# Finite State Machines implementation is Software

Dinesh Sharma
Microelectronics Group, EE Department
IIT Bombay, Mumbai

October 10, 2013

# Remembering ‘History’!

- The response of a system may depend not only on its current inputs but also on what has gone on previously.
- If the output depended only on the current inputs, we could express it as a function of its inputs and implement this function.
- But how do we handle dependence on ‘history’?

# Representation of ‘History’

- In a digital system, if the total number of permutations of events which could take place is finite, we could encode the ‘history’ as a digital word of appropriate size.
- This word would represent each possible sequence of events by a unique combination of ‘1’s and ‘0’s.
- Now we can represent ‘history’ by this word and derive the output as a logic function of the current inputs *and* this word.
- Obviously, every time an event occurs on the inputs, not only do we need to compute a new output but also update this word representing ‘history’.

# What are ‘States’?

- The word representing history denotes the current “state” of the system.
- Since the total number of distinct states which can occur is assumed to be finite, this kind of system is called a finite state machine (FSM).
- We can reduce the complexity of this system if we insist that two states are distinct if and only if the behaviour of the system is different for some sequence of events for these two states.

# When are states distinct

To give an example, suppose we have a penny in the slot type machine which accepts coins of 1 Rupee and 2 Rupee denominations.

- The subsequent behaviour of this machine is identical when it has received two 1 Rupee coins in the past or one 2 Rupee coin.
- Thus, even though the detailed history in the two cases is different, we can say that the “state” of the system is the same.
- We compute the output and the “next state” as functions of current inputs and the “current state”.

# Synchronous and Asynchronous FSMs

- Synchronous Finite State Machines update their outputs and their “current state” (to the computed “next state” value) at each tick of some global clock.
- Changes in inputs between clock ticks are ignored and it is assumed that a well behaved system will not allow the inputs to change while these are being sampled.
- Asynchronous state machines carry out this updating at each change in their inputs.

