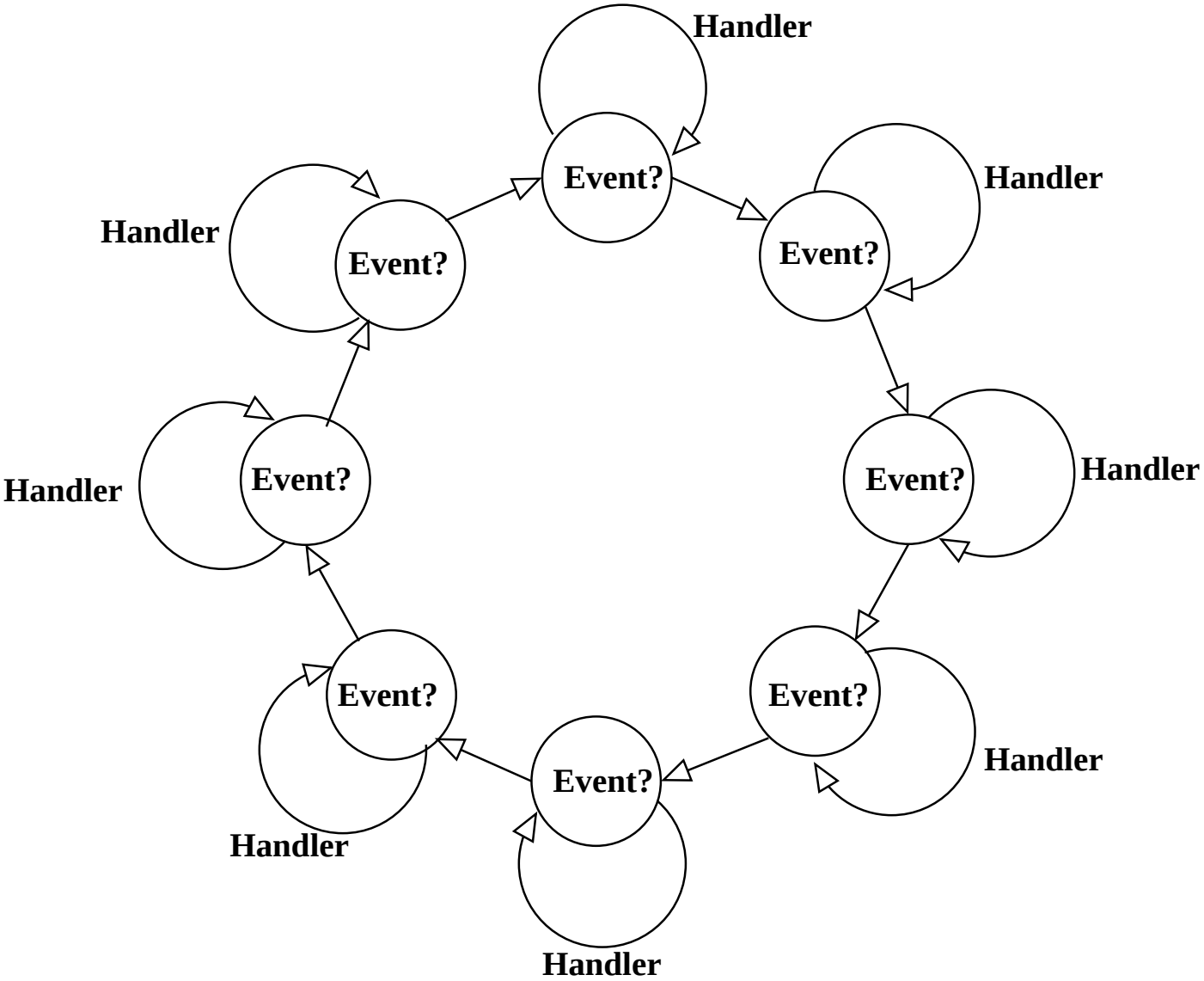
Keyboard Scanning using Finite State Machines

Dinesh Sharma
EE Department
IIT Bombay, Mumbai

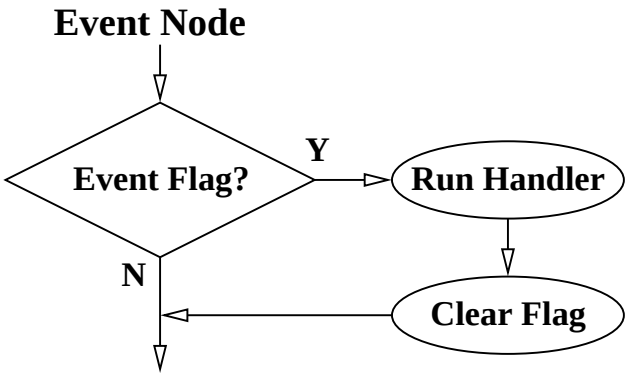
March 9, 2021

The event loop



Embedded systems typically operate in an endless loop, checking for events and running their handlers if the events have occurred.

