

TI-RSLK

Texas Instruments Robotics System Learning Kit



TEXAS INSTRUMENTS



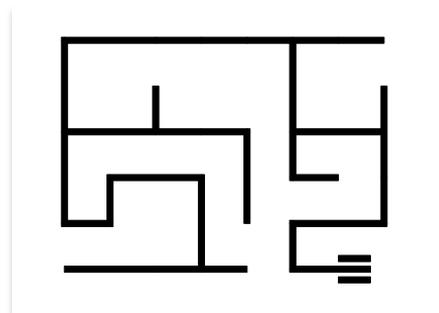
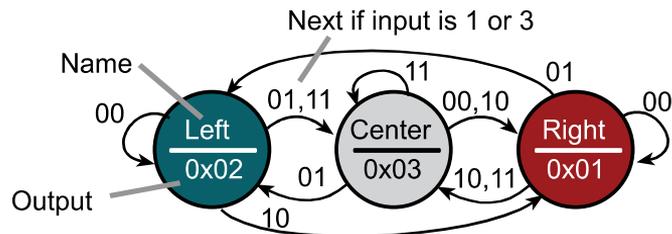
Module 7

Lecture: Finite State Machines -Theory

Finite State Machines - Theory

You will learn in this module

- C programming fundamentals
 - Arrays
 - Pointers
 - Structures
 - Time delays
- Develop debugging techniques such as
 - Watch windows
 - Breakpoints
 - Heart beats
- Solve problems with finite state machines
 - States, tables, graphs, input, outputs
 - Mealy versus Moore
- Design controller for a line tracking robot
 - Traffic light controller
 - Line-following robot





Abstraction

Software abstraction:

- Define a problem
 - Minimal set of basic concepts
 - Abstract principles / processes
- Separation of policy and mechanisms
 - Interfaces define what it does (policy)
 - Implementations define how it works (mechanisms)
- Straightforward, mechanical path to implementation

Three advantages of abstraction are:

- Faster to develop
- Easier to debug (prove correct) and
- Easier to change

Finite State Machine Rules

1. Simple structure: Input->Process->Output
2. Information is encoded by being in a state.
3. FSM controllers are very simple:
e.g., output, wait, input, go to next state.
4. Complexity is captured in the state graph
5. There is a 1-1 mapping between state graph and the software implementation



Finite State Machine (FSM)

What is a Finite State Machine?

- Set of inputs, outputs, states and transitions
- State graph defines input/output relationship

What is a state?

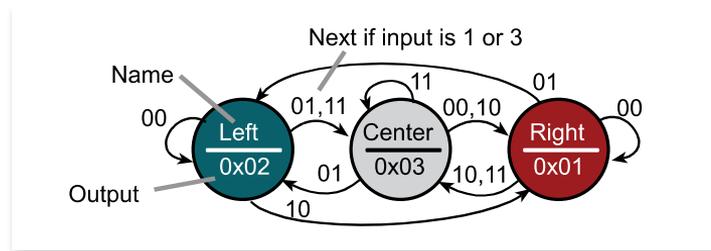
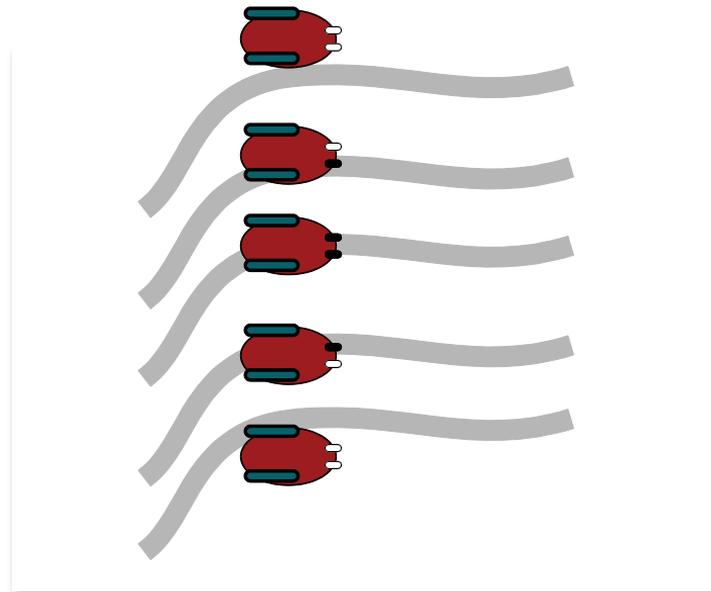
- Description of current conditions
- What you believe to be true

What is a state transition graph (or table)?

- Graphical interconnection between states

What is a controller?

- Software that inputs, outputs, changes state
- Accesses the state graph

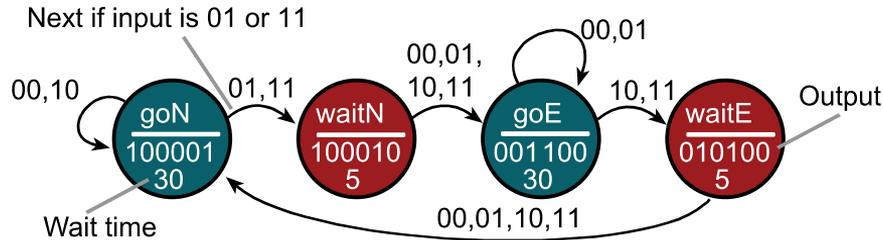
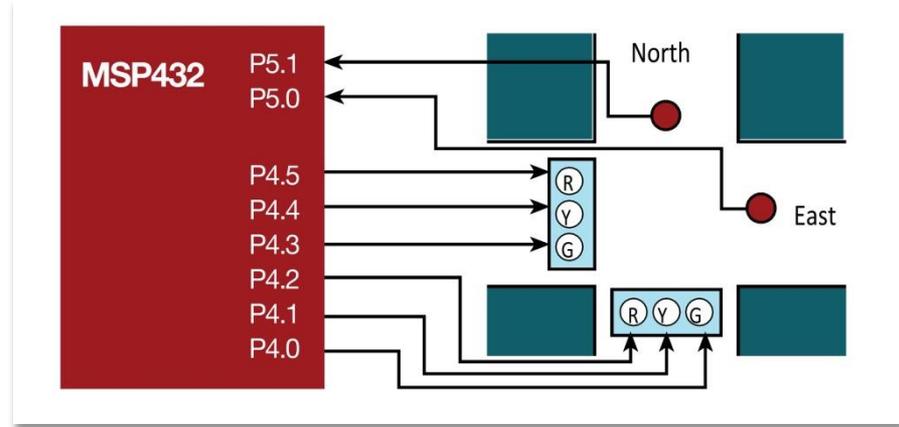




Finite State Machine (FSM)

What is a Finite State Machine (FSM)?

- Inputs (sensors)
- Outputs (actuators)
- Controller
- State transition graph



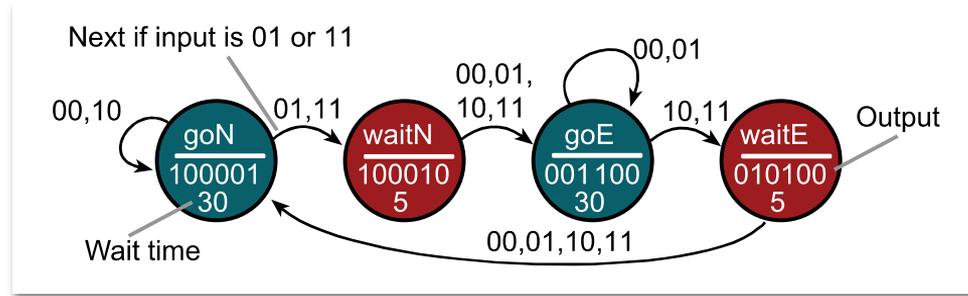
Controller loop

1. Output
2. Wait
3. Input
4. Next



Traffic Light Controller

State Transition Graph (STG)



State Transition Table (STT)

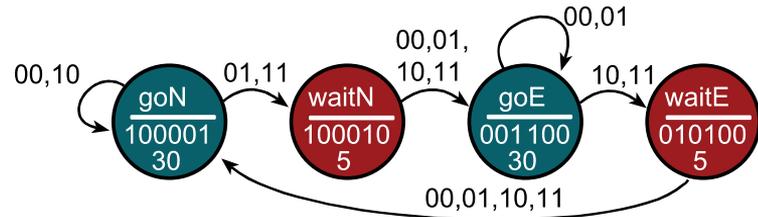
State \ Input	00	01	10	11
goN (100001,30)	goN	waitN	goN	waitN
waitN (100010,5)	goE	goE	goE	goE
goE (001100,30)	goE	goE	waitE	waitE
waitE (010100,5)	goN	goN	goN	goN



FSM Data Structure in C (Index into array)

```
const struct State {
    uint32_t Out;        // 6-bit output
    uint32_t Time;      // 1 ms units
    uint32_t Next[4];   // list of next states
};
typedef const struct State State_t;

#define goN    0
#define waitN  1
#define goE    2
#define waitE  3
State_t FSM[4] = {
    {0x21, 30000, {goN,waitN,goN,waitN}},
    {0x22,  5000, {goE,goE,goE,goE}},
    {0x0C, 30000, {goE,goE,waitE,waitE}},
    {0x14,  5000, {goN,goN,goN,goN}}
};
```



FSM Engine in C (Index into array)

```
void main(void) {
    uint32_t cs;    // index of current state
    uint32_t input; // car sensor input
    Traffic_Init(); // initialize ports and timer
    cs = goN;      // initial state
    while(1){
        // 1) set lights to current state's Out
        P4->OUT = (P4->OUT&~0x3F) | (FSM[cs].Out);
        // 2) specified wait for this state
        Clock_Delay1ms(FSM[cs].Time);
        // 3) input from car detectors
        input = (P5->IN&0x03);
        // 4) next depends on state and input
        cs = FSM[cs].Next[input];
    }
}
```

