counting, **s=0**: DONE/OFF) in combination with **cnt**
- Every time we press button 1, we reset the cnt to start 10 more blinks

Other checks and actions

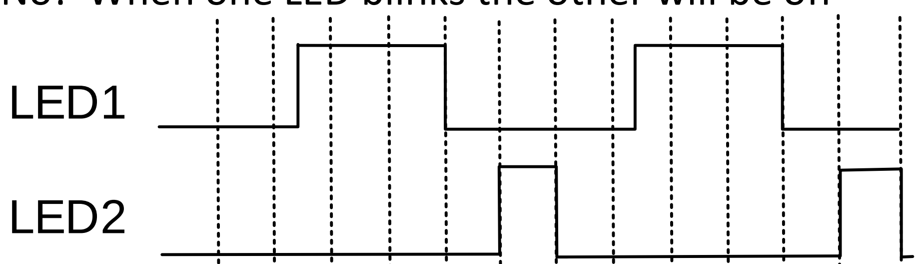
```
// Ad-hoc implementation
int main()
{
  int cnt=10;
  while(1)
  {
     if(checkInput(1) == 0) {
       cnt=0;
     }
     if(cnt < 10) {
       blink(250); // 1 blink per iter.
       cnt++;
     }
     if(checkInput(2) == 0) {
       // output to LCD
     }
     if(checkInput(3) == 0) {
       // enable motor
     }
     if(...) {
       // more tasks
     }
  }
  return 0;
}
```

Operations at Different Rates (1)

- Consider a program to blink one LED at a rate of 2 Hz and another at 5 Hz at the same time
- **Desired:**



- **Problem**: Does the code to the right work correctly?
  - No! When one LED blinks the other will be off




```
int main()
{
    while(1)
    {
        LED1_OFF();
        _delay_ms(250);
        LED1_ON();
        _delay_ms(250);

        LED2_OFF();
        _delay_ms(100);
        LED2_ON();
        _delay_ms(100);

    }

    return 0;
}
```

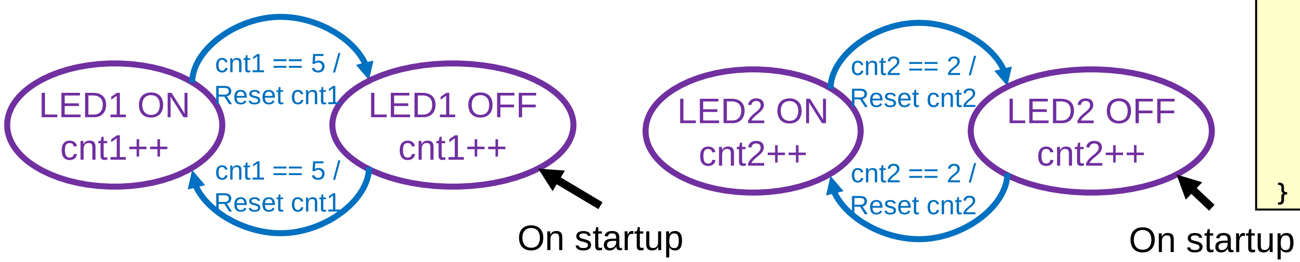
Operations at Different Rates (2)

- Use a SINGLE delay and separate state (**count**) variables to do work on each task at the "same time". This mimics "parallel" (aka multithreaded) execution.
- To determine that delay, find the GCD (Greatest Common Divisor) of the minimum periods that action is needed for each task.
  - **Task 1**: Flip the LED every **250 ms**
  - **Task 2**: Flip the LED every **100 ms**
  - Use a delay of **50ms** = GCD (250, 100)

```
int main()
{
    int cnt1 = 0, cnt2 = 0;

    // set initial state of LEDs as "off"
    LED1_OFF();
    LED2_OFF();

    while(1)
    {
        if(cnt1 == 5)
        {
            FLIP_LED1();
            cnt1 = 0;
        }
        cnt1++;
        if(cnt2 == 2)
        {
            FLIP_LED2();
            cnt2 = 0;
        }
        cnt2++;
        // Delay the minimum granularity
        _delay_ms(50);
    }
    return 0;
}
```



Operations at Different Rates (3)