The event loop

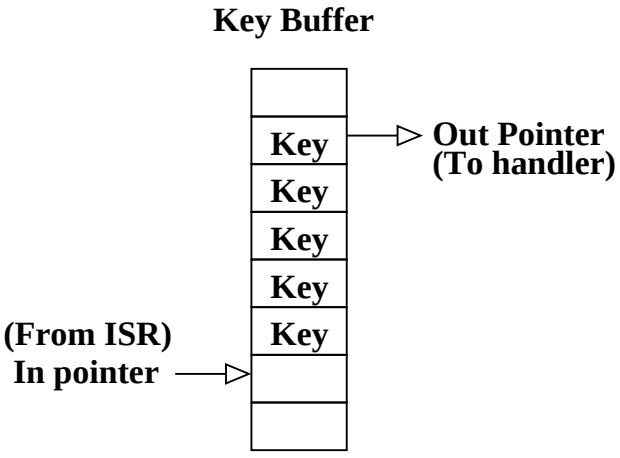


Each event is associated with a data structure, which typically contains an event flag. If this flag is found to be set, the program runs its handler and clears the flag.

But who sets this flag?

- This is typically done by an interrupt service 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



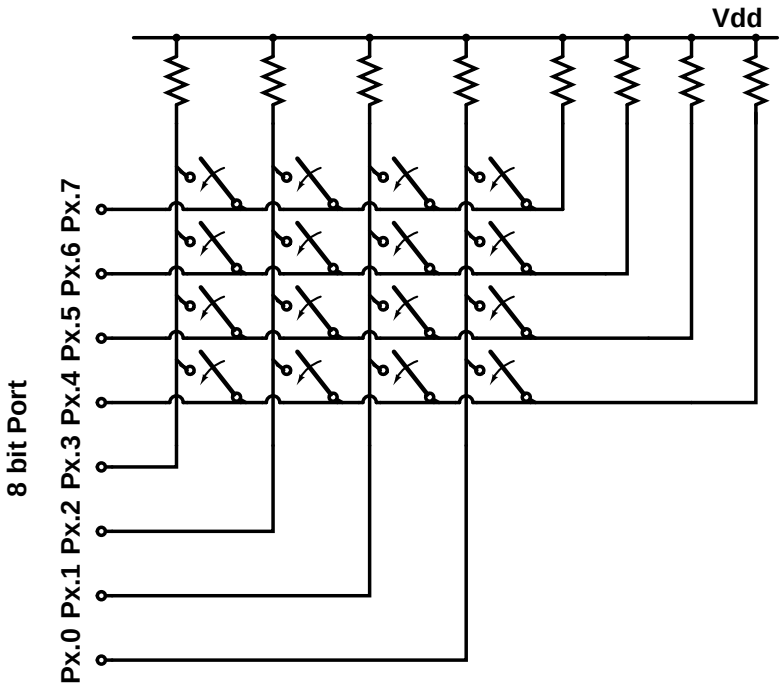
- The main software does not have to react to the event immediately as it occurs.
- For example, consider a keyboard event. When a key press is detected and confirmed, the key code will be entered in a buffer by the interrupt service routine.

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.

Thus key codes are collected in the buffer, till an end of line character is detected. Only then does the main program interpret the command.

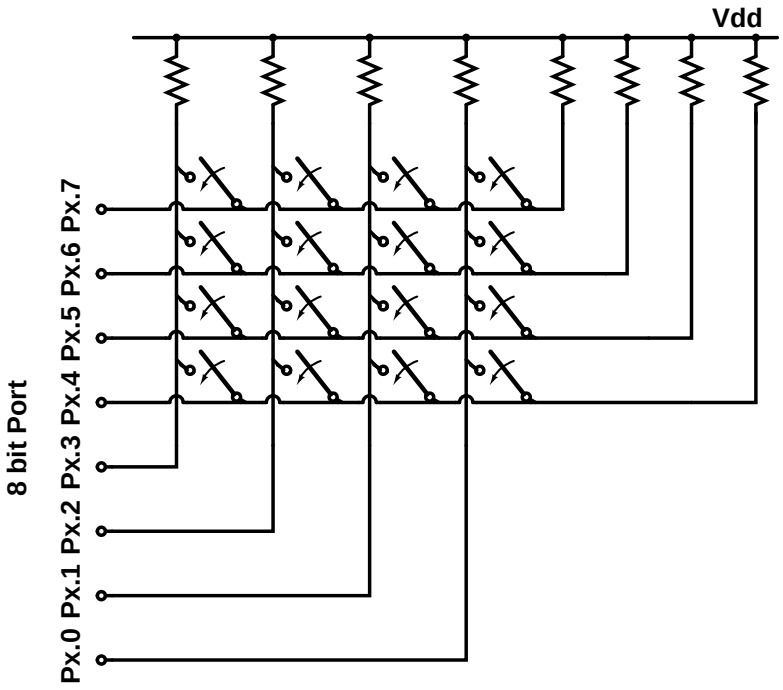
A 4x4 Key board

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 is to be used as an input, we write a ‘1’ to it, so the driver transistor is ‘OFF’ and only the weak pull up remains.

A 4x4 Key board

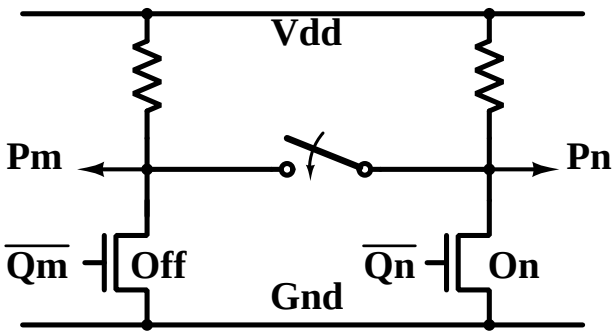


- Associated with each row-column combination is a key, which is just an inexpensive single pole switch.
- The switch will short its column to its row when pressed.
- If a key is pressed, we want to identify its row and column.