Friendly

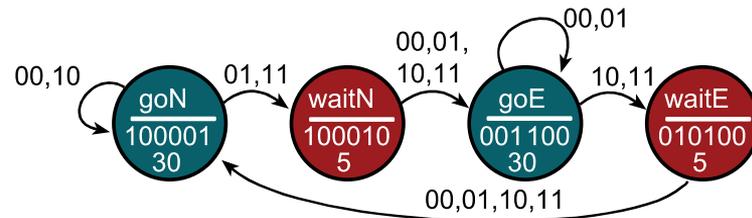
9 SysTick

Mask

FSM Data Structure in C (Pointer)

```
const struct State {
    uint32_t Out;      // 6-bit output
    uint32_t Time;    // 1 ms units
    const struct State *Next[4]; // next states
};
typedef const struct State State_t;

#define goN    &FSM[0]
#define waitN  &FSM[1]
#define goE    &FSM[2]
#define waitE  &FSM[3]
State_t FSM[4] = {
    {0x21, 30000, {goN, waitN, goN, waitN}},
    {0x22, 5000, {goE, goE, goE, goE}},
    {0x0C, 30000, {goE, goE, waitE, waitE}},
    {0x14, 5000, {goN, goN, goN, goN}}
};
```





FSM Engine in C (Pointer)

```
void main(void) {  
    State_t *pt;    // pointer to current state  
    uint32_t input; // car sensor input  
    Traffic_Init(); // initialize ports and timer  
    pt = goN;      // initial state  
    while(1){  
        // 1) set lights to current state's Out  
        P4->OUT = (P4->OUT&~0x3F) | (pt->Out);  
        // 2) specified wait for this state  
        Clock_Delay1ms(pt->Time);  
        // 3) input from car detectors  
        input = (P5->IN&0x03);  
        // 4) next depends on state and input  
        pt = pt->Next[input];  
    }  
}
```

Friendly

9 SysTick

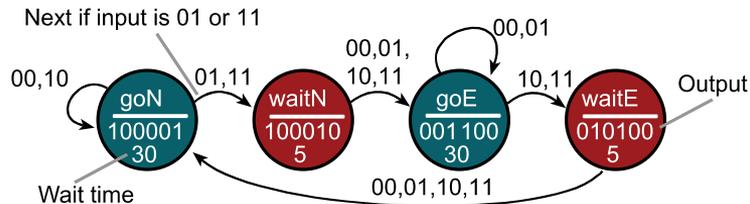
Mask



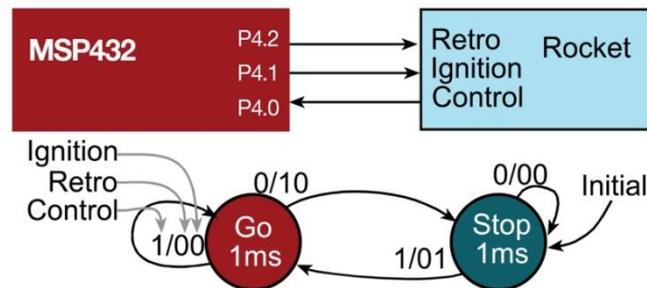
Mealy versus Moore

- Moore FSM
 - Output value depends on current state
 - Significance is the state
 - Input: when to change state
 - Output: how to be or what to do while in that state
- Mealy FSM
 - Output value depends on input and current state
 - Significance is the state transition
 - Input: when to change state
 - Output: how to change state

Inputs: Car sensors
Outputs: Traffic lights



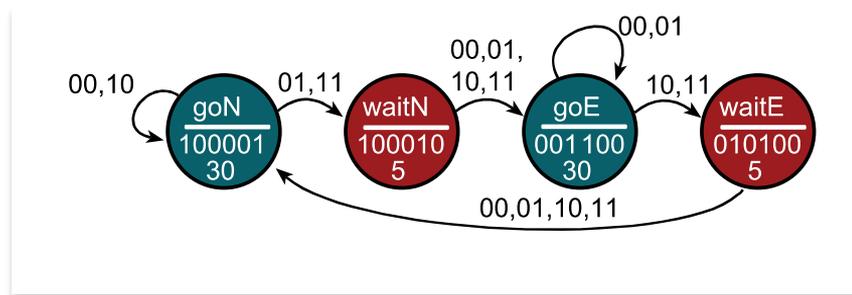
Inputs: Control
Outputs: Retro, Ignition





Summary

- Abstraction
 - Define a problem
 - Concepts / principles / processes
 - Separation of policy and mechanisms
 - Interfaces define what it does (policy)
 - Implementations define how it works (mechanisms)
- Finite State Machines
 - Inputs (sensors)
 - Outputs (actuators)
 - Controller
 - State graph
 - States
 - Implementations define how it works (mechanisms)