- To determine that delay, find the GCD (Greatest Common Divisor) of the minimum periods that action is needed for each task.
  - **Task 1**: Flip the LED every **250 ms;**
    Task 2: Flip the LED every **100 ms**
  - Use a delay of **50ms** = GCD (250, 100)
- We can use a single counter looking for multiples of the individual task periods (every 2 or every 5 iterations) using the modulo operator
- Can reset the count to 0 after the **Least Common Multiple** of the task periods
  - **LCM(250,100) = 500ms = 10 iterations.**

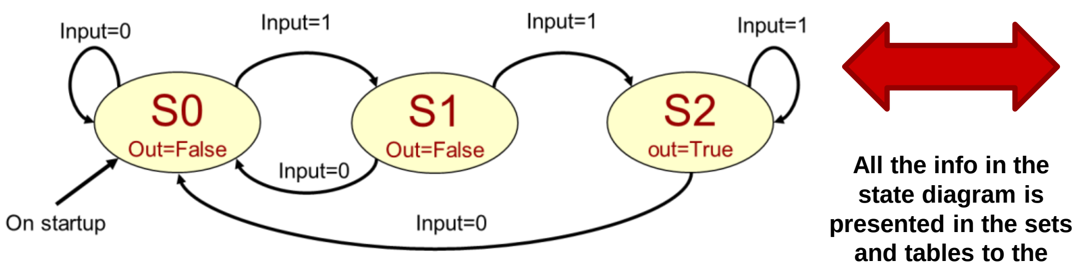
```
int main()
{
  int cnt = 0;

  // set initial state of LEDs as "on"
  LED1_ON();
  LED2_ON();

  while(1) {
    if(cnt % 5 == 0) {
      FLIP_LED1();
    }
    if(cnt % 2 == 0) {
      FLIP_LED2();
    }
    cnt++;
    if(cnt == 10)
      { cnt = 0; }
    // Delay the minimum granularity
    _delay_ms(50);
  }
  return 0;
}
```

Summary Definition

- To specify a state machine, we must specify 6 things:
  - A set of possible input values: {0, 1}
  - A set of possible states: {S0, S1, S2}
  - A set of possible outputs: {False, True}
  - An initial state = S0
  - A transition function:
    - {States x Inputs} -> the Next state
  - An output function:
    - {States x Inputs} -> Output value(s)



All the info in the state diagram is presented in the sets and tables to the right

