Lecture 07
Driver Design

CE346 – Microprocessor System Design

Branden Ghena – Fall 2022

Some slides borrowed from:
Josiah Hester (Northwestern), Prabal Dutta (UC Berkeley)

Northwestern

Administriva

- Project Proposals due Thursday!
  - A few are in so far and they look great and I'm super excited!!!!!
  - My goal is get you feedback by early next week

- Otherwise, class keeps going as usual
  - Still have four more lab sessions
  - Still have three more quizzes
  - Lots more content to cover

Today's Goals

- Deep-dive into driver design options
- Explore another aspect of device driver design
  - Non-blocking vs Blocking interfaces
- Discuss how interrupts interact with these
  - Event-loop as a partial alternative
- Consider how an LED matrix driver could be constructed

Outline

- **Driver Interfaces (Blocking and Non-Blocking)**
- Event-driven Model
- Continuous Operation

How should we write driver software?

- There are various knobs available to us from hardware
  - Polling, Interrupts, DMA

- There are also various software interface design
  - Synchronous
  - Asynchronous
    - Callback
    - Event-driven model

Synchronous device drivers

- Synchronous functions
  - Function call issues a command
  - Does not return until action is complete and result is ready

- Example: most functions we're used to
  - `sqrt()` for example
  - `printf()` also usually works this way (with some exceptions)

- Arduino interfaces are usually like this!
  - Easy to get started with and understand

Downside of synchronous code: the waiting

- How long will it take until the function returns?
  - Immediately, seconds, minutes?
- What if there’s an error and the device never responds?
  - More advanced interface could include a timeout option
- Synchronous designs require other synchronous designs
  - We can build synchronous interfaces from asynchronous ones
  - But we can’t go the other way

Asynchronous drivers

- Goal: let the hardware run on its own and have the code get back to it later
- Challenge: programmers don’t think that way
- Other challenge: how do we “get back to it later”?
  - Callbacks
  - Event-driven model

Callbacks

- Callbacks reuse a similar idea to interrupts
  - When the event occurs, call this function

- General pattern
  - Call driver function with one argument being a function pointer
  - Driver sets up interaction and returns immediately
  - Later the event happens and the driver calls the function pointer

Function pointers in C

- Harder than in Javascript or C++. Can't define anonymous function inline
  - Instead create a pointer to an existing function in your code

```
void myfun(int a) {
    // do something here
}

void main() {
    void (*fun_ptr)(int) = &myfun;
    fun_ptr(10); // dereference happens automatically
}
```

& is actually unnecessary.
With or without are identical.

Callback functions

- ```
  uint32_t timer_start(
                 uint32_t microseconds,
                 void (*callback_fn)(void*),
                 void* context
                 );
  ```
- ```
  timer_start(duration, my_timer_handler, context);
  ```

- “Context” is often provided as well (void*)
  - Ability for caller to pass an argument for the callback function
  - Often a pointer to a position in a structure or a shared variable to modify

Callbacks usually run in an interrupt mode

- If the interrupt handler calls the callback, the callback will be within that same interrupt mode
- Be careful which variables you modify!!
  - Could lead to concurrency issues if you modify a public structure
- Starts to get pretty annoying
  - Embedded systems deal with concurrency issues just like OS

Building synchronous code out of callbacks

- Callback handlers can be used to build synchronous code

```
void myfun(void* context) {
    *(boolean*)context = true; // context is the flag pointer
}

void timer_start_blocking(duration) {
    volatile boolean flag = false;
    timer_start(duration, &myfun, &flag);
    while (!flag) { }
}
```

Temp driver example

nu-microbit-base/software/apps/temp_driver/

- Some necessary functions
  - NVIC_EnableIRQ(irq); // TEMP_IRQn is for the Temperature Sensor
  - NVIC_SetPriority(irq, priority)

Outline

- Driver Interfaces (Blocking and Non-Blocking)
- **Event-driven Model**
- Continuous Operation