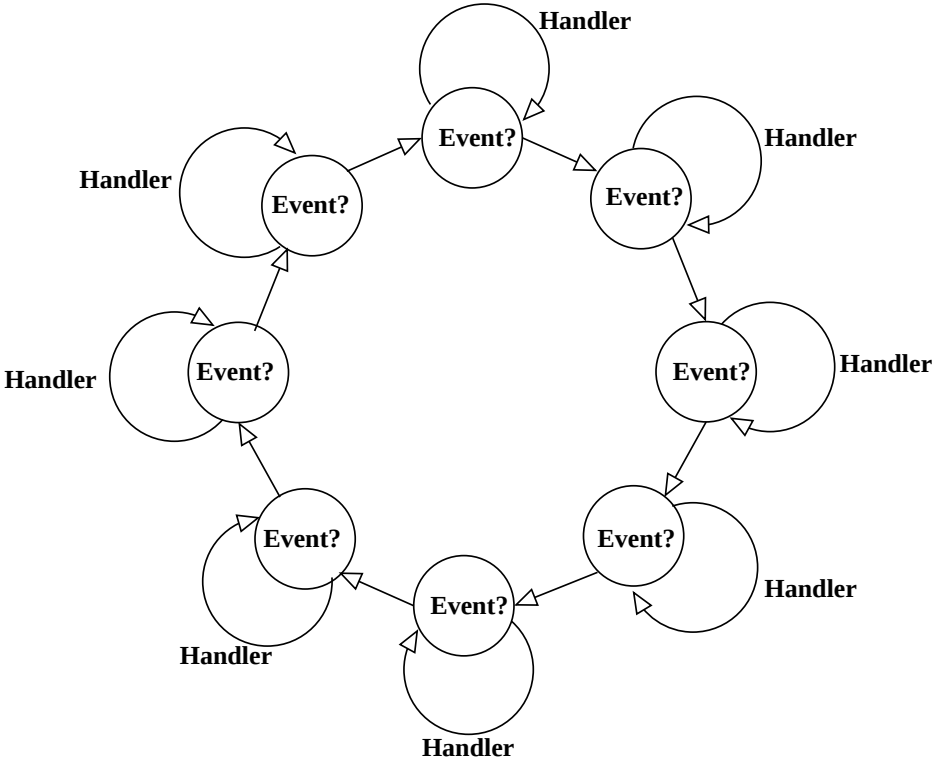
# Moore and Mealy FSMs

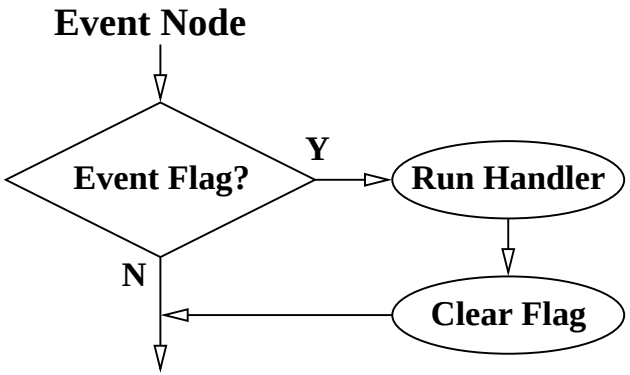
- A finite state machine whose outputs are functions of their "current state" only and not of their current inputs are called Moore machines.
- When the output is a function of the current state *as well as* the current inputs, the FSM is called a Mealy machine.

We illustrate the use of a finite state implementation for debouncing a 4x4 keyboard. For this, we describe the application and the hardware first and then discuss the FSM implementation for debouncing the keyboard.

# The event loop

As shown in the diagram below, embedded systems typically operate in an endless loop, checking for events and running their handlers if the events have occurred.



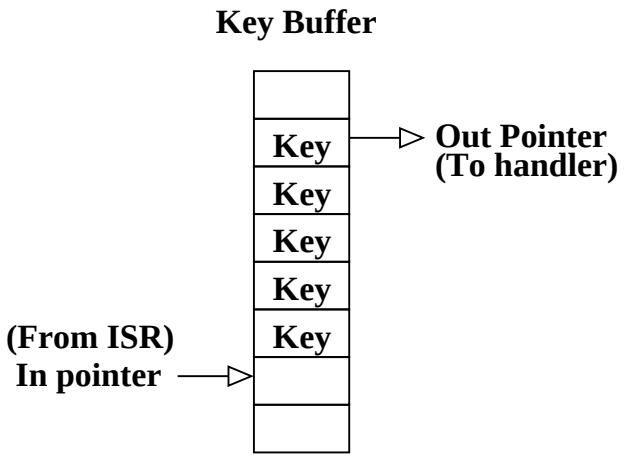



# The event loop

- Each event is associated with a data structure, which typically contains an event flag. If this flag is found to be set, the program runs its handler and clears the flag.
- But who sets this flag?
- This is typically done by an interrupt routine, which is triggered when the event occurs.
- The routine carries out whatever immediate action is required by the event, and then enters the data associated with it in the event data structure and sets the event flag.

# The event loop

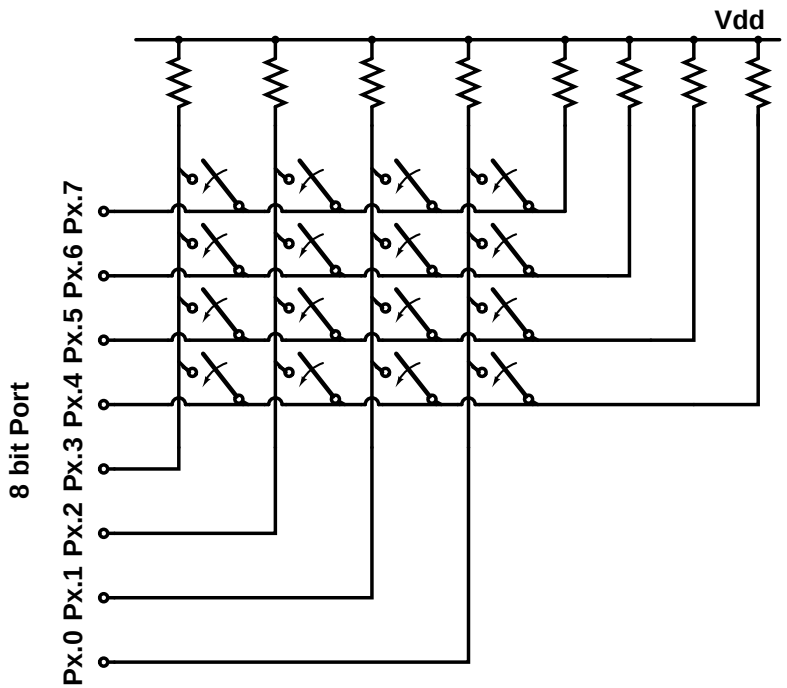
- The main software does not have to react to the event immediately as it occurs.
- For example, consider a keyboard event. When a key press is detected and confirmed, the key code will be entered in a buffer by the interrupt service routine.
- The main program can choose to react to the key buffer only when a complete command has been entered - and not on a key by key basis.