To detect switch closure, we need to analyse the effect of shorting two port lines.

Effect of Shorting

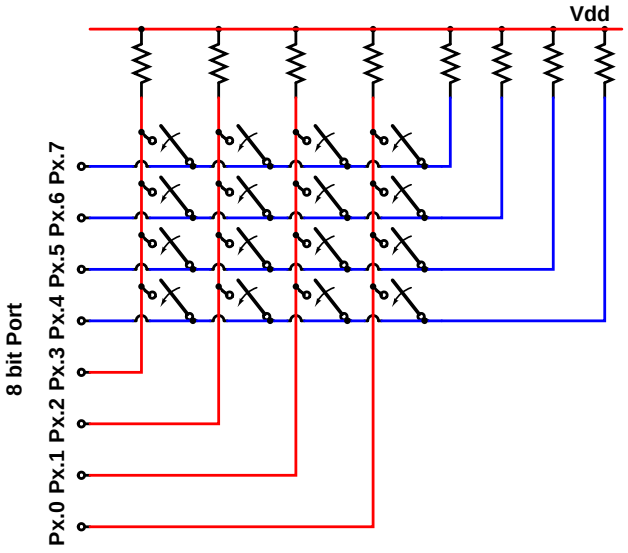
Consider the switch connected between port bit m (connected to a row) and port bit n (connected to a column).



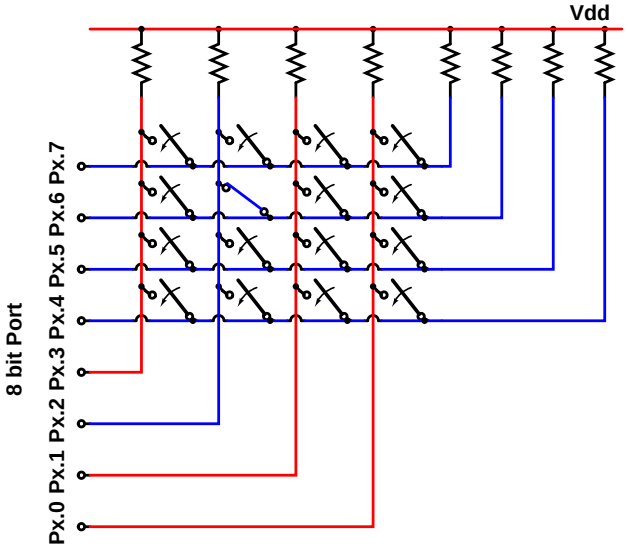
- If the switch is open, both port lines will be at their driven values. ($P_m = Q_m, P_n = Q_n$).
- If the switch is closed, both port lines will be '0' if either or both lines are driven to '0'.
- Thus a '0' overrides a '1'. Output will be '1' only when *both* port lines are driven to '1'. (Both n channel transistors are 'OFF').

In HDL terminology, we have a multiply driven node with a strong '0' and a weak '1'.

Reading the Key board



Assume that all column bits have been put in input mode by writing ‘1’s to them and all row lines are driven to ‘0’.
(This can be done by writing 0F to the port)



- If a switch in a particular row is closed, the corresponding column line will go to ‘0’ while all other columns will remain at ’1’.
- Thus, we can identify the column of the pressed key by writing ‘0’s to *all* the rows, ‘1’s to *all* the columns and reading the port.

Reading the Key board

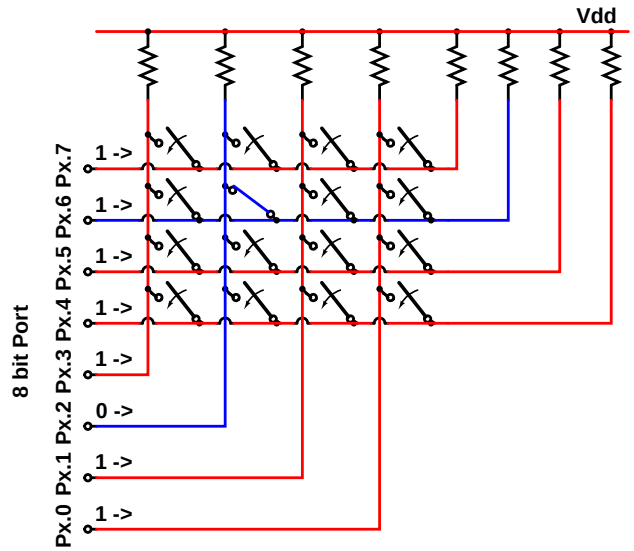
Having identified the column of the pressed key, how do we identify the row?

- We could just exchange rows with columns in the previous procedure.
- That is, write ‘0’ to all the columns, ‘1’s to all the rows and read the port to identify the row.

In fact we can do better.

Reading the Key board

- As before, write ‘0’ to all the rows, ‘1’s to all the columns (to make these bits inputs) and read the port, to identify the column.
- We now force the upper bits of *the read byte* to ‘1’ and write it back to the port.
- Only the column bit corresponding to the closed switch will be driven to zero.
  All row bits will be in input mode.
- If we now read the port, only the row and column bits corresponding to the pressed switch will be ‘0’.