Interrupts are frustrating

- We do not always want to block on every call
- We also do not want to deal with concurrency issues

- An alternative: one main event loop
  - Polls necessary sensors
  - Iterates through state machine and determine actions
  - Runs at a certain frequency

Event loop

- Rather than polling a single driver, poll all of them
  - Each time through the loop check all relevant inputs
  - Respond to events that are necessary
  - Sleep until ready to start again

```
while (1) {
    time start = get_time();
    boolean result = check_timer();
    if (result) { check_gps(); }
    adjust_throttle();
    delay_ms(1000 - (get_time() - start));
}
```

Downsides of event loop design

- Timeliness can be a problem

- How long between the timer being ready and the GPS being checked in this example?
  - Maximum of 1 second plus the time spent checking other stuff

```
while (1) {
    time start = get_time();
    boolean result = check_timer();
    if (result) { check_gps(); }
    adjust_throttle();
    delay_ms(1000 - (get_time() - start));
}
```

Top-half / Bottom-half handler design

- Top half
  - Interrupt handler
    - Immediately continues next transaction
    - Or signals for top half to continue (often with shared variable)

- Bottom half
  - Performs logic to actually process and respond to the event
  - Run in a non-interrupt context when the scheduler is ready for it
    - Usually safe to run it even while interrupts could be occurring

Temperature event-loop example

nu-microbit-base/software/apps/temp_event_loop/

- Some necessary functions
  - NVIC_EnableIRQ(irq); // TEMP_IRQn is for the Temperature Sensor
  - NVIC_SetPriority(irq, priority)

Outline

- Driver Interfaces (Blocking and Non-Blocking)
- Event-driven Model
- **Continuous Operation**

Continuous operation

- For some sensors/actuators they might be continuous updating in the background

- For those, we only need one init_and_start function and a read function
  - Continuous sensors are always ready with the most recent sample

  - Continuous actuators will always update to the new command as soon as possible
    - They might skip a command if you give it multiple very quickly

Continuously updating temperature

- Temperature driver design
  1. In the interrupt handler, copy over the value
  2. Start the next event, which will automatically re-trigger the interrupt

  - No more is_ready() function, data is always ready with the most up-to-date value

- This would mean a TON of interrupts
  - Probably want to combine with a timer to run it more slowly

LED Matrix design

- This is a good example of a continuous operation actuator

- General driver design
  - Split operation between a Model and a View (Model-View-Controller design)

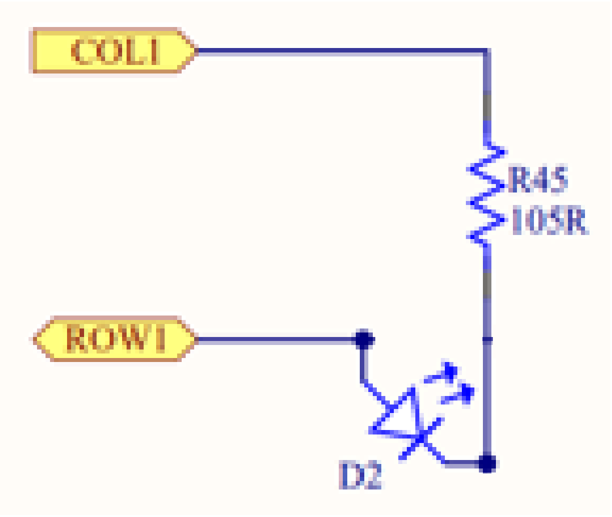
  - Model contains what you want the state of the LEDs to be
    - Only updates when the user calls a function
    - Updates immediately (non-blocking)

