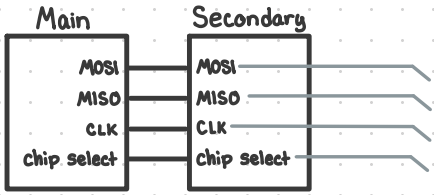
# LAB 2

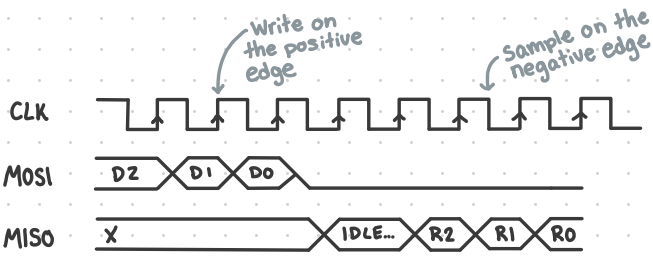
Kat + Joesph

# How SPI Works




MOSI main sends data to secondary (Main Out Secondary In)
MISO Secondary sends data to main (Main In Secondary Out)
Clock Shared clock signal from Main
Chip select main selects which secondary device to communicate with

MOSI and MISO sent bit by bit in serial (continuous stream, no packets)
- ↳ MOSI sent with MSB first
- ↳ MISO sent with LSB first

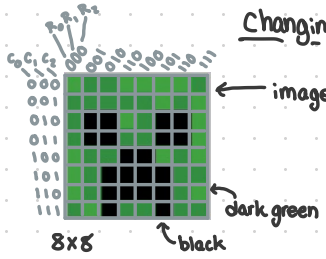



## SPI Transaction

1) Main outputs CLK
2) Chip select set low
3) Main sends data through MOSI, secondary reads it
4) Secondary sends data back through MISO (if needed)

# PART 2 Sending Serialized Data over SPI

## Changing the test pattern:




if green is 0 and black is 1 we can define the shape with gates

eyes $(c_1 \hat{} c_0) \,\&\, (\sim R_2 \,\&\, R_1)$

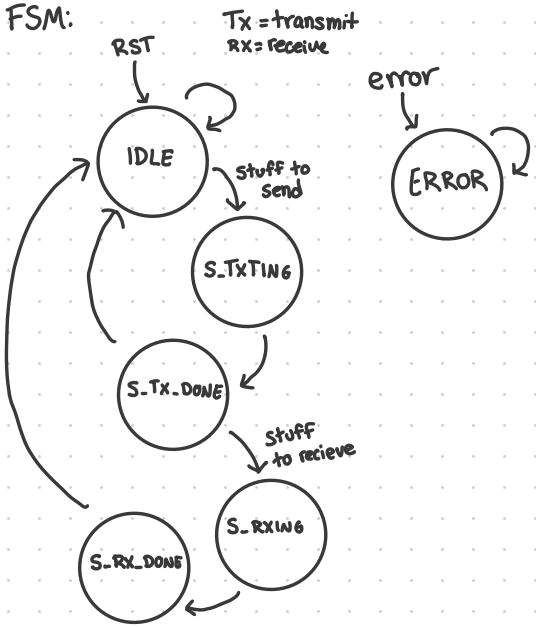
mouth $(c_1 \hat{} c_2 \,\&\, R_2) \,\&\, ((c_1 \hat{} c_0) \hat{} (R_1) \mid (R_1 \hat{} R_0))$

black cells = eyes | mouth

We know the screen into 16x16 blocks
So it can thus be split into 8x8 blocks

## Spi_controller

FSM:




Note:
SPI clock is twice as frequent as screen
↳ we can do pos edg behavior and neg edg behavior

oops we did more than we needed to

**IDLE State:**
reset o_valid, i_ready, o_data, Tx and Rx data
if i_valid set bitcount and state to TXING

IDLE ⟶ TXING

**TXING State:**
loop through bitcount
when end of the loop is reached
(bitcount == 0) State change

TXING ⟶ TXDONE

**TXDONE State**
if there's no stuff to read state → IDLE and i_R=1
if there's stuff to read state → RXING

TXDONE ⟶ IDLE
TXDONE ⟶ RXING

**RXING State**
Shift through input data (loop down through bitcount)

RXING ⟶ RXDONE

**RXDONE State**
Set i_R 0_v and 0_DATA
return to IDLE

RXDONE ⟶ IDLE

sclk →
csb →
mosi →
miso ←

# Learning from Professional Code

## ili94341 display controller

### Configuration FSM

There are 2 stored states: `cfg_state`, and `cfg_state_after_wait` (cfg_ omitted in explanation for lengthiness).