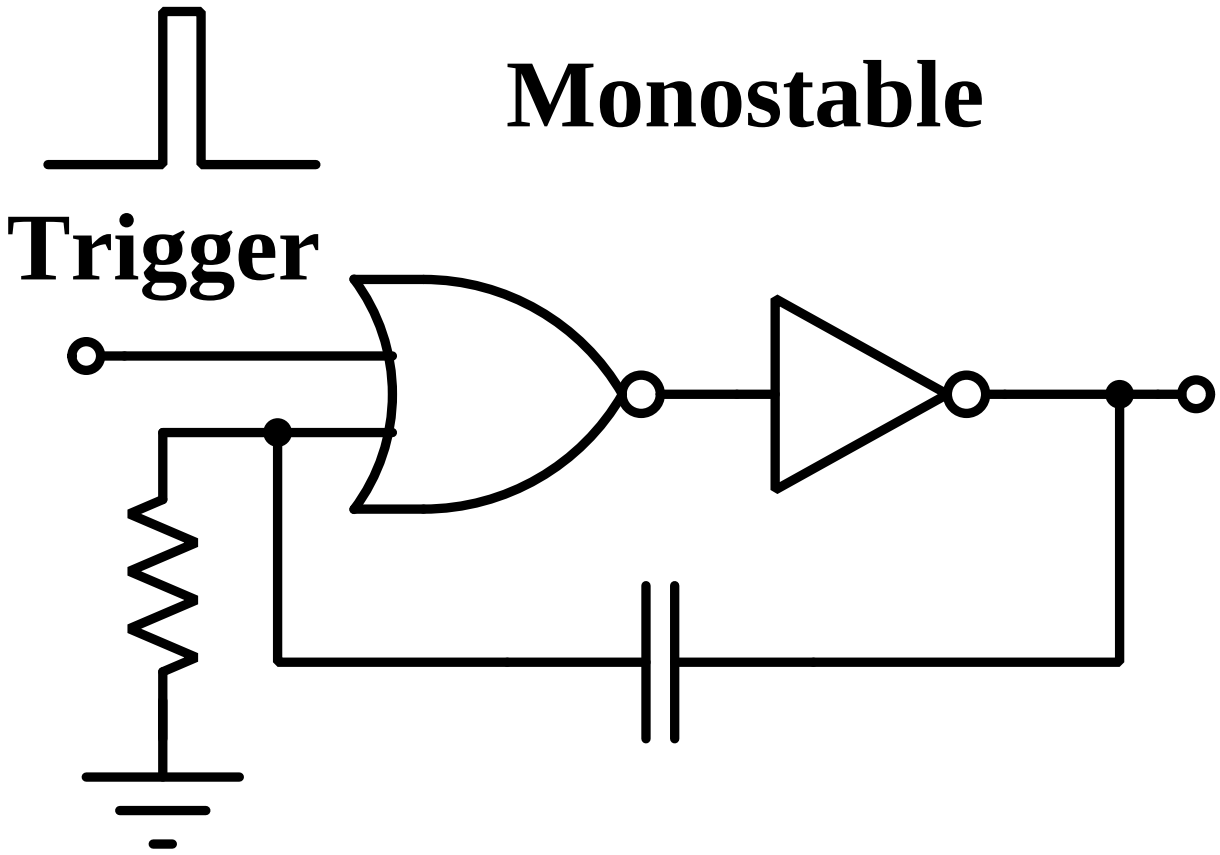


Debouncing the keys

- 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.

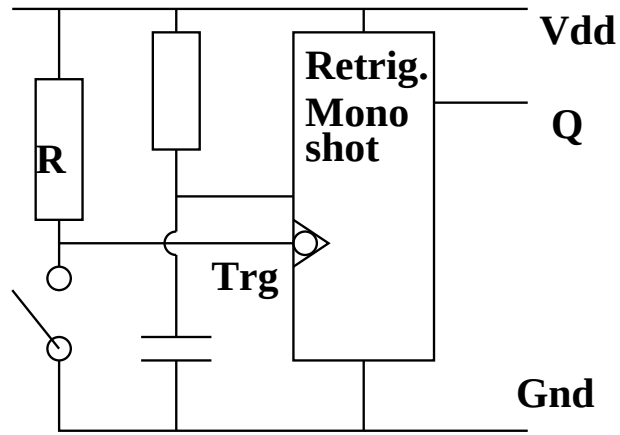
Keys may be debounced using hardware or software.

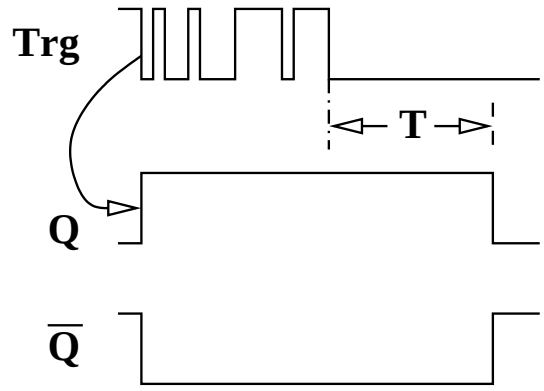
A mono-stable flipflop



Hardware Debouncing

- Assume that the contact provides the trigger to a re-triggerable mono-shot with a time constant T.
- T should be much larger than the duration of key bouncing.
- When the first contact is made, it triggers the mono-shot whose output gets set.
- Subsequent contacts will just re-trigger the mono-shot, 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

- This method requires a re-triggerable mono-shot for each key.
- There are other hardware techniques – such as low pass filtering.
- However, all of these add to the complexity of the circuit and to its cost.