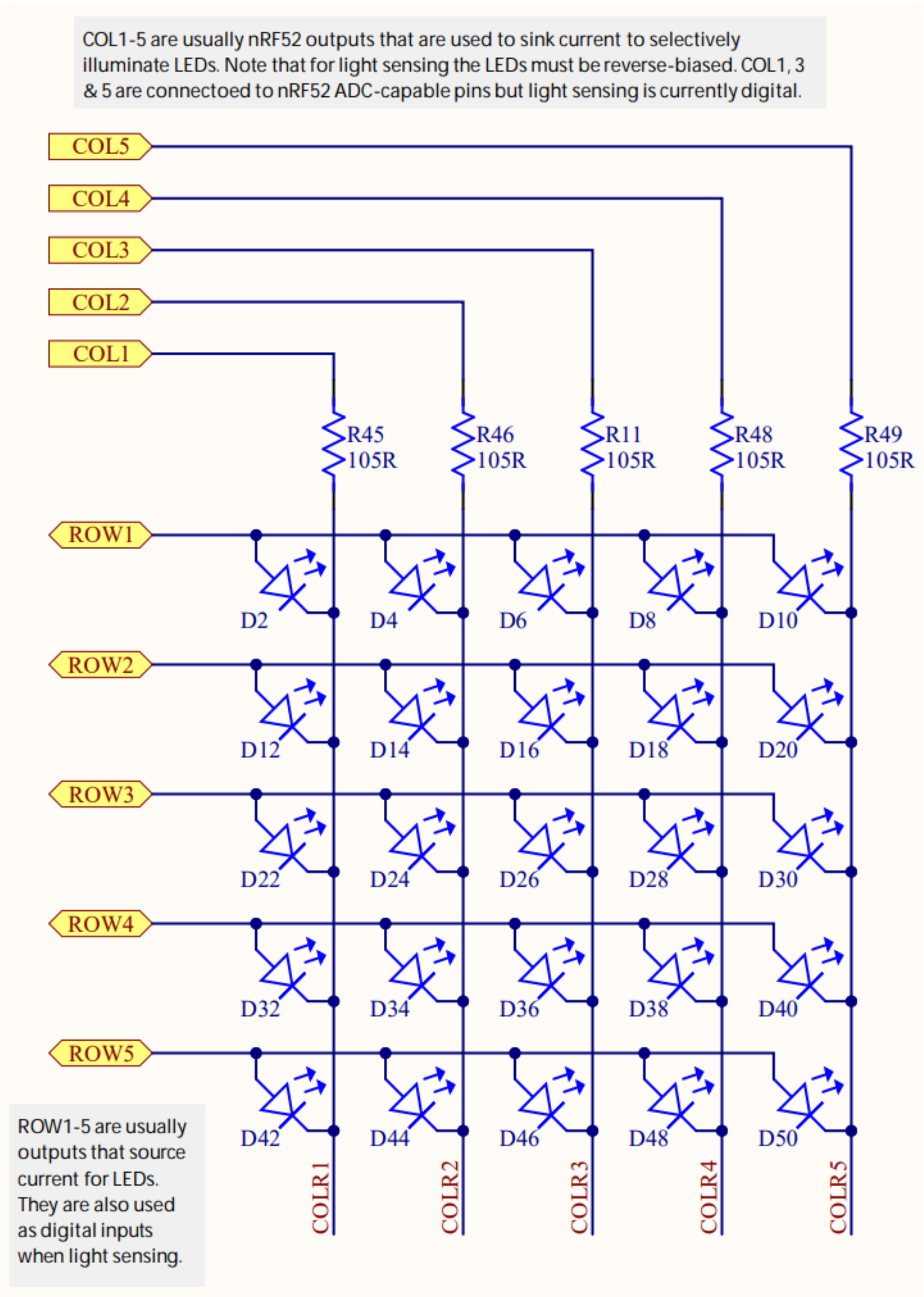
  - View contains the code to take the model and display it on the LEDs
    - Continuously updates the LED states with a timer

LEDs on the Microbit

- Use two GPIO pins to control each LED
  - Row high as VDD
  - Column low as Ground