On reset the delay counter and ROM address are set to 0. `data_command_b`, which specifies between sending data and sending commands (1 for data, 0 for commands), is set to 1. Both `state` `state_after_wait` become `GET_DATA_SIZE`. In this next state the ROM address is increased by 1, `state_after_wait` becomes `GET_CMD` and `state` becomes `MEM_WAIT`. In the `MEM_WAIT` state `state` becomes `state_after_wait`. The state now transitions to the most recent state of `state_after_wait`, `GET_CMD`, where `state_after_wait` becomes `SEND_CMD` and `state` becomes `MEM_WAIT`. In `MEM_WAIT`, `state` is once again set to `state_after_wait`, which is now `SEND_CMD`. This elongated transition kills time since the memory has latency. Now in sending mode, for as long as the ROM data isn't 0, `state_after_wait` becomes `GET_DATA` and `state` becomes `SPI_WAIT`. This SPI waiting state depends on the delay counter (which counts down to 0) or the SPI transaction wire `i_ready`. When either `i_ready` is true or the countdown ends, the counter is reset, `data_commandb` is set to 1 (data), and `state` becomes `state_after_wait`, triggering a transition to the `GET_DATA` state. Now the ROM address is iterated through in increments of 1, each time (with more delays built in) switching to the `SEND_DATA` state. When there are no more bytes of data remaining (with another state-change delay built in) the `DONE` state is triggered and the Main FSM starts.

### Main FSM

Once again there are two stored states: `state` and `state_after_wait`.

The first state of this FSM is set by the end of the previous FSM, beginning at `TX_PIXEL_DATA_START`. `data_comandb` is set to 0 (command) and after transitioning temporarily to the `WAIT_FOR_SPI` state to wait for `i_ready` to be true, the state is advanced to `TX_PIXEL_DATA_START` where `data_commandb` is set to 1 (data) and after another run through the waiting state `state` becomes `INCREMENT_PIXEL`. Now each pixel is incremented through column by column, row by row, with state-change delays (repeating the previous few state changes) between each pixel to make sure `i_ready` is true every time. When every pixel has been taken

care of, `state` becomes `START_FRAME` where `data_commandb` is set to 0 (command), `state` is set to `WAIT_FOR_SPI` until `i_ready` is once again true, in which case the `START_FRAME` state begins the cycle anew.

# Ft6206 controller

## Main FSM

Once again there are two stored states: `state` and `state_after_wait`.

On reset the initial state is `INIT` where the state becomes `SET_THRESHOLD_REG`. Here `state_after_wait` becomes `SET_THRESHOLD_DATA` and `state` becomes `WAIT_FOR_I2C_WR` in which `state` becomes `state_after_wait` once `i_ready` is true. Once this is the case, `state` becomes `SET_THRESHOLD_DATA` where `state` becomes `IDLE` once waiting again for the I2C write. In the `IDLE` state, once `i_ready` and `ena` are true, the active register becomes `TD_STATUS` and the state transitions to `GET_REG_REG`. Now, after again waiting for I2C write, the state transitions to `GET_REG_DATA` which then takes a detour through `WAIT_FOR_I2C_RD` to make sure `i_ready` and `o_valid` are true before switching to `GET_REG_DONE`. In this state if `o_valid` is false it switches to `IDLE`. Otherwise the active register increments and some logic determines if there's a touch, two touches, or no touches, otherwise it reads data from memory. When this process ends `state` becomes S_TOUCH_DONE`where some final touches are made like fixing the orientation. Once this is finished,` state `returns to` IDLE` and the process repeats.

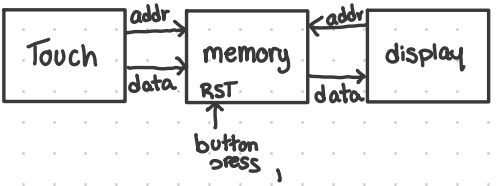
# PART 3 Interfacing with VRAM

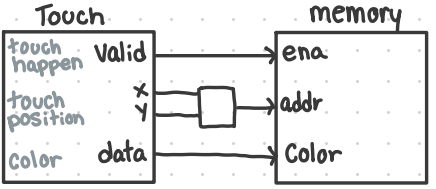
FSM that can

1) clear memory on button press

2) update memory based on touch values

3) emit draw signals based on memory

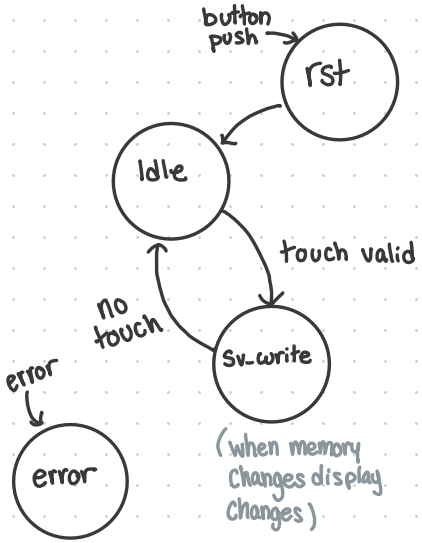



converting x and y to address

x and y 12 bits each, x=disp width
y=disp height

display = 320 x 240 pixles




RST
count down each pixel of the screen (like a loop) and reset every bit in vram to 0