Therefore it is worth exploring software techniques for key debouncing.

Software Debouncing

- We can use a software technique for key debouncing which uses a Finite State Machine approach.
- The strategy is similar to the mono-shot approach.
- We assume that any change of state of a key which is faster than (say) about 100 milliseconds is due to bouncing and must be rejected.
- Therefore, we read the keyboard twice with an interval of 50 milliseconds. We consider a key properly pressed or released only if we find it in the same state on two consecutive readings.

Software Debouncing using State Diagrams

- In the interval between successive reads of the keyboard, we would like to make the micro-controller free to perform its normal functions.
- Thus, 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.

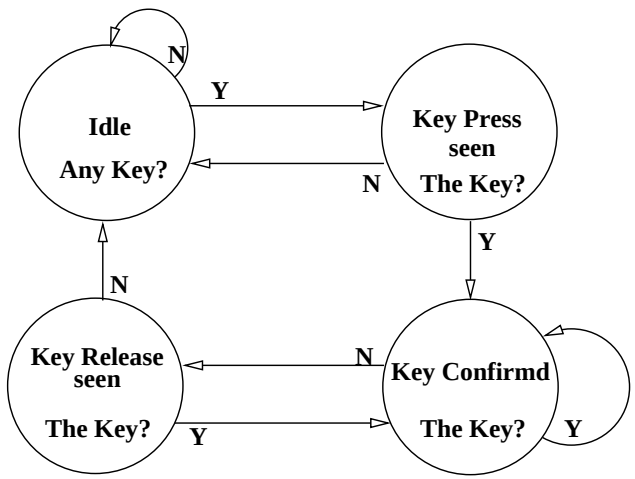
Software Debouncing using State Diagrams

- We program a timer to generate interrupts at 50 ms intervals.
- At each interrupt, we perform actions which will keep track of the state of the system, and then return from the interrupt to enable the processor to perform its usual functions.
- 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 should be localized to the interrupt service routine.
- The main routine interacts only with the key buffer.

A simple state diagram for debouncing

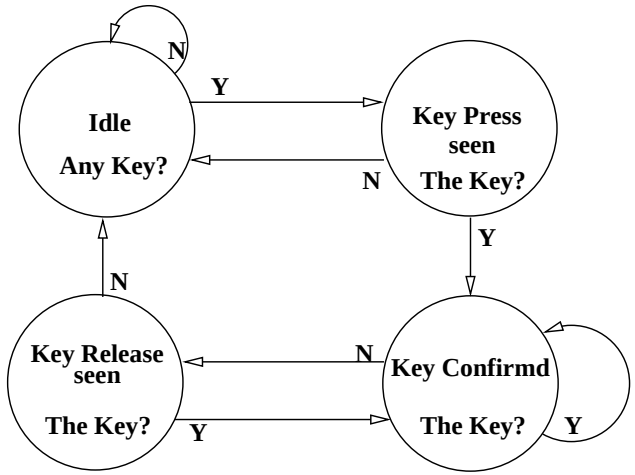
Our simple system has only four states: Idle, Key Press seen, Key Press confirmed and Key release seen.

- In the Idle state, there is no previous history and we check if *any* key is found pressed.
- If so, the key which is pressed is identified and stored as “the key”.
- In other states, we check if “the key” is found to be pressed.



A simple state diagram for debouncing

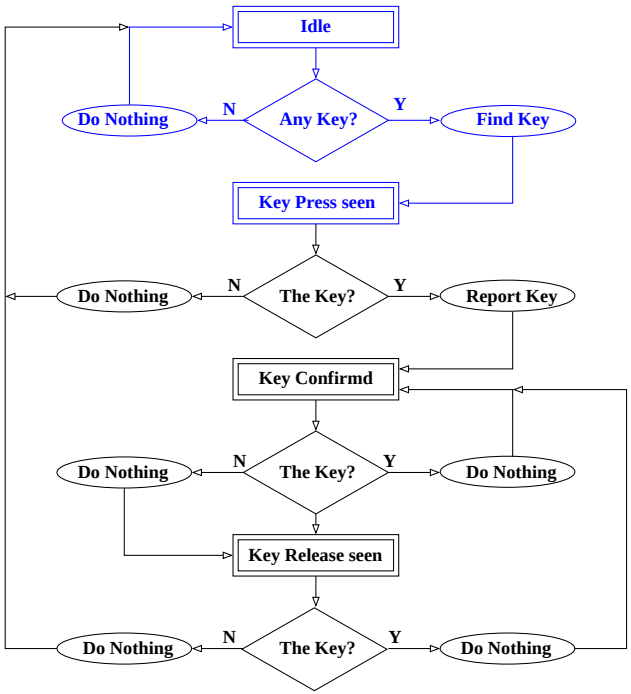
- The system transitions through various states as shown on the right.
- The interrupt service routine returns as soon as the transition to the next state (which could be the same as the current state) is made.



The actual flow of state transitions can be visualized better by a flow diagram.