# A 4x4 Key board

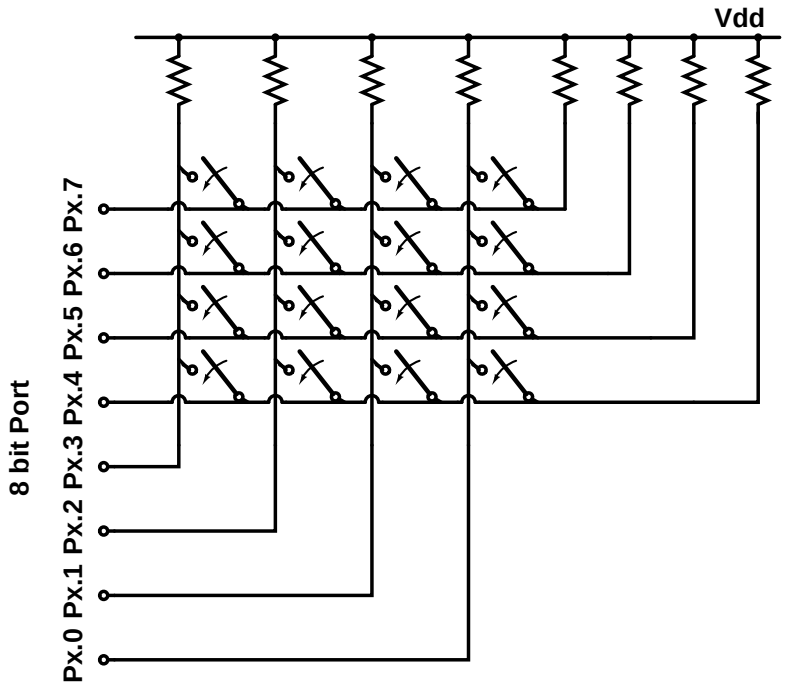
Consider a 4x4 keyboard connected to a port of a micro-controller as shown below:

- All port lines are pulled up to Vdd by resistors.
- The value of resistors is sufficiently high that these provide just a weak pull up.
- If a line to which a '1' is written is shorted with a line to which a '0' is written, the resultant value will a '0' because the pull up is weak.



# A 4x4 Key board

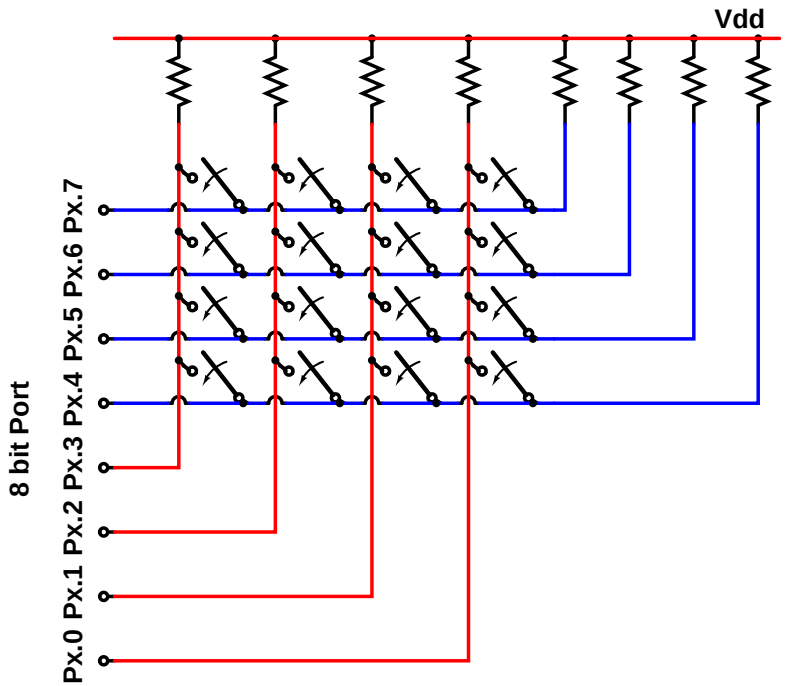
- Associated with each row-column combination is a key, which is just an inexpensive single pole switch.
- The switch will short its column to its row when pressed.