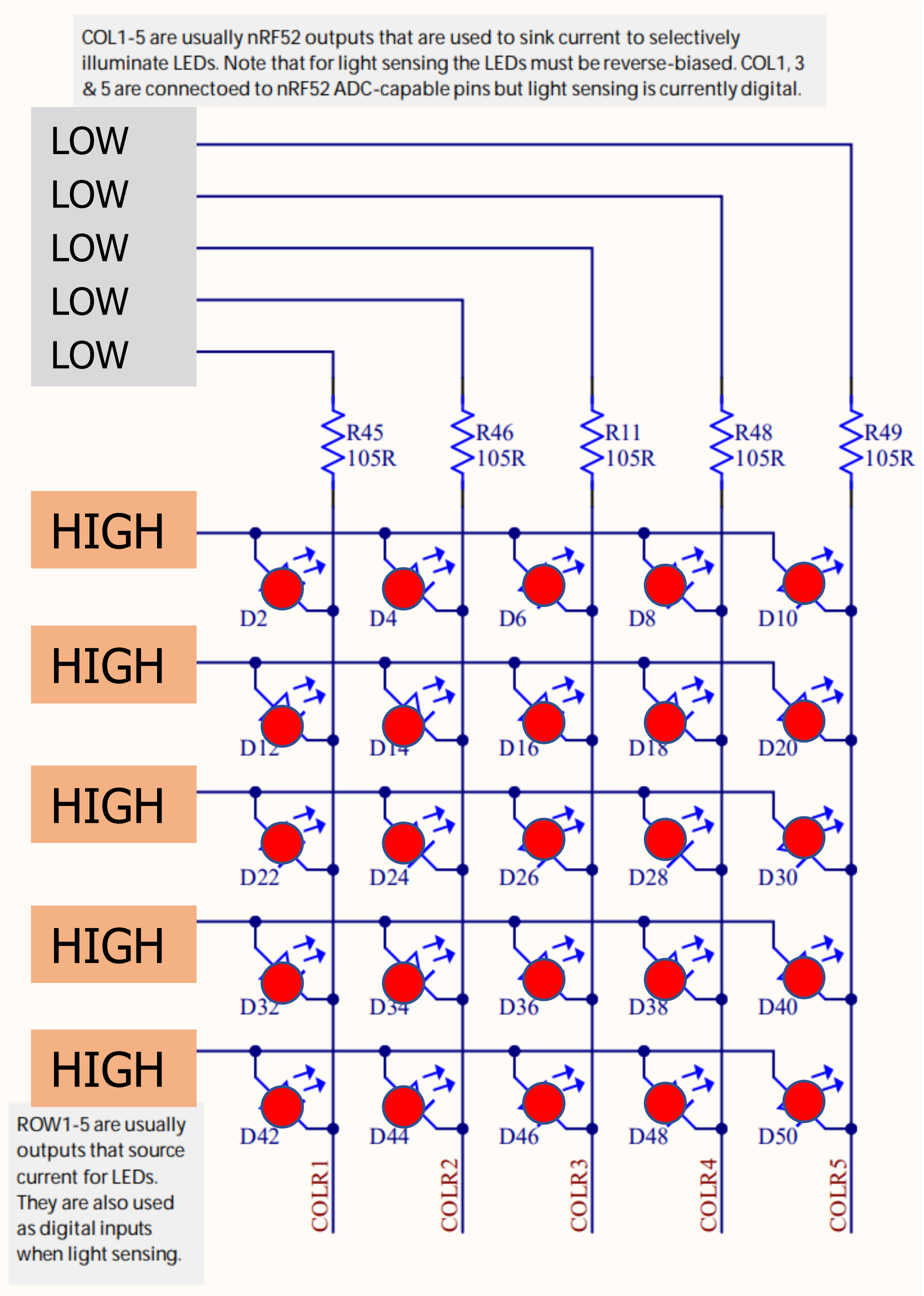
- Remember, connections only exist where there are dots