A flow diagram for debouncing

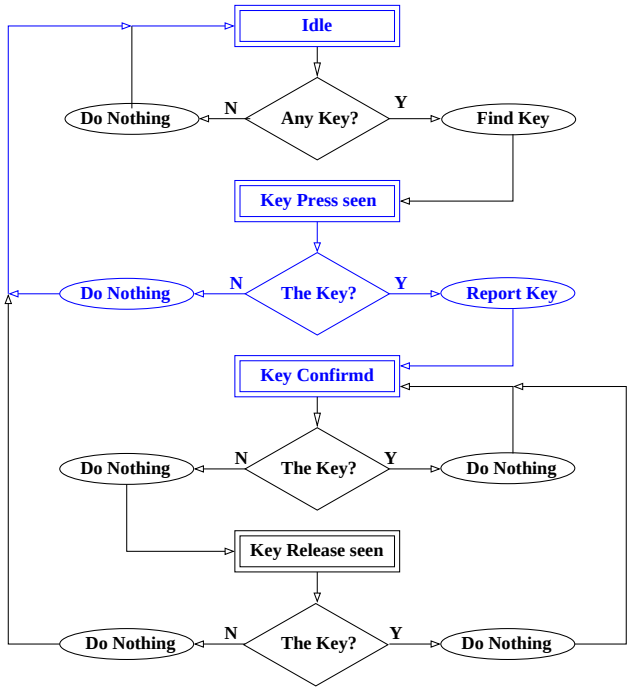
Here rectangles represent states, diamonds represent tests and ovals represent action.



- In the “Idle” state, we check if any key is pressed.
- If not, we stay in the “Idle” state.
- If a key is found pressed, we identify it and store its value as “the key”. and enter the state “Key Press Seen”

After entering the next state, we return from the interrupt and resume only after 50 ms.

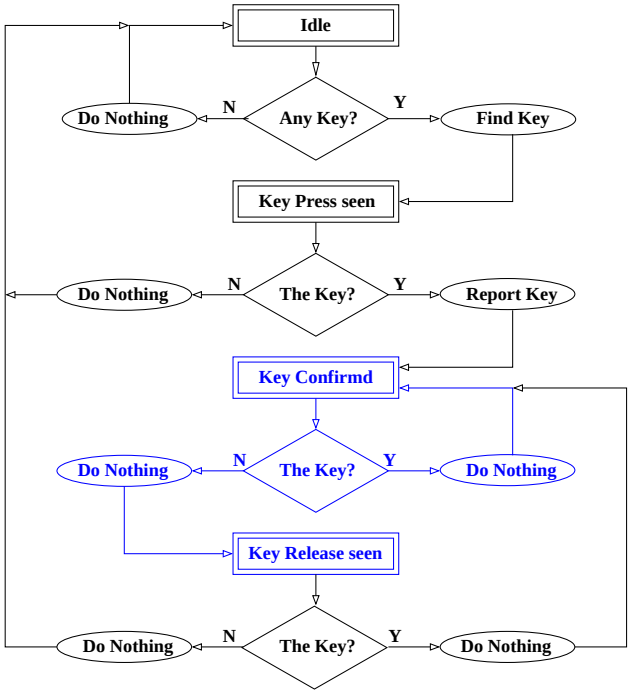
A flow diagram for debouncing



- In the “Key Press Seen State” we check if “the key” is still pressed.
- If not, we ignore the fact that we had seen this key pressed and return to the “Idle” state.
- If the key is still found pressed, we report it to the main program. This may involve entering its value in the key buffer, setting the status flag, etc.
- After this we enter the state “Key Confirmed”.

After entering the next state, we return from the interrupt and resume only after 50 ms.

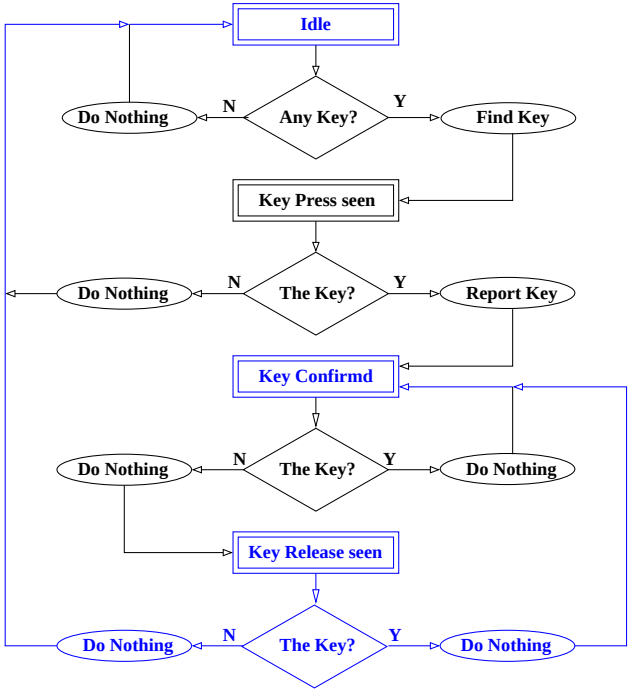
A flow diagram for debouncing



- In the “Key Confirmed” state, we are waiting for the confirmed key to be released.
- We check if “the key” is pressed.
- If so, we do nothing and remain in the “Key Confirmed” state.
- If not, we enter the state “Key Release Seen”.