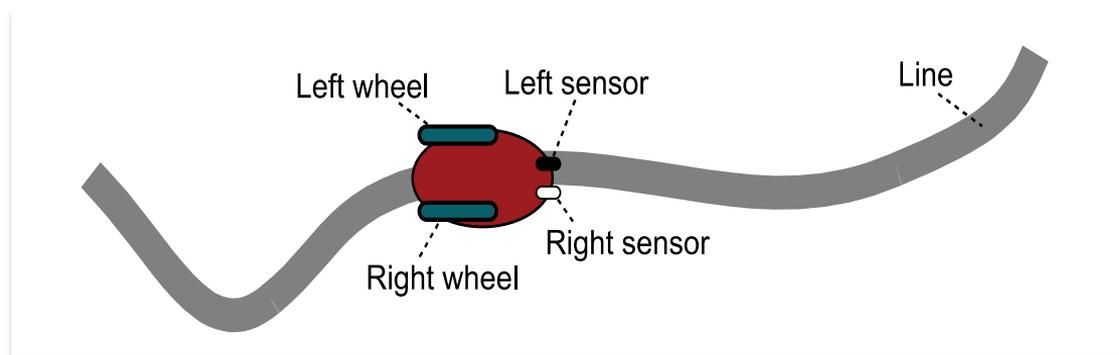


Module 7

Lecture: Finite State Machines – Line Follower



Simple Line Tracker



Two Sensors

1,1 on line
1,0 off to the right
0,1 off to the left
0,0 lost

Left, Right

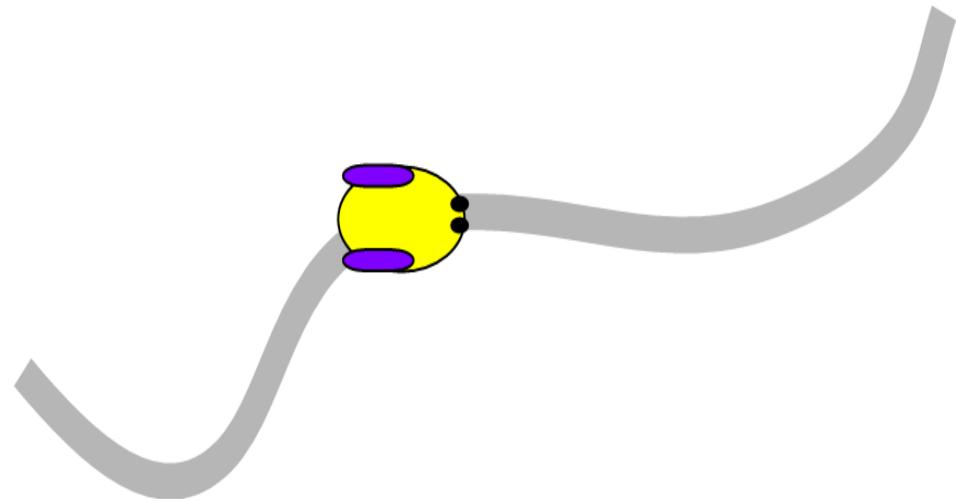
Two Motors

1,1 go straight
1,0 turn right
0,1 turn left

Left, Right

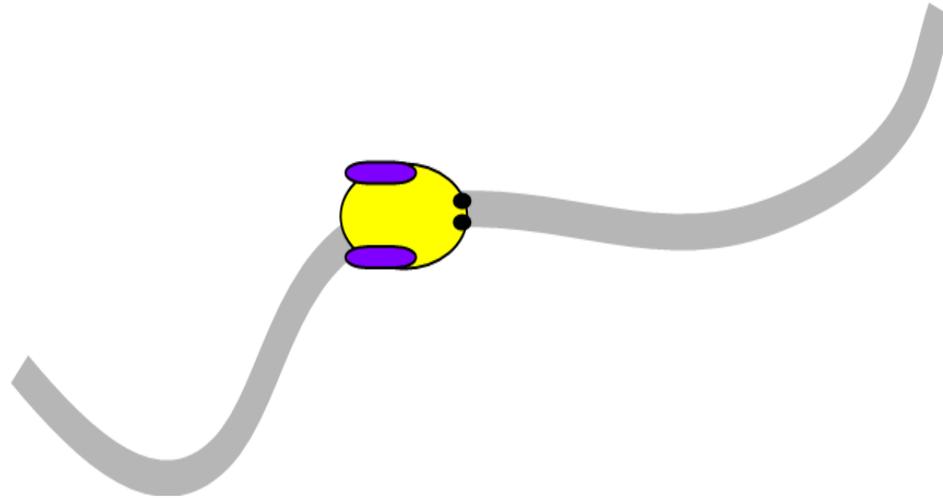


Strategy



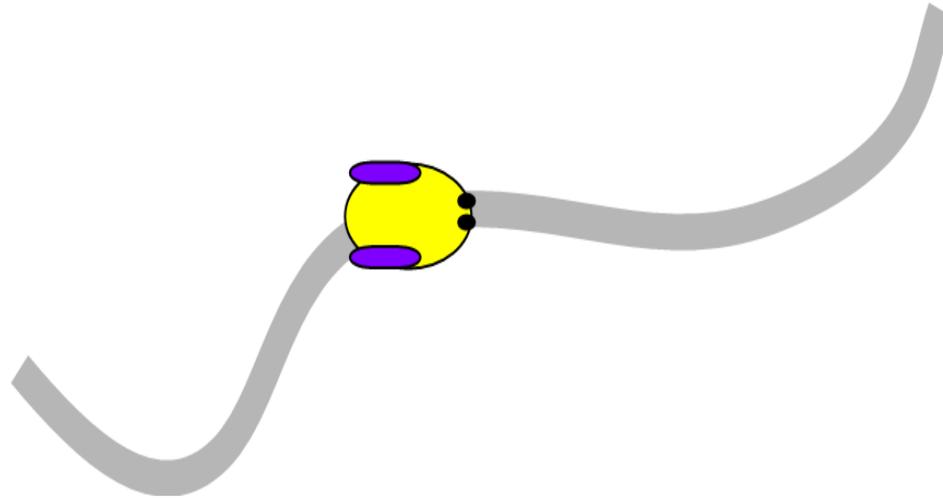


Strategy



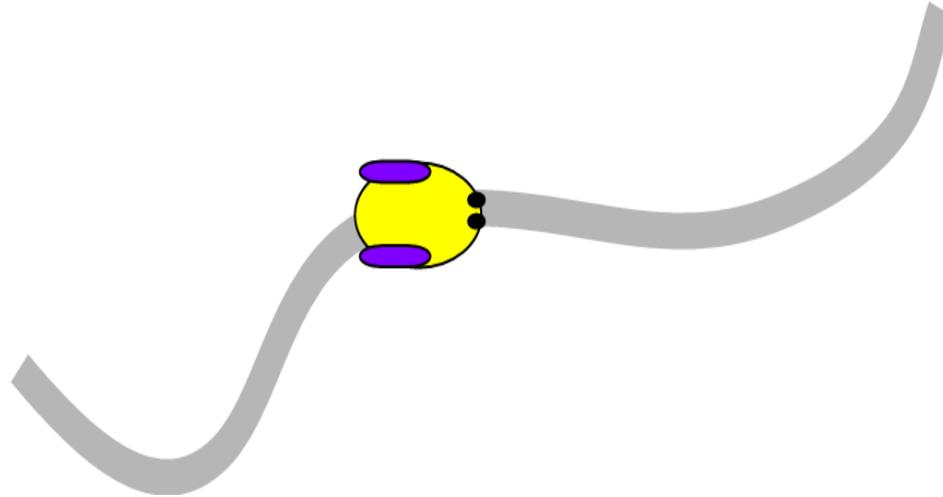


Strategy



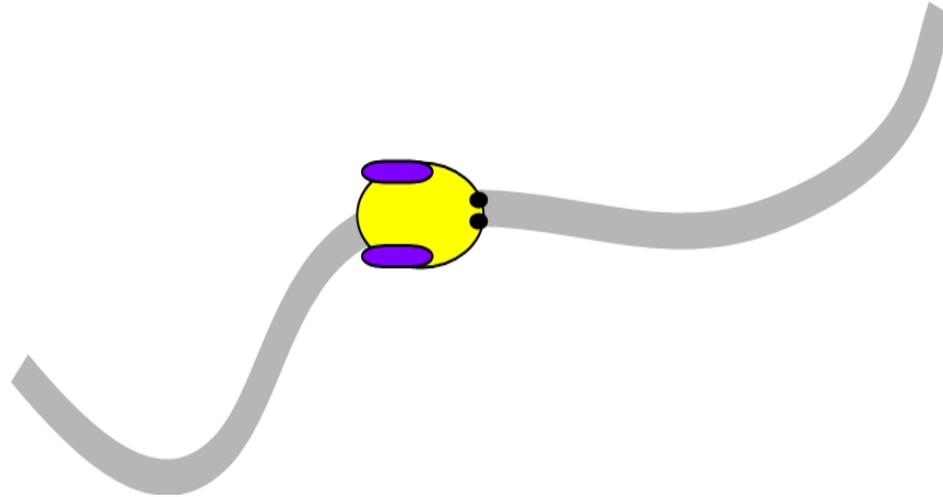