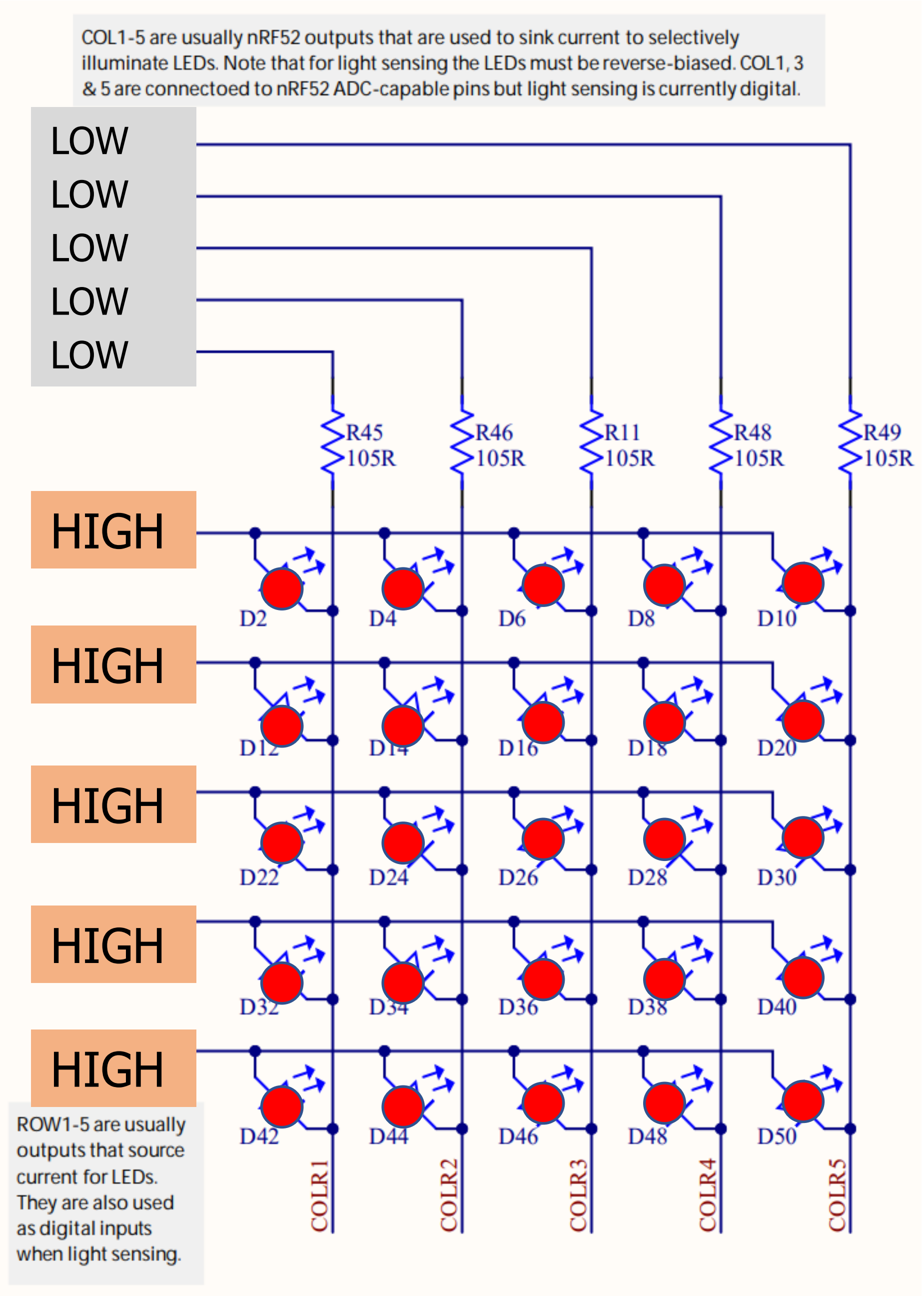
Controlling the LED matrix

- We can light up all the LEDs at once:
  - Set all rows to High
  - Clear all columns to Low



