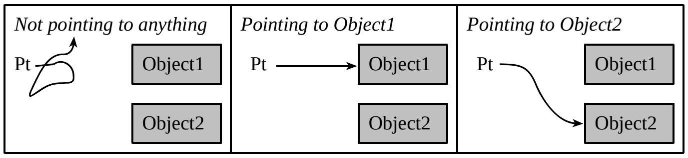
Recap

- **Finite State Machines**
  - **Being in a state has meaning**
  - **Moore: the output is related to being in a state**
  - **Mealy: the output is required to change state**
  - **Arrows are state transitions: pointers**
  - **1-1 mapping from state graph to data structure**

Overview

- **Finite State Machines (Section 8.7)**
- **State graph to C**



Pointer is an address

Figure 6.1. Pointers are addresses pointing to objects. The objects may be data, functions, or other pointers.

We have used arrays in Lab 4
We have used the stack for subroutine calls
Lab 5 will create a graph in assembly
This lecture will implement the FSM in C

Arrays in C: Put in RAM if you want to change values

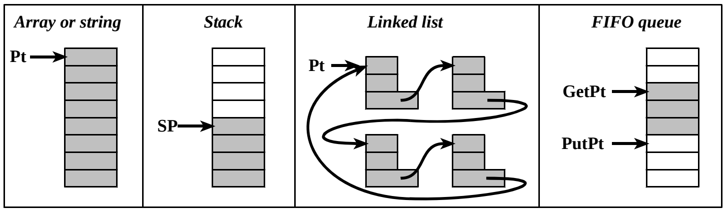
```
unsigned short Buffer[8];
```

Arrays in C: Put in ROM if values are fixed

```
const char Data[4]=
    {0x05,0x06,0x0A,0x09};
```

Arrays

- **Length**
- **Precision**
- **Signed/unsigned**
- **RAM or ROM**



Access arrays by index

```
unsigned char Index;
void Stepper_Init(void){
  DDRT |= 0x0F; // PT3-0 are outputs
  PTT = 0x09;   // first data
  Index = 0;    // first index
}
void Stepper_CW(void){
  PTT = Data[Index];      // rotate 15deg
  Index = 0x03&(Index+1); // next index
}
```

Access arrays by pointer

```
unsigned char *Pt;
void Stepper_Init(void){
  DDRT |= 0x0F; // PT3-0 are outputs
  PTT = 0x09;   // first data
  Pt = Data;    // pointer to first
}
void Stepper_CW(void){
  PTT = *Pt;      // rotate 15deg
  if(Pt == &Data[3]){
    Pt = Data;    // pointer to first
  } else{
    Pt++;         // next value
  }
}
```

Variable length arrays can use a termination code

```
const char Data[5]={0x05,0x06,0x0A,0x09,0};
unsigned char *Pt;
void Stepper_Init(void){
  DDRT |= 0x0F; // PT3-0 are outputs
  PTT = 0x09;   // first data
  Pt = Data;    // pointer to first
}
void Stepper_CW(void){
  PTT = *Pt;            // move stepper
  Pt++;                 // next address
  if(*Pt == 0){         // end?
    Pt = Data;          // start over
  }
}
```

6.5. Structures

Combine into one object multiple parts with

- Different types
  - signed numbers,
  - characters,
  - unsigned numbers,
  - pointers
- Different precision
  - 8-bit,
  - 16-bit
  - Arrays (must be fixed length)

```
const struct port{
  unsigned char AndMask;    // bits that can change
  unsigned char OrMask;     // bits that must stay high
  unsigned char *Addr;      // Port Address
  unsigned char Name[10];   // ASCII string
};
typedef const struct port portType;
portType PortT={0x15,0x82,0x0240,"PTT"};
```