Strategy



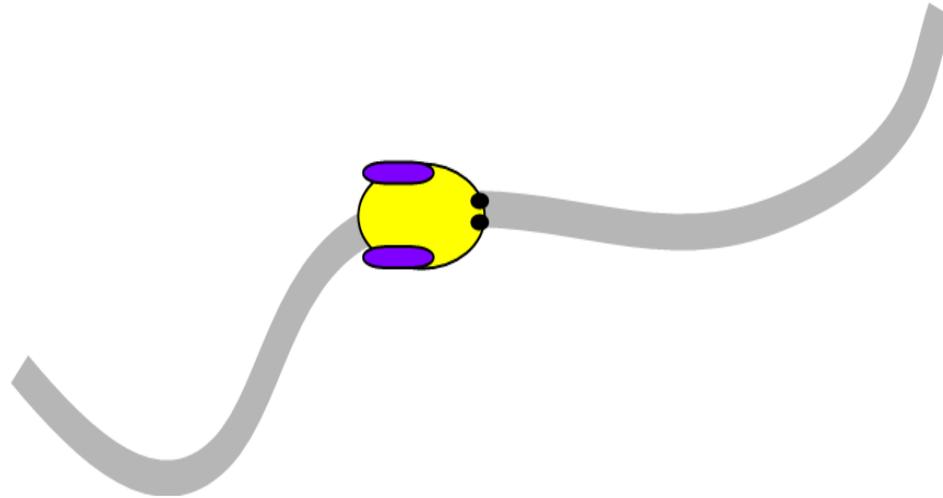


Strategy



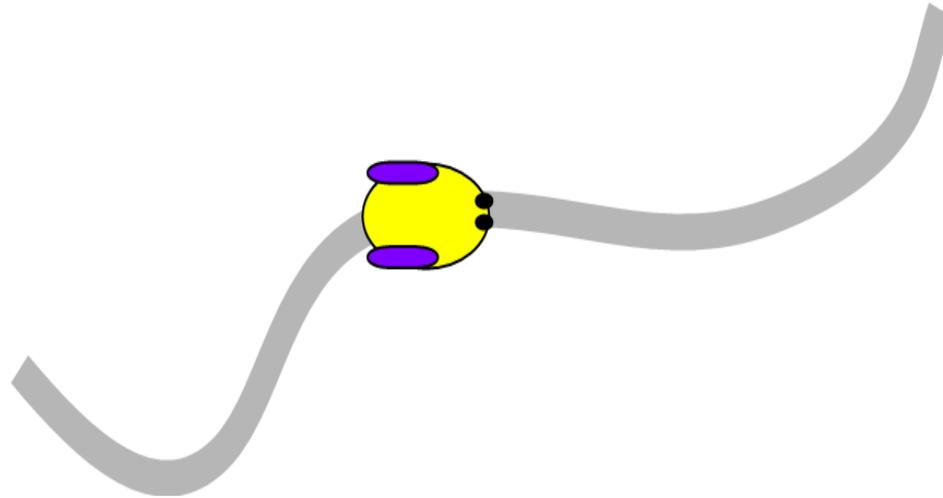


Strategy



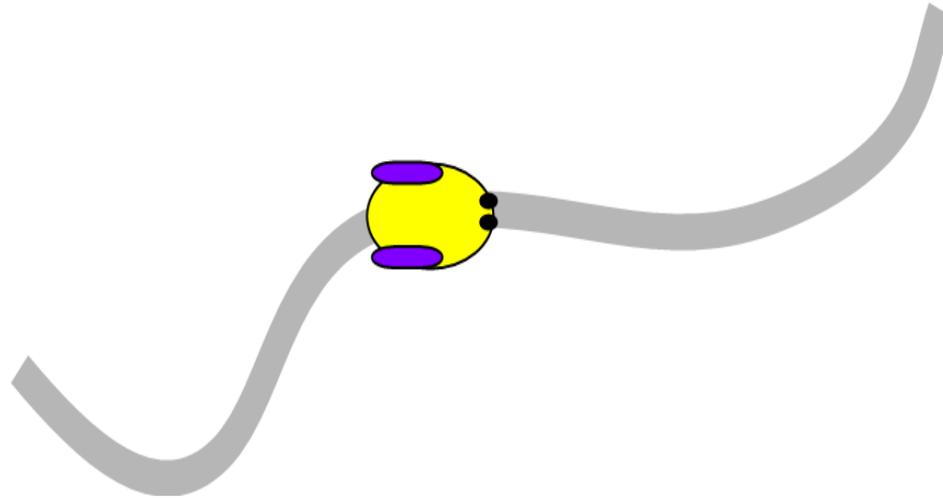


Strategy



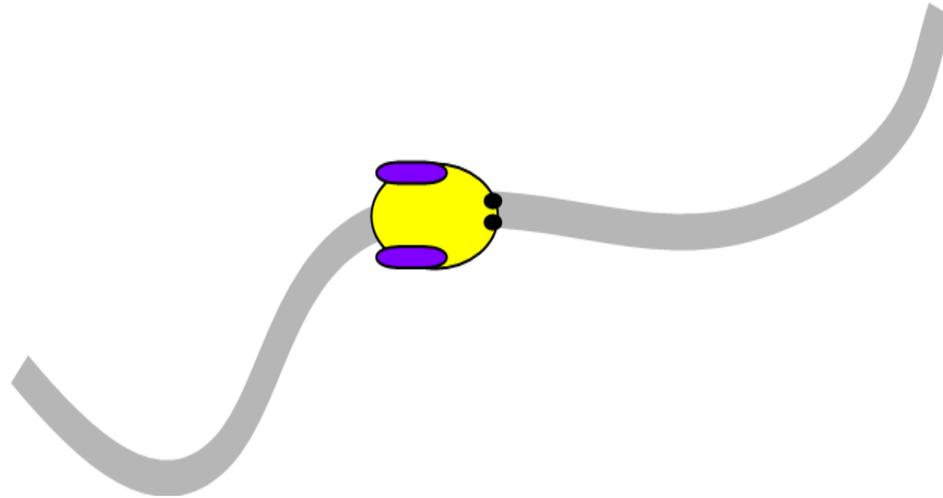


Strategy



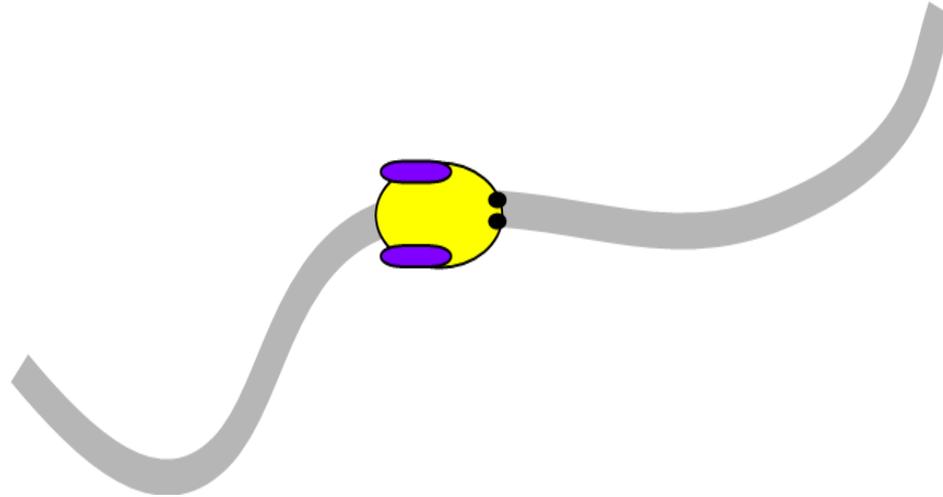


Strategy



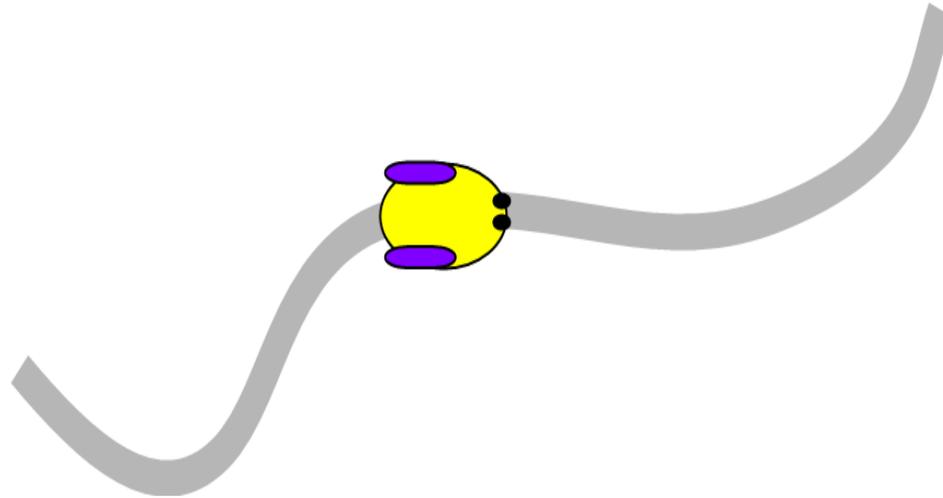


Strategy



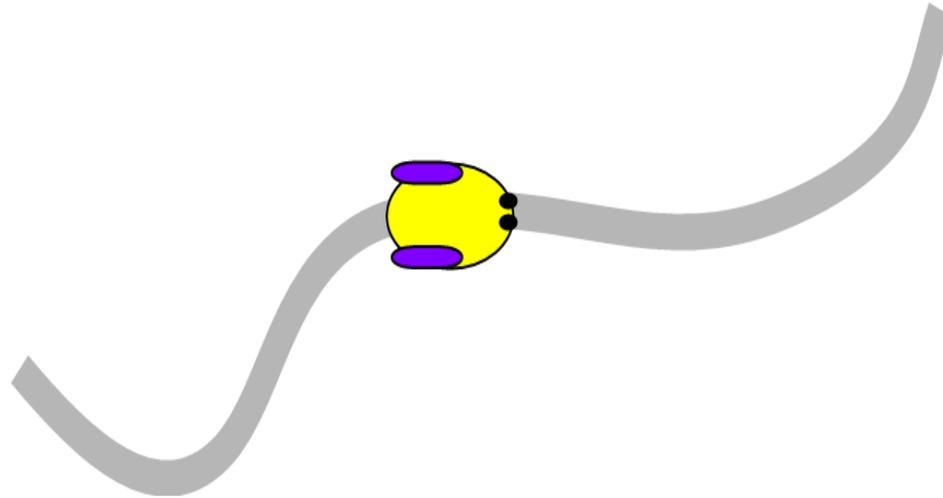


Strategy



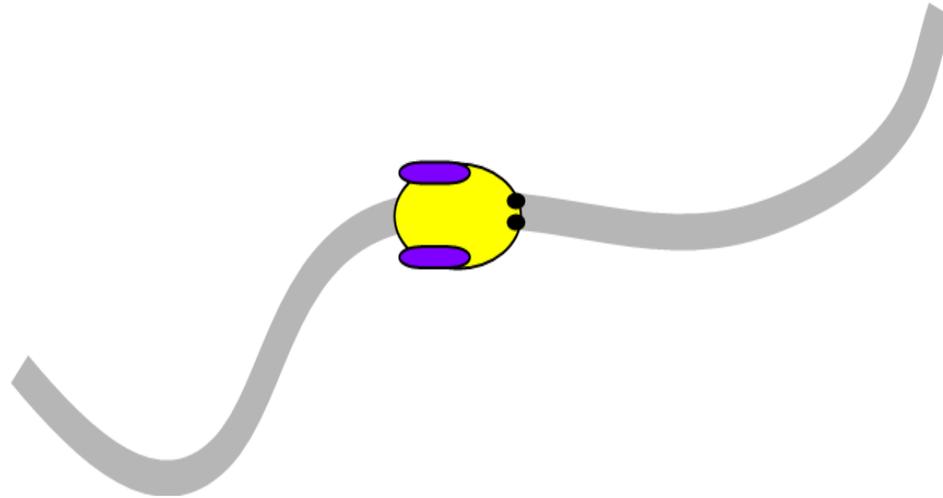


Strategy



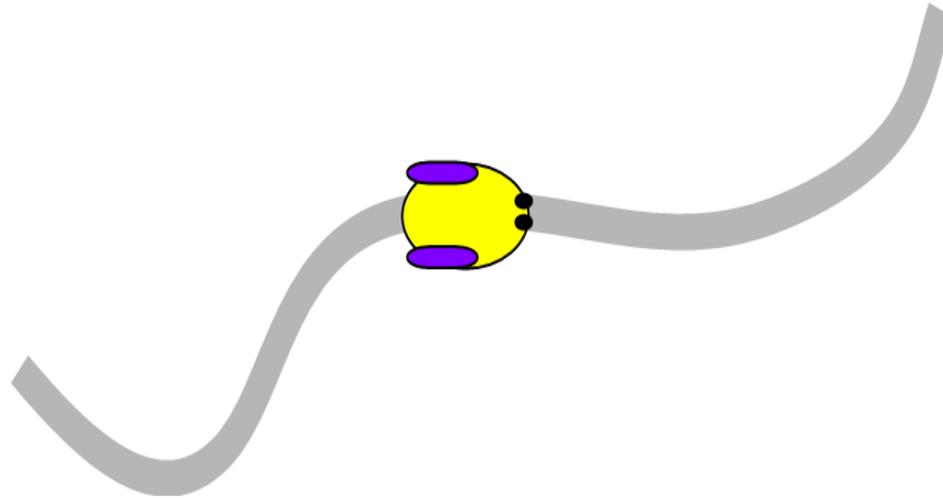


Strategy