Controlling the LED matrix

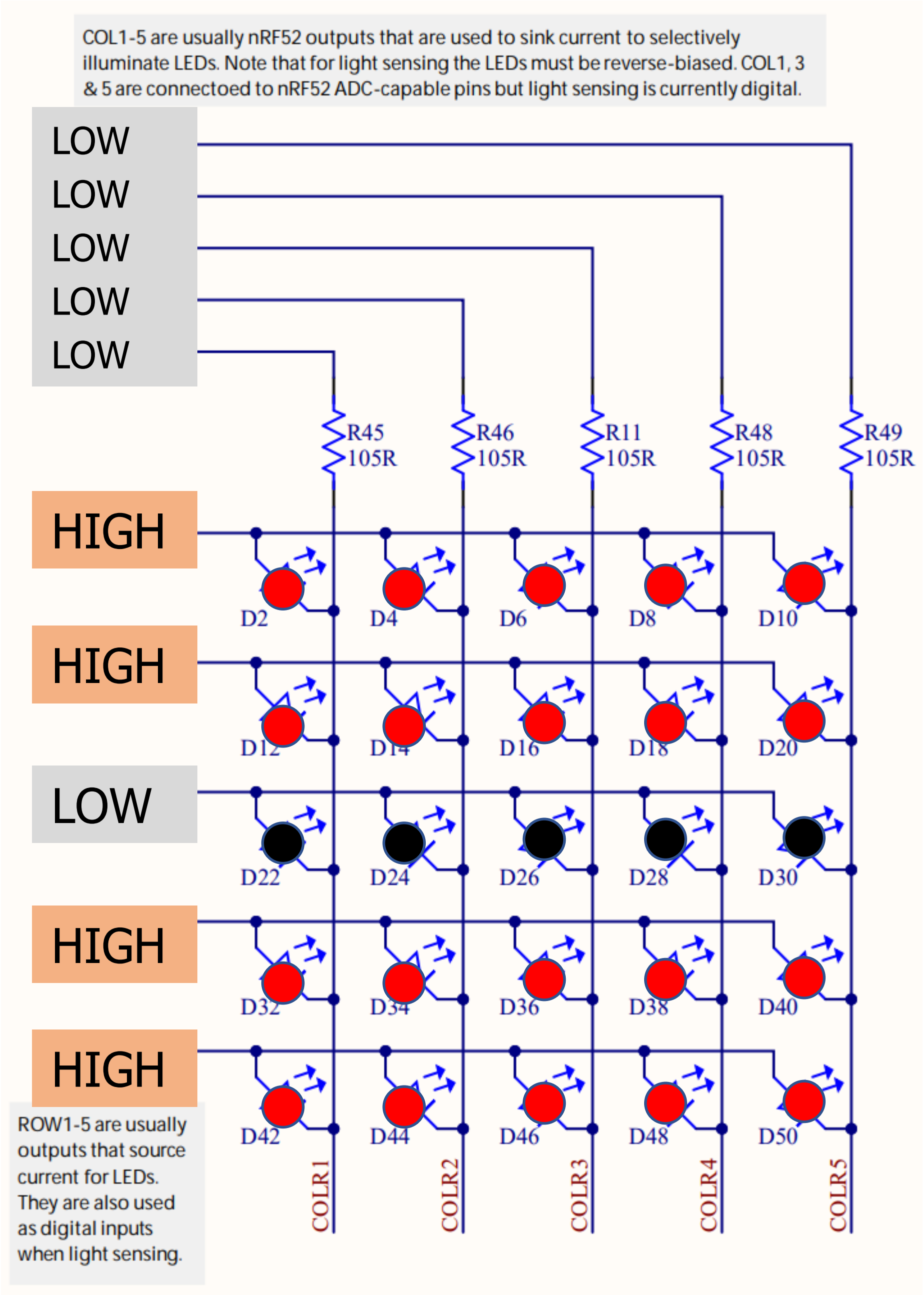
- But now how do we turn off the right middle LED?



Can we control by row?