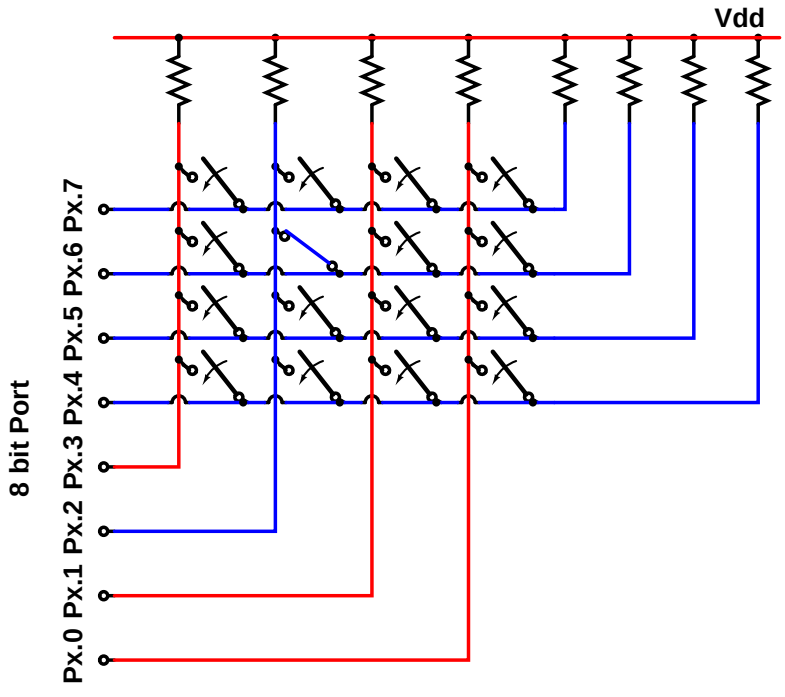
# Reading the Key board

- Assume that all column bits have been put in input mode by writing ‘1’s to them.
- We pull all row low by writing ’0’s to them.

(In the diagram here, red represents ‘1’, blue represents ‘0’.)



- If any key is pressed, it will short its row to its column.
- Rows are driven to ‘0’.
- So the corresponding column will be driven to ’0’, too.
- Thus, we can identify the column of the pressed key by reading the port.
- only the bit corresponding to this column will be ‘0’, others will be ’1’.



# Reading the Key board

So we have found the column number of the pressed key.

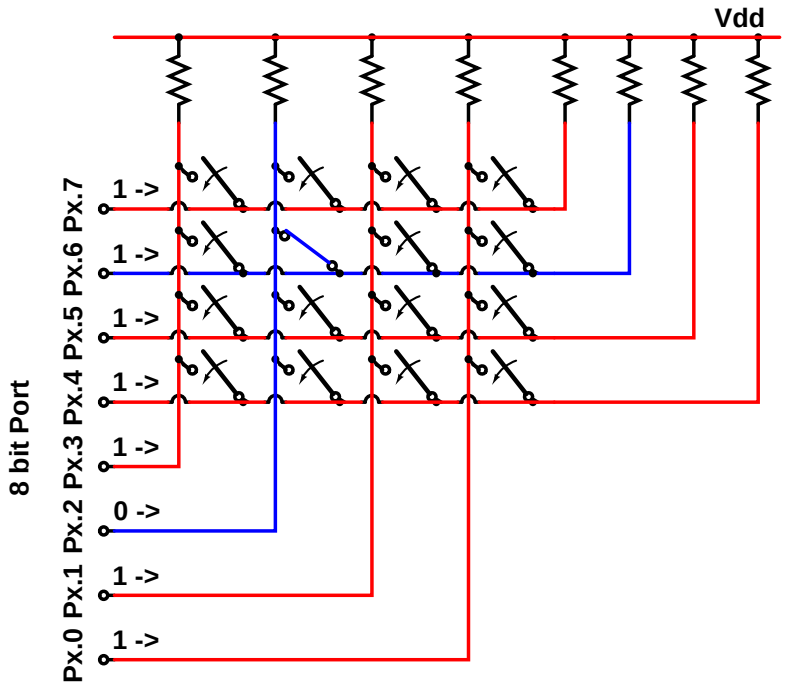
How do we find the row number?