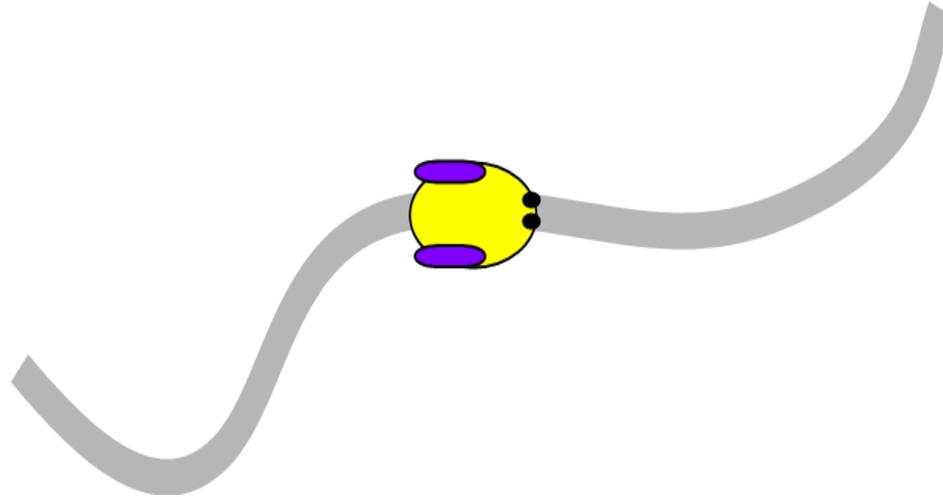


Strategy



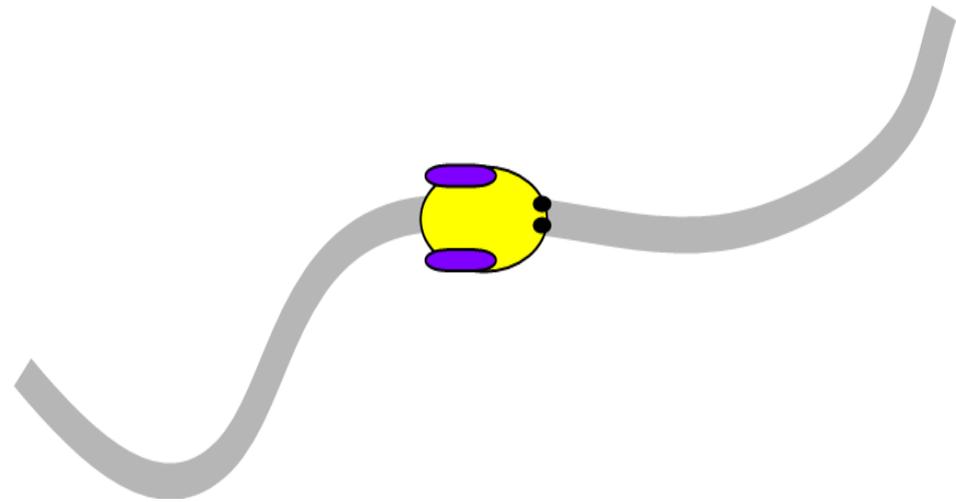


Strategy



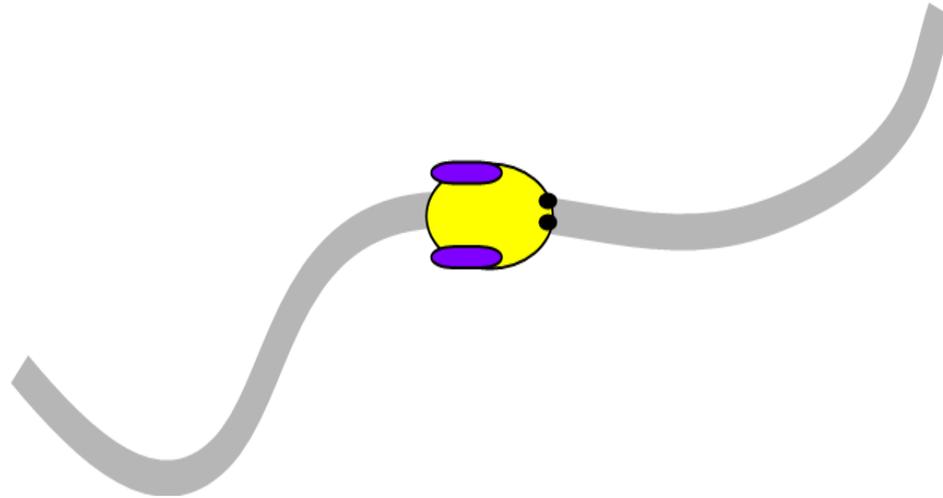


Strategy



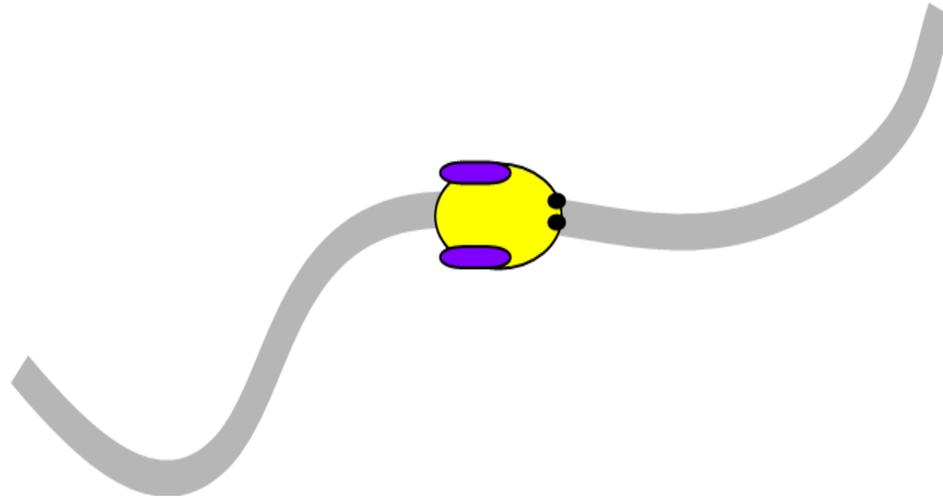


Strategy



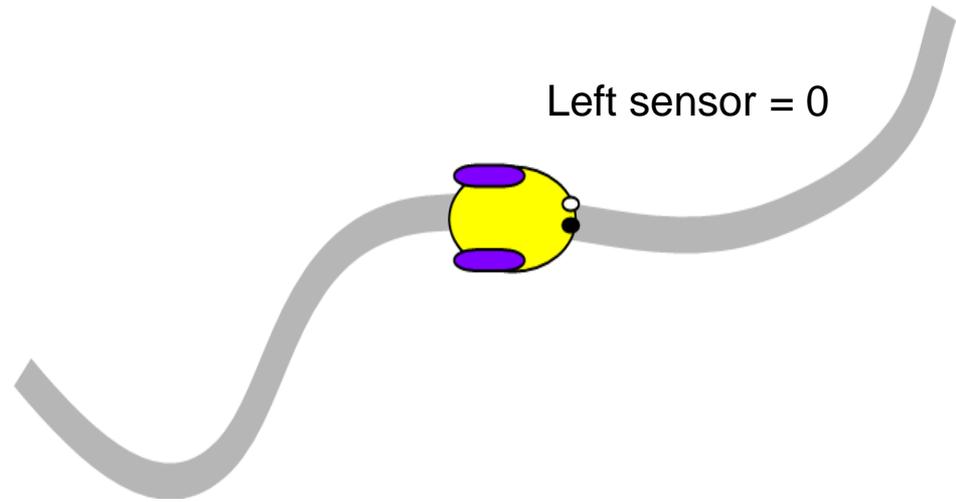


Strategy



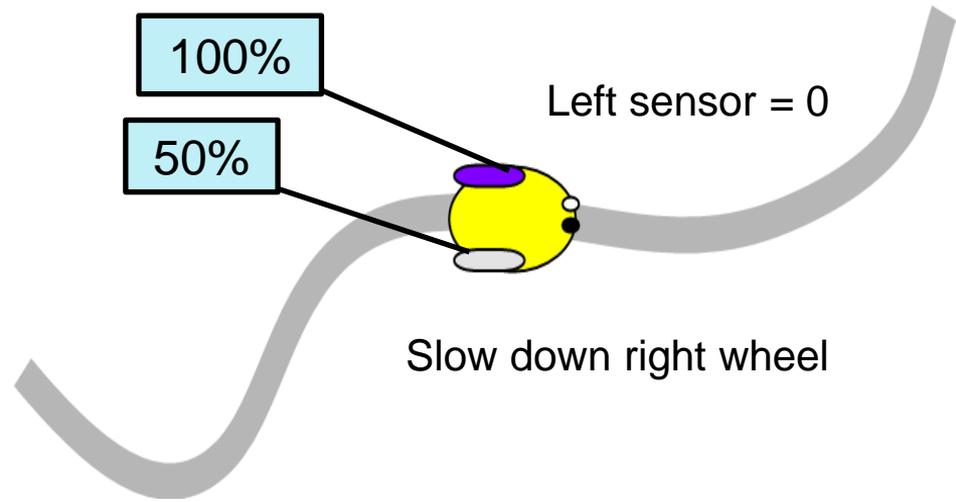


Strategy



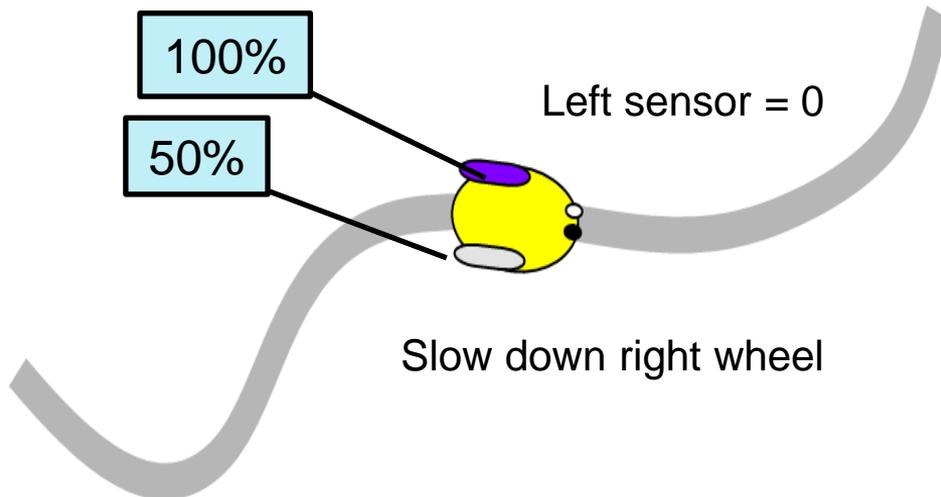


Strategy



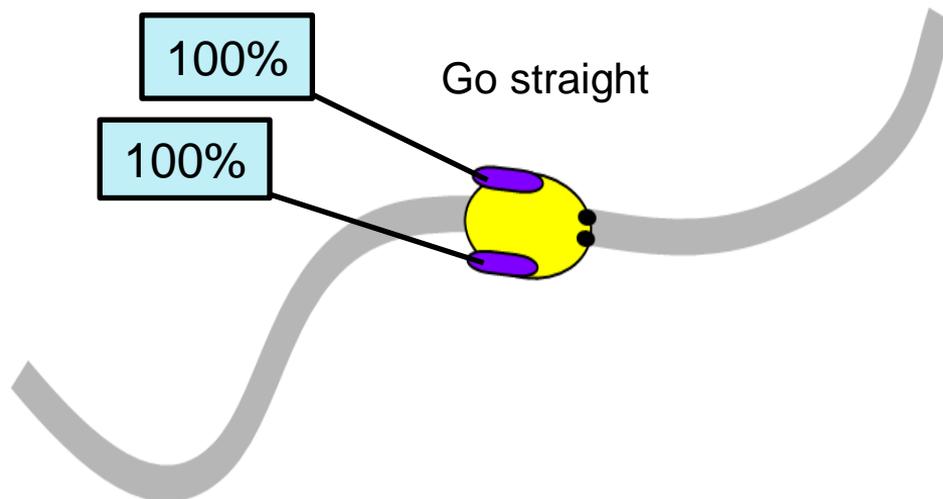


Strategy



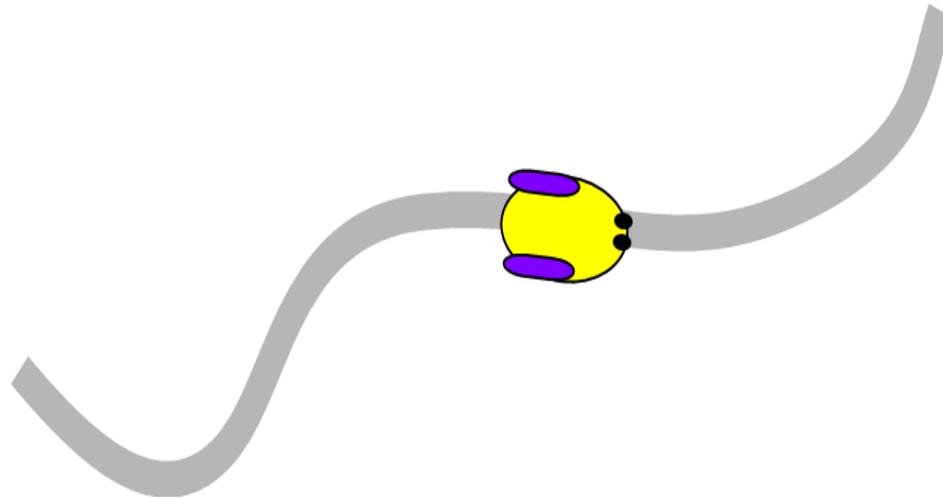


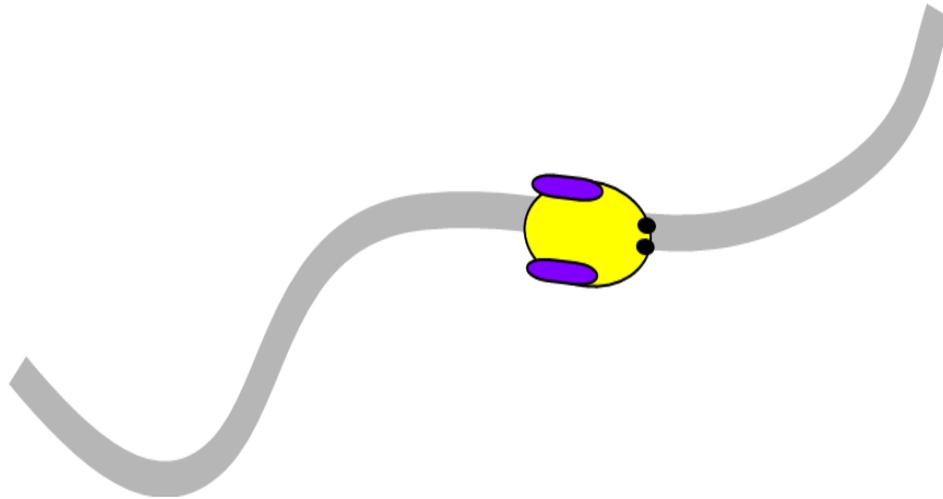
Strategy