After entering the next state, we return from the interrupt and resume only after 50 ms.

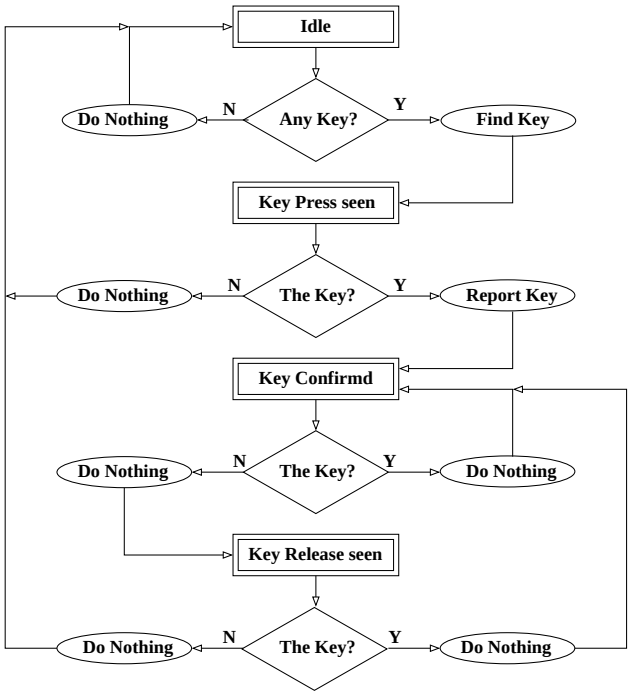
A flow diagram for debouncing



- In the “Key Release Seen” state, we want to confirm that the key release is stable.
- We check if “the key” is pressed.
- If so, we ignore the fact that we had seen the key release.
- Then we return to the “Key Confirmed” state.
- If not, the key release is stable.
- Processing of this key is complete and we return to the “Idle” state.

After entering the next state, we return from the interrupt and resume only after 50 ms.

Implementing the flow diagram



- For a clean implementation of the state diagram, Whenever no action is required, we have put down a special kind of action called “do nothing”.
- Now we always follow the sequence: state $\rightarrow$ test $\rightarrow$ action $\rightarrow$ state $\rightarrow$ return.
- This sequence is general and will implement **any** state diagram - not just this one.

So we write code for a generic FSM implementation and use a data structure to store the specifics of the FSM.

Implementing a generic state diagram

For a generic implementation of any 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.
4. The interrupt service routine returns after the transition to the next state is made.
5. All tests and actions are implemented as subroutines.

All details specific to a particular state diagram are now confined to test and action subroutines and various data structures.

Numerical Labeling for Keyboard Scan

States	
State	Label
Idle	0
Key Press Seen	1
Key Confirmed	2
Key Release Seen	3
Tests	
Test	Label
Any key pressed?	0
'The' key pressed?	1
Actions	
Action	Label
Do nothing	0
Find Key	1
Report Key	2

The current state, and next state are stored in variables. Values of "The Key", test number and action number are also stored in variables. We also define the following Jump Tables:

JmpTest:	
0	AJMP AnyKey
1	AJMP TheKey

JmpAct:	
0	AJMP DoNothing
1	AJMP FindKey
2	AJMP ReportKey

Implementing the keyboard state diagram

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

State	Test
0	0
1	1
2	1
3	1

St.	Yes Case	
	Action	Next St.
0	1	1
1	2	2
2	0	2
3	0	2

St.	No Case	
	Action	Next St.
0	0	0
1	0	0
2	0	3
3	0	0

States: 0:Idle 1:Press 2:Conf. 3:Release
Tests: 0:AnyKey 1:TheKey
Actions: 0:Nothing 1:FindKey 2:ReportKey

Generic Implementation of an FSM

When the timer interrupt occurs, we

- Save all registers to be used and PSW on the stack.
- Stop the timer, reload count, restart the timer.
- Re-enable interrupts.
- Using the current state as an index, find the test number for this state. (Load test tab address into DPTR, current state into A. The test number can now be obtained through MOVC A, @A+dptr).
- A now contains the test number.

Generic Implementation of an FSM

- A contains the test number.
- Save DPH, DPL on stack.
- Move JmpTest to DPTR.
- Call a subroutine which can jump to the numbered test subroutine using the test jump table.
- On return, restore DPL,DPH from the stack.

Generic Implementation of an FSM

- Check the return value from the test. A yes answer can be stored as a set flag, while a no answer could be represented by a cleared flag.
- depending on the return flag, load DPTR with the address of either the yes part of the transition table or the no part.
- Find the offset from DPTR of the action number based on the current state.
- Look up the action number from the loaded array.

Generic Implementation of an FSM

- A contains the action number.
- Save DPH, DPL on stack.
- Move JmpAct to DPTR.
- Call a subroutine which can jump to the numbered test subroutine using the action jump table.
- On return, restore DPL,DPH from the stack.

Generic Implementation of an FSM

- Using the current state as an index, look up the next state.
- Make the current state = next state.
- Restore all registers saved on stack when the timer interrupt occurred.
- Return from interrupt.

And We Are All Done!

Really!!