- We could use a technique which is identical to column location,
- We write ‘0’ to all the columns, ‘1’s to all the rows.
- When we read the port, there will be a ‘0’ in the corresponding row.

In fact we can do better!

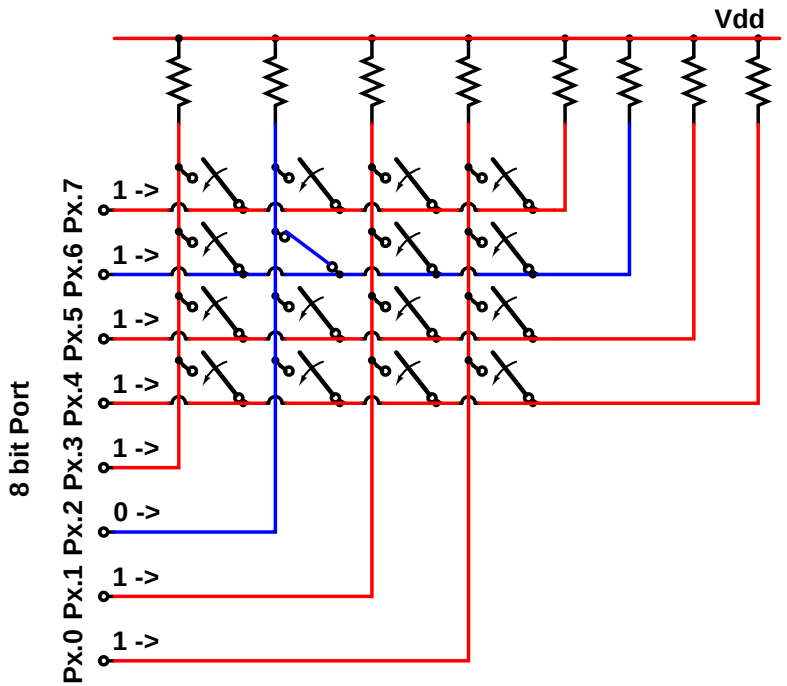
# Reading the Key board

- We write ‘0000’ to the row lines, and ‘1111’ to the column lines.
- We now read the port and examine the nibble corresponding to the columns.
- If any switch is closed, the corresponding bit in its column will be ‘0’.
- We now write back *this same nibble* to the columns, and ‘1’s to all rows.



# Reading the Key board

- Since the column in which the key is situated would have been read as ‘0’, it will now be *driven* to ‘0’.
- Because of the shorting provided by the switch, the corresponding row will also go to ‘0’.
- We can now identify the row by checking which row bit is ‘0’.



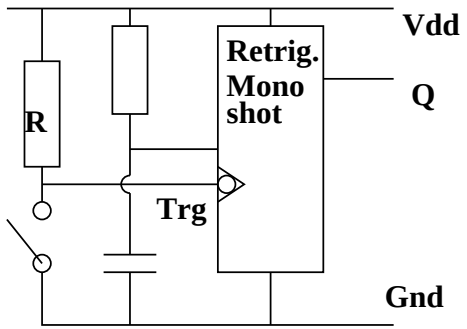
# Debouncing Keys

- When a key is pressed, contact is made and broken many times before the key settles to its ‘open’ or ‘shorted’ state.
- This is called ‘key bounce’.
- Obviously, wrong values will be read if we read the keyboard while the keys are still bouncing.
- Techniques for getting rid of this problem are known as key debouncing.

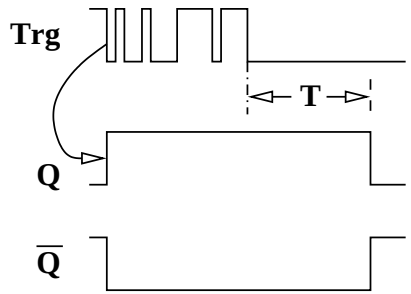
Keys may be debounced using hardware or software.

# Hardware debouncing techniques

Debouncing in hardware may use re-triggerable monoshots or other devices. Assume that the contact provides the trigger to a re-triggerable monoshot with a time constant T.