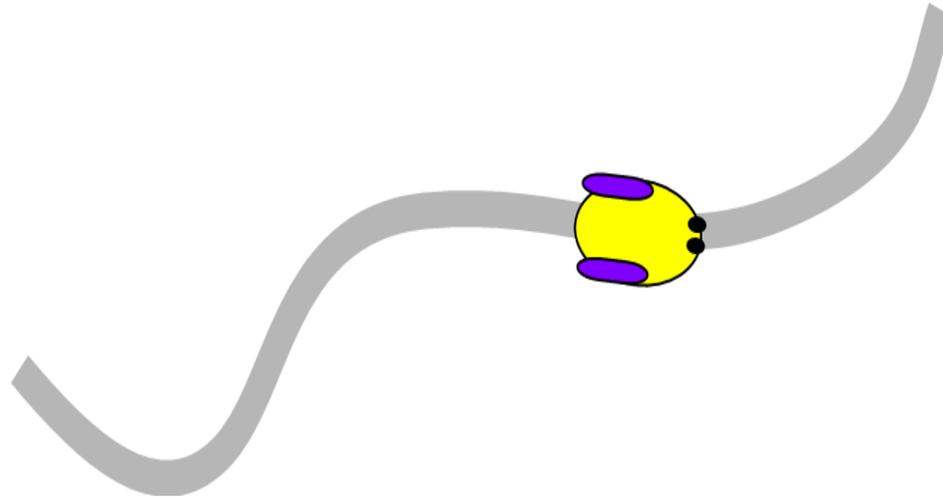
Strategy





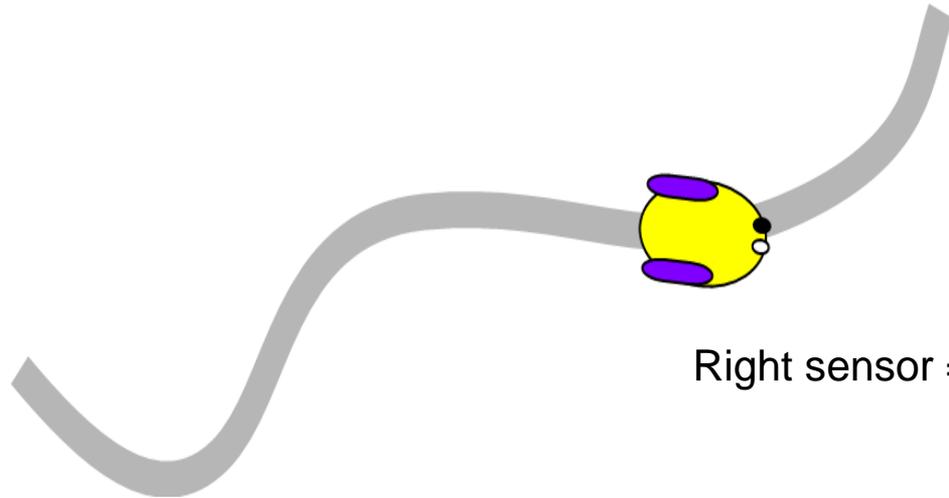


Strategy



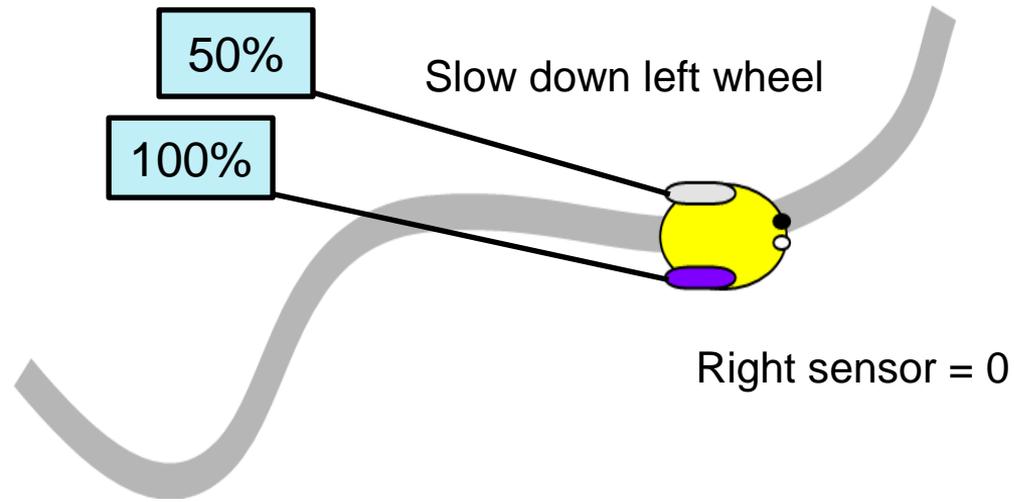


Strategy



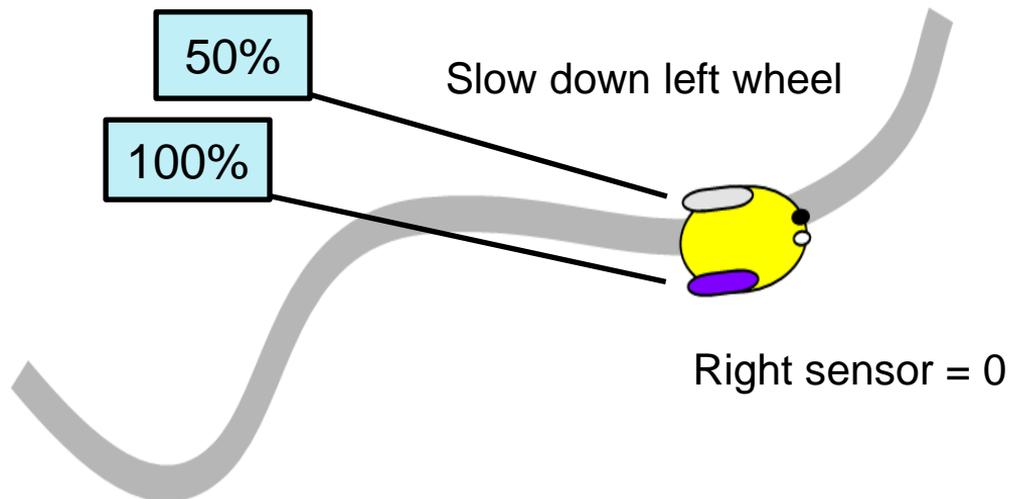


Strategy



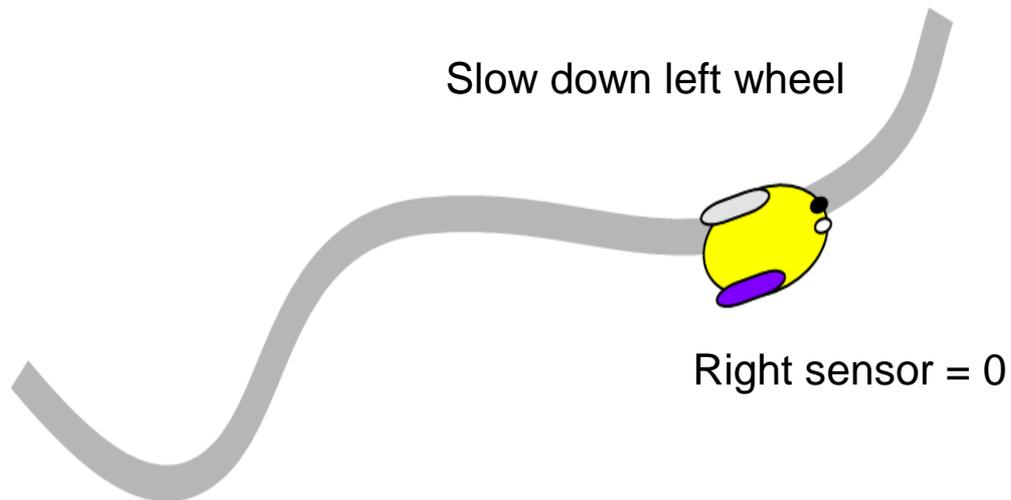


Strategy



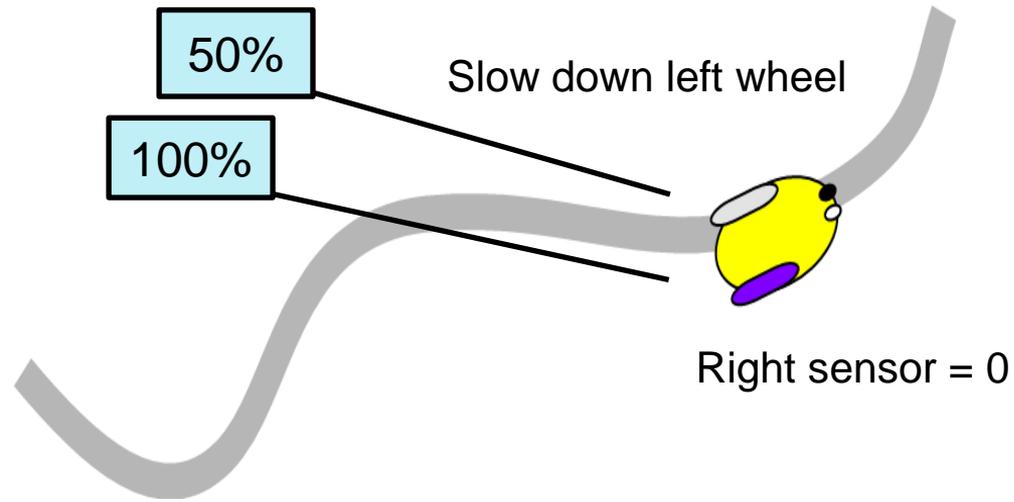


Strategy



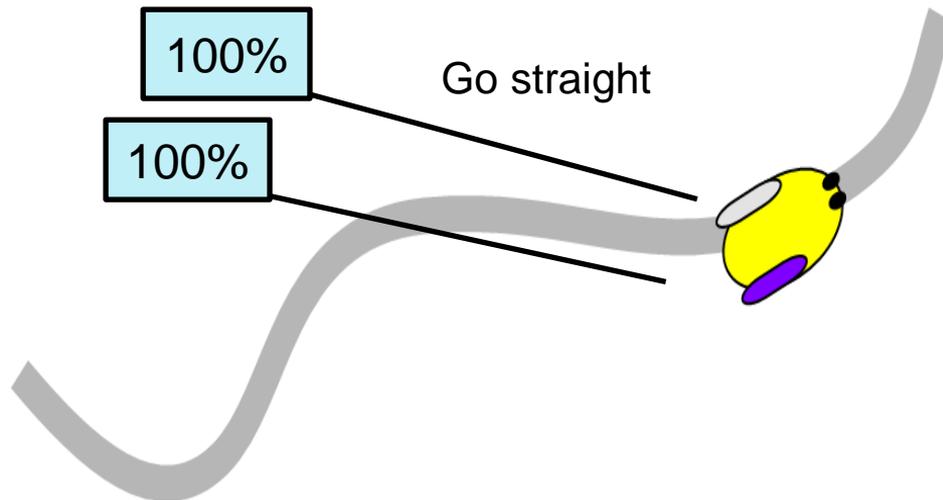


Strategy





Strategy





Strategy





States

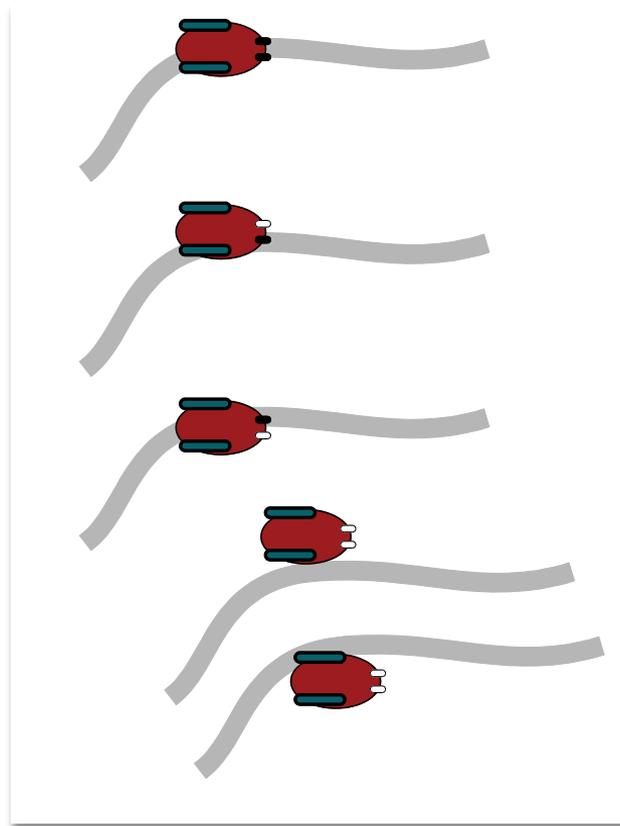
State Motor

Center 1,1

Left 0,1