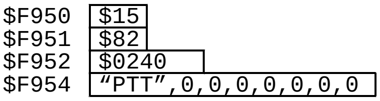


Figure 6.13. A structure collects objects of different sizes into one object.

```
void OutputT(unsigned char in){
unsigned char new,old;
  old = (*PortT.Addr);           // read previous value
  old = old & ~(PortT.AndMask); // clear bits
  new = in & PortT.AndMask;     // bits that can change
  new = new | PortT.OrMask;     // must be high
  new = new | old;
  (*PortT.Addr) = new;          // output
}
void OutputAny(portType *pt, unsigned char in){
unsigned char new,old;
  old = (*pt->Addr);           // read previous value
  old = old & ~(pt->AndMask); // clear bits
  new = in & pt->AndMask;     // bits that can change
  new = new | pt->OrMask;     // must be high
  new = new | old;
  (*pt->Addr) = new;          // output
}
```

Traffic Light Controller

PT1=0, PT0=0 means no cars exist on either road
PT1=0, PT0=1 means there are cars on the East road
PT1=1, PT0=0 means there are cars on the North road
PT1=1, PT0=1 means there are cars on both roads

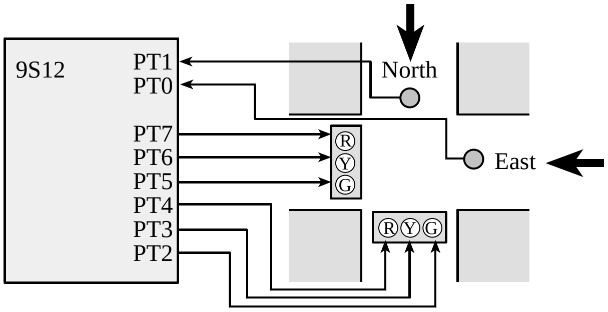


Figure 6.19. Traffic light interface.

`goN`, PT7-2 = 100001 makes it green on North and red on East
`waitN`, PT7-2 = 100010 makes it yellow on North and red on East
`goE`, PT7-2 = 001100 makes it red on North and green on East
`waitE`, PT7-2 = 010100 makes it red on North and yellow on East

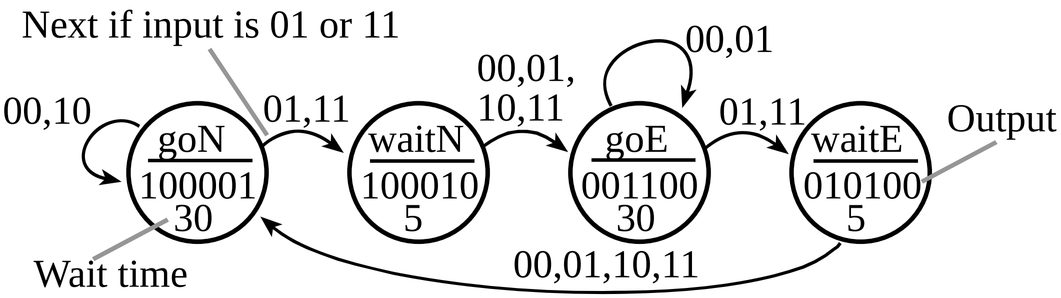


Figure 6.20. Graphical form of a Moore FSM that implements a traffic light.

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`**

Table 6.4. Tabular form of a Moore FSM that implements a traffic light.

```
// Linked data structure
const struct State {
  unsigned char Out;
  unsigned short Time;
  const struct State *Next[4];};
typedef const struct State STyp;
#define goN   &FSM[0]
#define waitN &FSM[1]
#define goE   &FSM[2]
#define waitE &FSM[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}}};
void main(void){
STyp *Pt;  // state pointer
unsigned char Input;
  Timer_Init();
  DDRT = 0xFC;  // lights and sensors
  Pt = goN;
  while(1){
    PTT = Pt->Out<<2;  // set lights
    Timer_Wait10ms(Pt->Time);
    Input = PTT&0x03;  // read sensors
    Pt = Pt->Next[Input];
  }
}
```

How do we prove to the judge it works?

Log all **`(input,time,output)`** *data* (like Lab 4*)*

Prove it works for a machine with a few states

then show the 1-1 mapping

The bottom line

FSM is good if:

1) the FSM is easy to understand,
2) the FSM is easy to change,
3) the state graph defines exactly what it does,
4) the state graph is 1-1 with the data structure,
5) each state has the same format.

In other words, if all you see is the state graph, there should be no ambiguity about what the machine does.