Electronics 101:

Switches:

Consider a simple circuit :

Picture

When no power on 1,2 floats

⇨ We see noise when switch is open

When switch is open the input goes high

- When switch is closed we get a direct connect between V_{cc} & Gnd
  - Short, smoke
    - Too much current from V to ground

R1 limiits flow to a small amount

- A pull-up resistor

(Resistant) R = (Voltage) V = SV

LEDs:

Buttons:

- Fundamental form of input
- usually tied to general purpose I/O
  - sometimes on an ext interrupt
- we must sample the i/p to detect presses
  - is the button up or down?

- keypads are little more complex:

image

- A key press connect 2 wires

- pressing A connects I_3 & I_6

- We activate I_5 thru I_8 one at a time
- On each, we sample I_1 thru I_4
    - An active input indicates a connection => press

- how often do we need to sample?

    - (Constraint)how fast can a human press/release?

        - sampling 100x a second is more than enough

        - 10x a sec is common

            - we have to worry about microcontroller loading issues

                - spending too much cycles on waiting user input

- problems :

(a) when a button is pressed, the signal goes:

image

(b) this means we can't use a single sample to decide on state

Note: what if it's an interrupt?

(c) we also need to stop declaring a transition.

Hysteresis : Solution to (holding button vs quick clicking button multiple times):

- We accumulate state to decide when somethings is on or off
    - if we sample with on, increment
    - if we sample with off, decrement
- we then define on/off values and min/max values
    - avoid runaway counting when stable.

e.g. on=10 , off=4, max=12, min=0

- init to 0 & off.

image

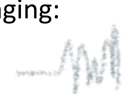
- we tune our settings to ensure that we handle noise
    - based on the h/w
        - experimentation

Note:

(1) We can use this to filter any noisy i/p(input)
    a. With buttons, we call this debouncing
(2) Easy to implement for multiple inputs with a table
    a. Stores our constants & accumulator
    b. Sampling simply iterates over the table

c. Easy to change and add inputs

Ringing:



Button issues:

- 1) do we response to a keypress or a keyrelease?

    - what do people expect vs what we can actually do

- 2) interrupts

    - if we have an interrupt that fires whenever we see a level, how do we actually deal with each interrupt?

        - stay in the ISR until release (against our design prinicle , simple)

        - disable that interrupt

            - when do we re-enable?

        Solution: trigger interrupt on <u>edges</u>

            - Check pin within ISR to determine state

- 3) what about key repeats?

    - on for x milliseconds, trigger another key press

- 4) delays between key presses

    - filtering hardware or software can incur a delay before triggering an on.

    Result:

        - Fast keypresses are ignored
        - But if we shorten the debounce, we will get false positive due to the fact that noise getting thru

- 4.5) lots of hardware filtering will give us occasional bounces

    - need to determine how you development board behave

- 5) multiple key presses

    - what if we want to detect:

        - 1) button "A" is pressed , do i

        - 2) button "B" is pressed, do j

        - 3) button "A" then "B" is pressed, do k

        - 4) button "A" and "B" are pressed, do l

    - we must measure how closely spaced the presses are:

        - time space to declare separate

- time space to declare consecutive
- time space to declare together
- say we're sampling 10times/s
- people won't actually press at exactly the same time.
    - we have to assume there is always a gap

Solution: need a state machine

Event processing

- debouncing is low-level state **** to give us clean on/off states
- buttons have semantic meaning that we must interpret (app specific)
- we use state machines to track input & trigger actions
- state machines

    States => record previous input chain and their order

    Transitions => define the next state based on the next input values

    Actions => use states & transition to generate ouputs

    - Can be in state or on transition

e.g.

image

Notes:

- our states have meaning
    - A,B,C don't cut it
- we don't have terminating states
    - run forever

- we don’t have a stream of characters

    - we can sample inputs and <u>not</u> transition

    - happened since nothing changed

- we don’t trigger an event to transition

    - we sample and see if we need to change

        - we are doing this on a heartbeat

    - with interrupts, ISR changes a value that our finite state machine code checks at next heartbeat.

- outputs of state machine : can be a sequence of operations

Real-time FSMs

- time adds another input to FSMs(state machine)

- we can transition based on amount of time passed

    e.g. timeout to return to start state

    e.g. timeout to another state if no input change

    e.g. multiple transition for the same input value input value based on how much time has passed while in current state

e.g.

image

- We use our heartbeat to timestamp when things happen

    0r

- Have a count in FSM for or baseline
    - Can be per state and/or per input

```
while(1)

{

        sleep_mode();

        sample_inputs();

        processFSM();

}
```

FSM Implementation:

a) Variable to track your current state
    a. Use an enum to define all possible states
        i. Enum X
        {
                S1,
                S2,
                S3,

NUM_STATES

}

- Array of states is easy [NUM_STATES]

b) Transitions

  a. Look-up table

    i. A row for each state & column for each input values

      1. Entries indicate next state

         – or –

      2. Use function pointers to call for each state

         a. Routine takes inputs & update state & generate output

```
while(1)
{
        State = table[state](……);
}
```

      3. Just use a big switch

         a. A case for each state

         b. Could call a routine

         c.