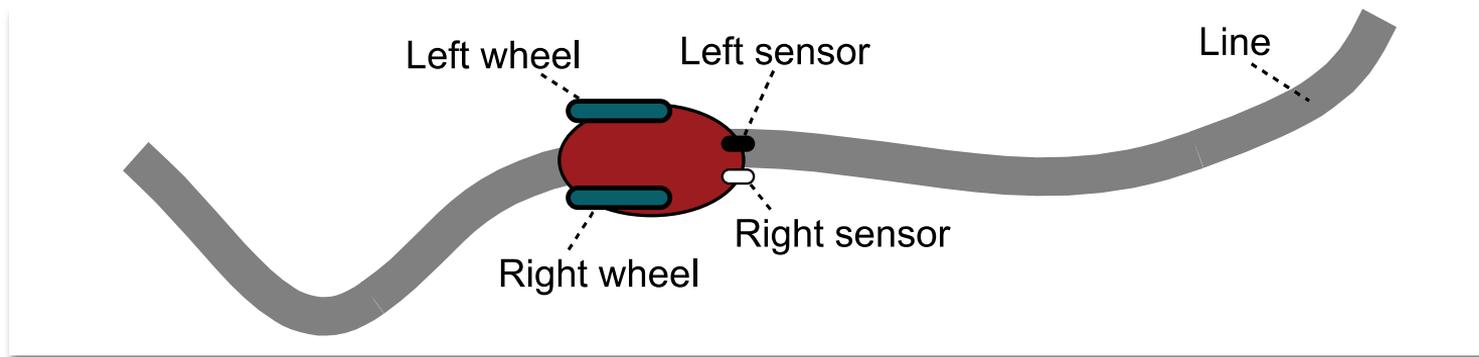
Right 1,0

Motors respond in 100ms,
so run FSM every 50ms





Simple Line Tracker



Two Sensors

1,1 on line

0,1 off to the left

1,0 off to the right

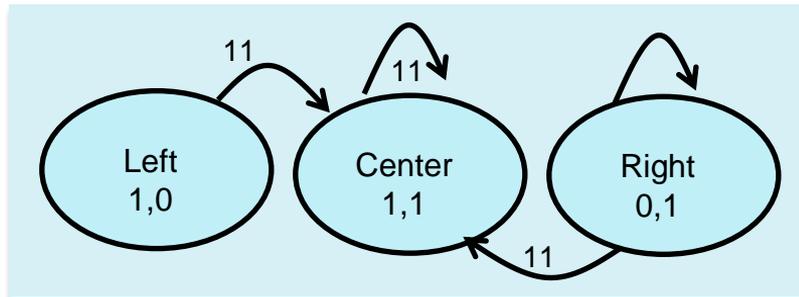
0,0 lost

Left, Right



State Transition Table

State	Motor	In=0,0	In=0,1	In=1,0	In=1,1
Center	1,1				Center
Left	1,0				Center
Right	0,1				Center



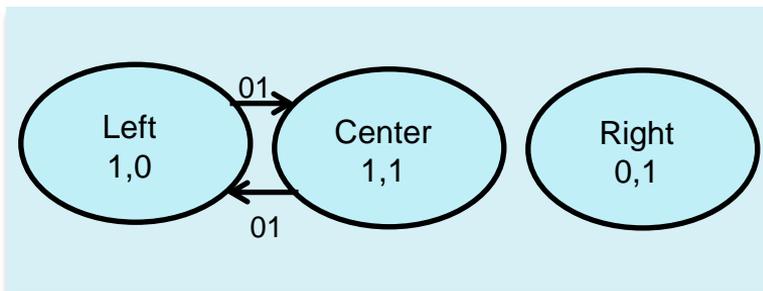
```
State_t fsm[3]={
  {0x03, 1, {
    Center }},
  {0x02, 1, {
    Center }},
  {0x01, 1, {
    Center }}
};
```