An introduction to Finite State Machines

8051 Programming Techniques

- Digital circuits whose output depends only on the current inputs are called combinational circuits.
- However, we can have circuits whose output depends on the sequence of previous inputs. Such circuits are called sequential digital circuits.
- Obviously, we cannot go back in history indefinitely – so practically realisable circuits have their outputs dependent on a finite sequence of inputs.
- Such circuits are called Finite State Machines.
- A finite state machine produces outputs which are functions of the ‘history’ of inputs.

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 subsequent 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.

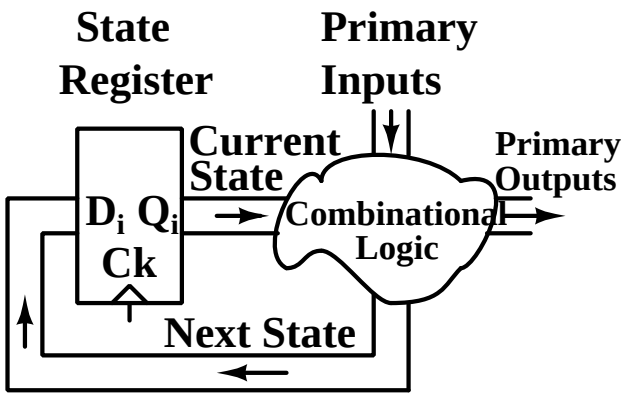
- The subsequent behaviour of this machine is identical when it has received two 1 Rupee coins in the past or one 2 Rupee coin.
- Thus, even though the detailed history in the two cases is different, we can say that the "state" of the system is the same.
- We compute the output and the "next state" as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

Hardware Implementation of FSMs

Flipflops store the current state in a state register.



- Next State is generated as a combinational function of current state and primary inputs. This Next State is presented to D inputs of the state register.
- Primary outputs are also produced using a combinational function of the current state and primary inputs.

States are updated at every clock edge in a synchronous FSM.

States are updated whenever there is event on any of the primary inputs in case of an asynchronous FSM.

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.
- FSMs can be implemented in software by using current state and next state as variables.
- Next state is computed from the current state and primary inputs.
- Primary outputs are produced from the current state, and optionally the current inputs.

We shall illustrate the use of a finite state implementation in software for debouncing a 4x4 keyboard.

State Diagram for Keyboard Debouncing

Looking up an entry in an array by index

- Assume that we have an array loaded in ROM using DB/DW directives.
- Assume that each entry in the array is m bytes in size.
- We want to get the i'th entry in this array.

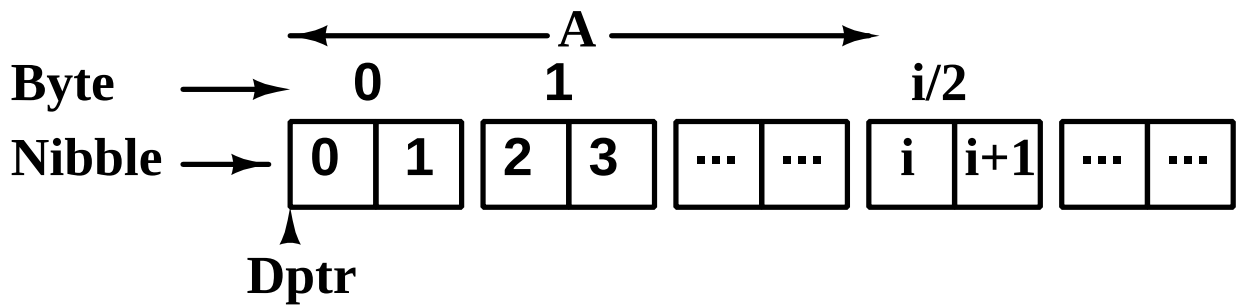
Move the start of the array into DPTR.
Offset from the start = i * m bytes. Compute i*m in A.

(For small m, additions may be more efficient compared to mult).
Save this value in B to compute the next offset easily.

Now, MOVC A, @A+DPTR will get the first byte of the entry.
Increment B, move B to A and repeat the above to get the next byte(s).

Looking up a nibble in an array by index

What if it is a packed array of nibbles? (2 nibbles/byte)



We want to get the i'th nibble from the array.

- Move the start of the array into DPTR.
- Byte Offset from the start = i/2. Compute it in A, using ROR.
  Carry will indicate even/odd nibble.
- Now, MOVC A, @A+DPTR will get the byte containing the nibble of interest.
- Assuming even nibbles in upper bits, odd nibbles in lower bits, Use SWAP A if even nibble is required (carry is zero). Zero out the other nibble by using ANL A, #0FH.

Use of jump/call tables

Suppose we want to call the i'th routine in an array of subroutines.

- AJMP instruction takes 2 bytes. It takes the top 5 bits from the updated PC and 11 lower bits from the destination label to construct the 16bit destination address.
- Therefore the AJMP instruction and the subroutine should lie in the same 2 KB address window.
- We construct a jump table by defining an array of AJMP instructions.

Use of jump tables

```
ORG ...
; Chosen to have last 11 bits=0
 JmpTab:   AJMP   sub1
           AJMP   sub2
           ...    ...

 sub1:     ...
           ...
           ...
 sub2:     ...
           ...
           ...
 ....:     ...
```

The calling program puts the index in A, the jump table address in DPTR and calls CallByNo.

CallByNo:
ADD A, A ; 2 Bytes per entry
JMP @A+DPTR

This will jump to the location containing the required AJMP instruction.
From there, it will go to subi
When Subi does a ret, it returns directly to the main caller.