Inputs: {0, 1}

States: {S0, S1, S2}

Outputs: {False, True}

Initial State: S0

	Inputs	
State	**0**	**1**
S0	S0	S1
S1	S0	S2
S2	S0	S2

State Transition Function

State	Outputs
S0	False
S1	False
S2	True

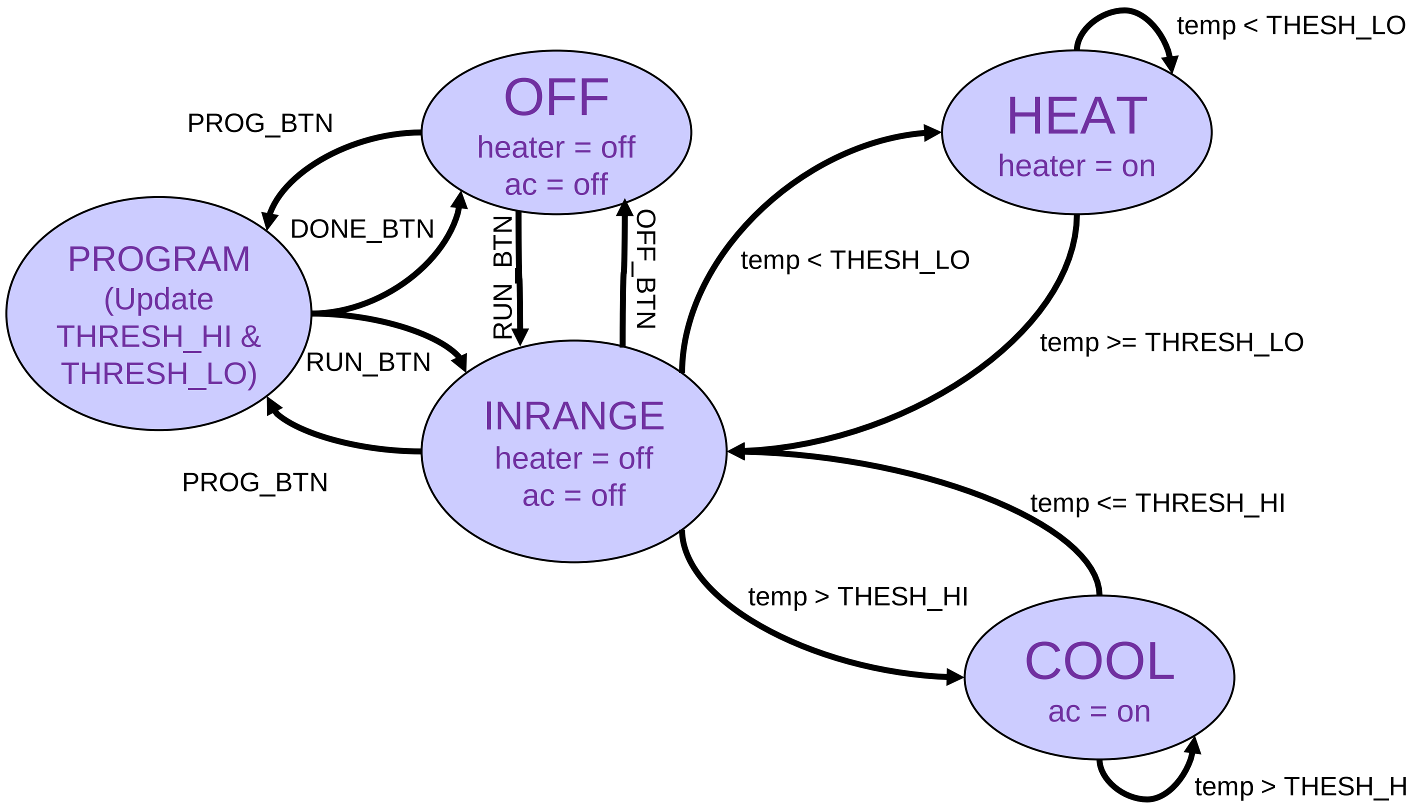
Output Function

HW (Instruction Cycle) & Software (String Matching)

MORE EXAMPLES IF TIME

Thermostat

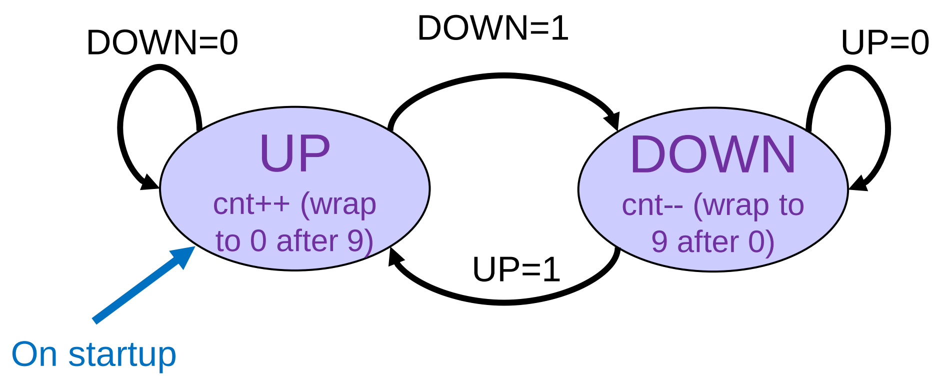
- Sample state machine to control a thermostat

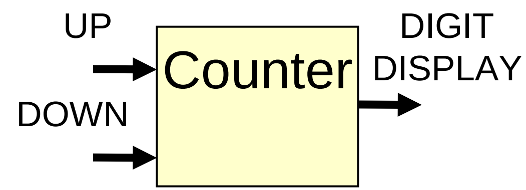


Counter Example

- Consider a system that has two button inputs: UP and DOWN and a 1-decimal digit display. It should count up or down at a rate of 500 milliseconds and change directions only when the appropriate direction button is pressed
- Every time interval we need to poll the inputs to check for a direction change, update the state and then based on the current state, increment or decrement the count

State Machine to count up or down (and continue counting) based on 2 pushbutton inputs: UP and DOWN





More State Machines

- State machines are all over the place in digital systems
- Instruction Cycle of a computer processor