On the line, so go straight



State Transition Table

State	Motor	In=0,0	In=0,1	In=1,0	In=1,1
Center	1,1		Left		
Left	1,0		Center		
Right	0,1				



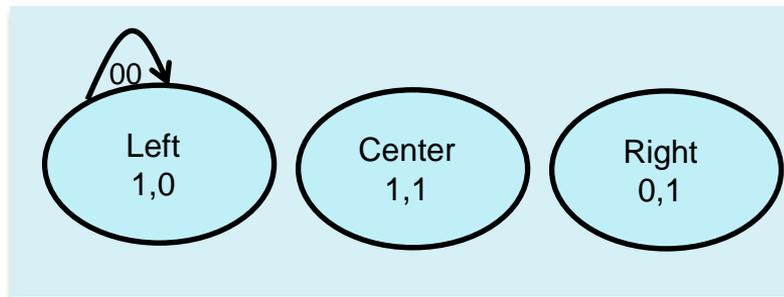
```
State_t fsm[3]={
  {0x03, 1, { Left,           }},
  {0x02, 1, { Center,       }},
  {0x01, 1, {           }}
};
```

Off to left, so toggle right motor, turn right



State Transition Table

State	Motor	In=0,0	In=0,1	In=1,0	In=1,1
Center	1,1				
Left	1,0	Left			
Right	0,1				



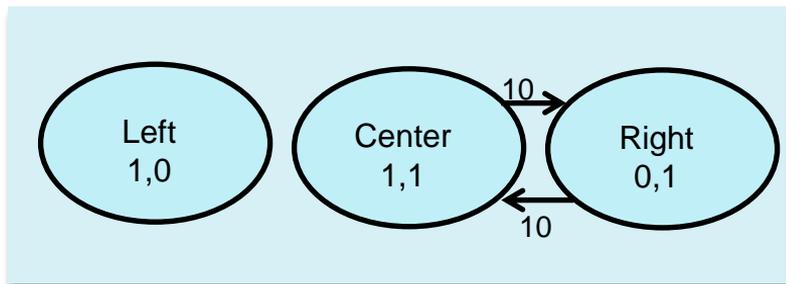
```
State_t fsm[3]={
  {0x03, 1, {
    Center }},
  {0x02, 1, { Left,
    Center }},
  {0x01, 1, {
    Center }}
};
```

Way off to left, so stop right motor, turn right



State Transition Table

State	Motor	In=0,0	In=0,1	In=1,0	In=1,1
Center	1,1			Right	
Left	1,0				
Right	0,1			Center	



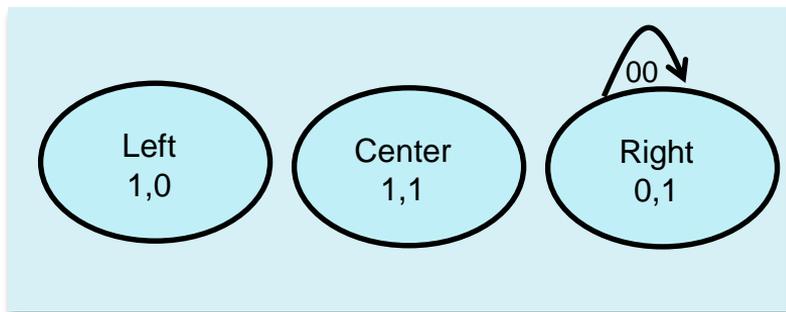
```
State_t fsm[3]={
  {0x03, 1, {      Right      }},
  {0x02, 1, {      }},
  {0x01, 1, {      Center     }}
};
```

Off to right, so toggle left motor, turn left



State Transition Table

State	Motor	In=0,0	In=0,1	In=1,0	In=1,1
Center	1,1				
Left	1,0				
Right	0,1	Right			



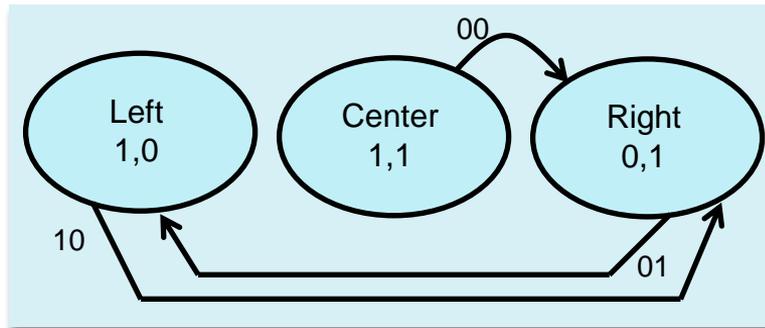
```
State_t fsm[3]={
  {0x03, 1, {
  {0x02, 1, {
  {0x01, 1, { Right,
};
```

Way off to right, so stop left motor, turn left



State Transition Table

State	Motor	In=0,0	In=0,1	In=1,0	In=1,1
Center	1,1	Right			
Left	1,0			Right	
Right	0,1		Left		



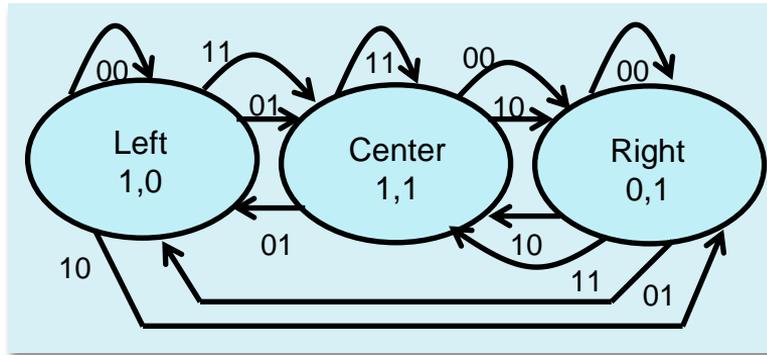
```
State_t fsm[3]={
  {0x03, 1, { Right,           }},
  {0x02, 1, {           Right }},
  {0x01, 1, { Left,           }}
};
```

Weird things that shouldn't happen



State Transition Table

State	Motor	In=0,0	In=0,1	In=1,0	In=1,1
Center	1,1	Right	Left	Right	Center
Left	1,0	Left	Center	Right	Center
Right	0,1	Right	Left	Center	Center



```
State_t fsm[3]={
  {0x03, 1, {Right, Left, Right, Center}},
  {0x02, 1, {Left, Center, Right, Center}},
  {0x01, 1, {Right, Left, Center, Center}}
};
```

Motors respond in 100ms, so run FSM every 10ms



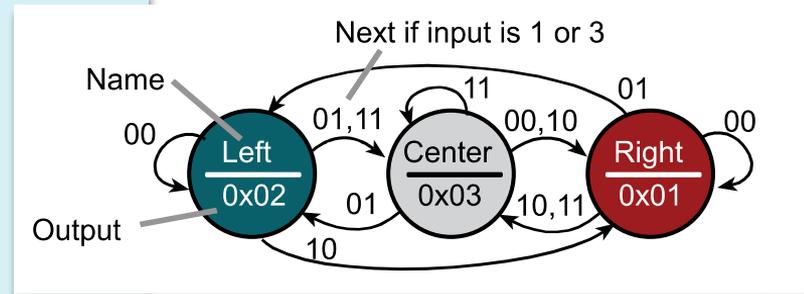
Robot Implementation

```

struct State {
    uint32_t out;           // 2-bit output
    uint32_t delay;        // time to delay in lms
    const struct State *next[4]; // Next if 2-bit input is 0-3
};
typedef const struct State State_t;

#define Center &fsm[0]
#define Left   &fsm[1]
#define Right  &fsm[2]
State_t fsm[3]={
    {0x03, 50, { Right, Left,  Right, Center }}, // Center
    {0x02, 50, { Left,  Center, Right, Center }}, // Left
    {0x01, 50, { Right, Left,  Center, Center }}, // Right
};
State_t *Spt; // pointer to the current state
uint32_t Input; // 00=off, 01=right, 10=left, 11=on
uint32_t Output; // 3=straight, 2=turn right, 1=turn left
int main(void){
    Clock_Init48MHz();
    Motor_Stop(); // initialize DC motors
    Spt = Center;
    while(1){
        Output = Spt->out; // set output from FSM
        Motor_Output(Output); // do output to two motors
        Clock_Delaylms(Spt->delay); // wait
        Input = Reflectance_Center(1000); // read sensors
        Spt = Spt->next[Input]; // next depends on input and state
    }
}

```



12 Motors

13 Timers

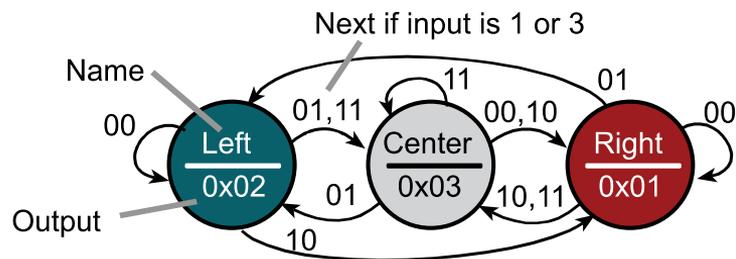
9 SysTick

6 GPIO



Summary

- Abstraction
 - Define a problem
 - Concepts / principles / processes
 - Separation of policy and mechanisms
 - Interfaces define what it does (policy)
 - Implementations define how it works (mechanisms)
- Finite State Machines
 - Inputs (sensors)
 - Outputs (actuators)
 - Controller
 - State graph
 - States
 - Implementations define how it works (mechanisms)



ti.com/rslk



IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to TI's Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2019, Texas Instruments Incorporated