- Assume that T is much larger than the duration of key bouncing.
- When the first contact is made, it triggers the monoshot whose output gets set.
- Subsequent contacts will just retrigger the monoshot, so the output will remain set.
- After the last key bounce, the output will return to zero after a delay T.




Thus, while the input is bouncing, the output remains steady.

# Hardware debouncing techniques

- The monoshot method requires a re-triggerable monoshot for each key.
- There are other hardware techniques – such as low pass filtering.
- All of these add to the complexity of the circuit and to its cost.
- Software technique for key debouncing may be more competitive.

# Key debouncing in software

- We assume that any change of state of a key which occurs faster than (say) about 20 milliseconds is due to bouncing and must be rejected.
- We read the keyboard twice with an interval of 10 milliseconds.
- We consider a key pressed or released only if we find it in the same state on two consecutive readings.

In between, we would like to make the micro-controller free to perform its normal functions.

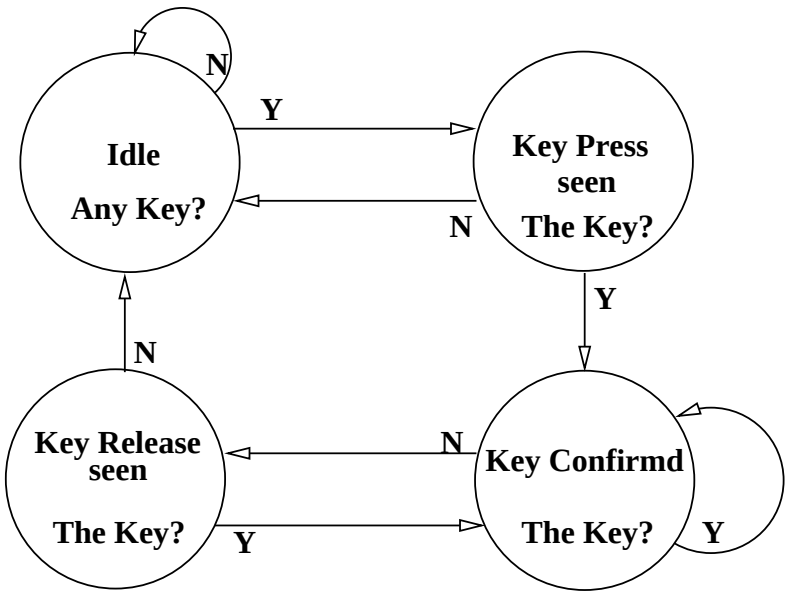
# Key debouncing in software

To keep the microcontroller free to perform its functions between timer interrupts,

- We need to keep a record of the previous state of a key in order to compare if it has changed.
- This can be done systematically using a state diagram approach.
- We would like to set up the state diagram in such a way that only when a debounced key press is detected, the key code is entered in the key buffer.
- All details of the key debouncing mechanism are localized to the interrupt service routine and the main routine is only aware of the key buffer.

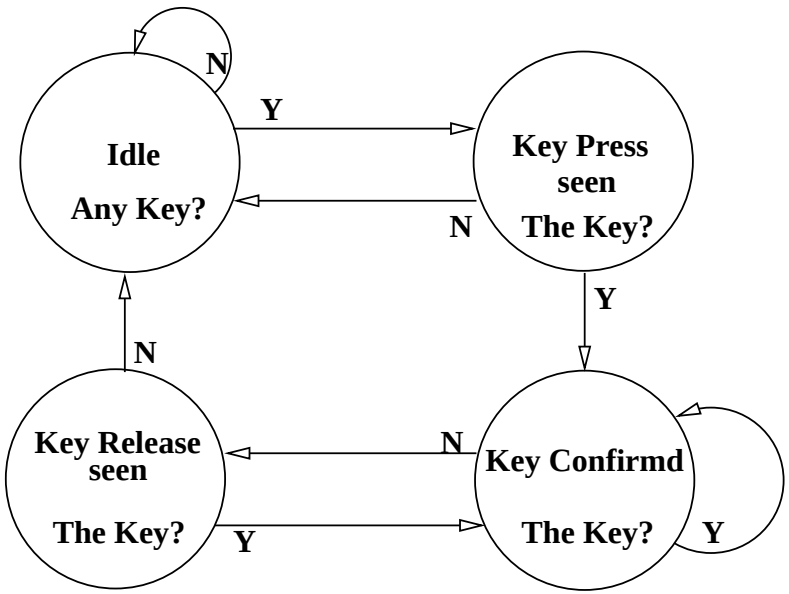
# State Diagram for Key de-bouncing

- Our simple system has four states: Idle, Key Press seen, Key Press confirmed and Key release seen.
- In the Idle state, there is no previous history and we perform the test: “Is any key found pressed?”
- If so, the key which is pressed is identified and stored as “the key”.



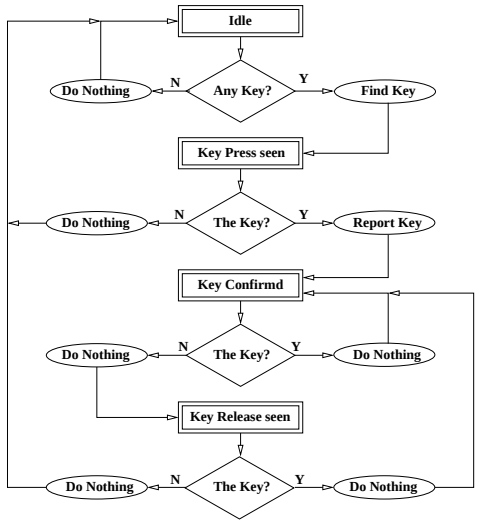
# State Diagram for Key de-bouncing

- In other states, we perform the test: Is “the key” found pressed?
- The system transitions through various states as shown on the right.
- The interrupt service routine returns as soon as there is a transition to the next state
- The next state could be the same as the current state.



# Flow Diagram

The actual flow of state transitions can be visualised better by the following diagram.




- Here rectangles represent states, diamonds represent tests and ovals represent action.
- Whenever no action is required, we have put down a special kind of action called “do nothing”.

# Constraints for Software Implementation

For a clean implementation of the state diagram, we place the following constraints:

1. Every state is associated with a unique test. As soon as we know what state we are in, we know which test to perform.
2. Each test returns just a ‘Yes’ or a ‘No’ answer.
3. Depending on the answer from the test, a selected action is performed and a transition is made to the next state. If no action is required, we invent an action called ‘do nothing’!
4. The interrupt service routine returns after the transition to the next state is made.

# Labeling

To actually implement the system, we use the follwing labeling:

| State | Label |
|---|---|
| Idle | 0 |
| Key Press Seen | 1 |
| Key Confirmed | 2 |
| Key Release Seen | 3 |

| Test | Label |
|---|---|
| Any key pressed? | 0 |
| 'The' key pressed | 1 |

| Action | Label |
|---|---|
| Do nothing | 0 |
| Find Key | 1 |
| Report Key | 2 |

# State Table

Now we can represent the state diagram by the following table:

| State | Test | Yes Case | | No Case | |
|---|---|---|---|---|---|
| | | Action | Next St. | Action | Next St. |
| 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 2 | 2 | 0 | 0 |
| 2 | 1 | 0 | 2 | 0 | 3 |
| 3 | 1 | 0 | 2 | 0 | 0 |

# Implementation

Each test and action is written as a subroutine.
We store the current state in a variable.
Addresses of test and action subroutines are stored in arrays.
When the timer interrupt occurs, we

1. Stop the timer, reload count, restart the timer.
2. Re-enable interrupts.
3. Using the current state as an index, find the test number for this state.
4. Using the test number as an index, look up the address of the test routine from the test address array.
5. Call the test routine.

# Implementation

When the test routine returns:

1. Check the return value. A yes answer can be stored as a set flag, while a no answer could be represented by a cleared flag.
2. depending on the return flag, load dptr with either the yes part of the transition table or the no part.
3. Look up the action number using the current state as an index.
4. Look up the address of the action routine from the action address array.
5. Call the selected action routine.
6. Look up the next state corresponding to the test answer and make current state = next state.

# Generic Implementation of an FSM

If we look at the above implementation, the dependence on the specific problem is confined

- to the test and action routines
- and to the data values in the various arrays and tables.

The logic of the program remains the same, whatever the state diagram.

# Generic Implementation of an FSM

Therefore we can develop a generic version of the “state diagram engine” and use it whenever a problem can be described by a state diagram.

All we need then are the data structures and test and action subroutines specific to the problem, which can be linked with the generic “state diagram engine” to give a program for the new state diagram.