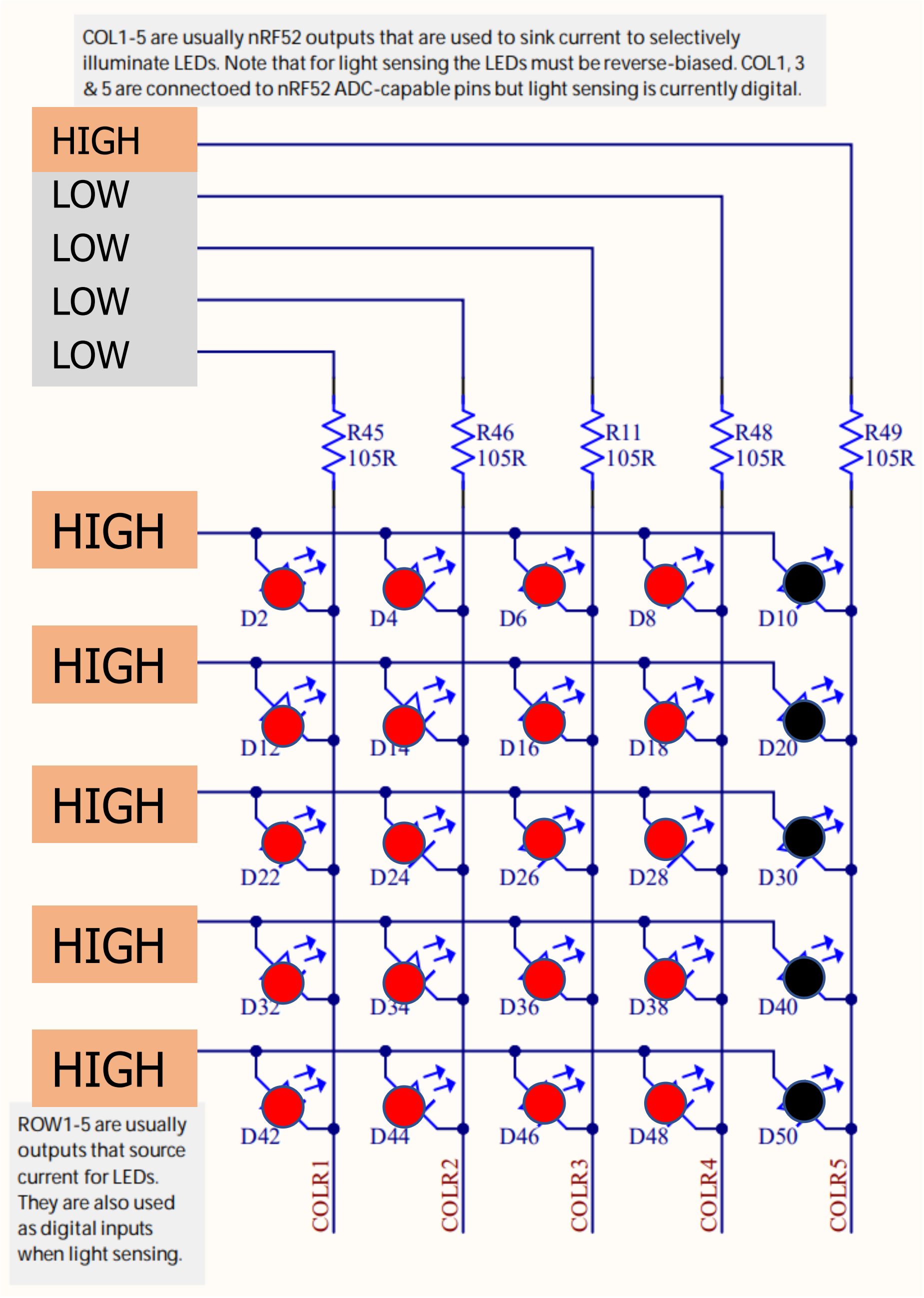
- But now how do we turn off the right middle LED?

- What if we clear the row to Low?
  - Messes up the entire row



Can we control by column?

- But now how do we turn off the right middle LED?
- What if we set the column to High?
  - Messes up the entire column
- We don't actually have arbitrary control over the whole thing at once

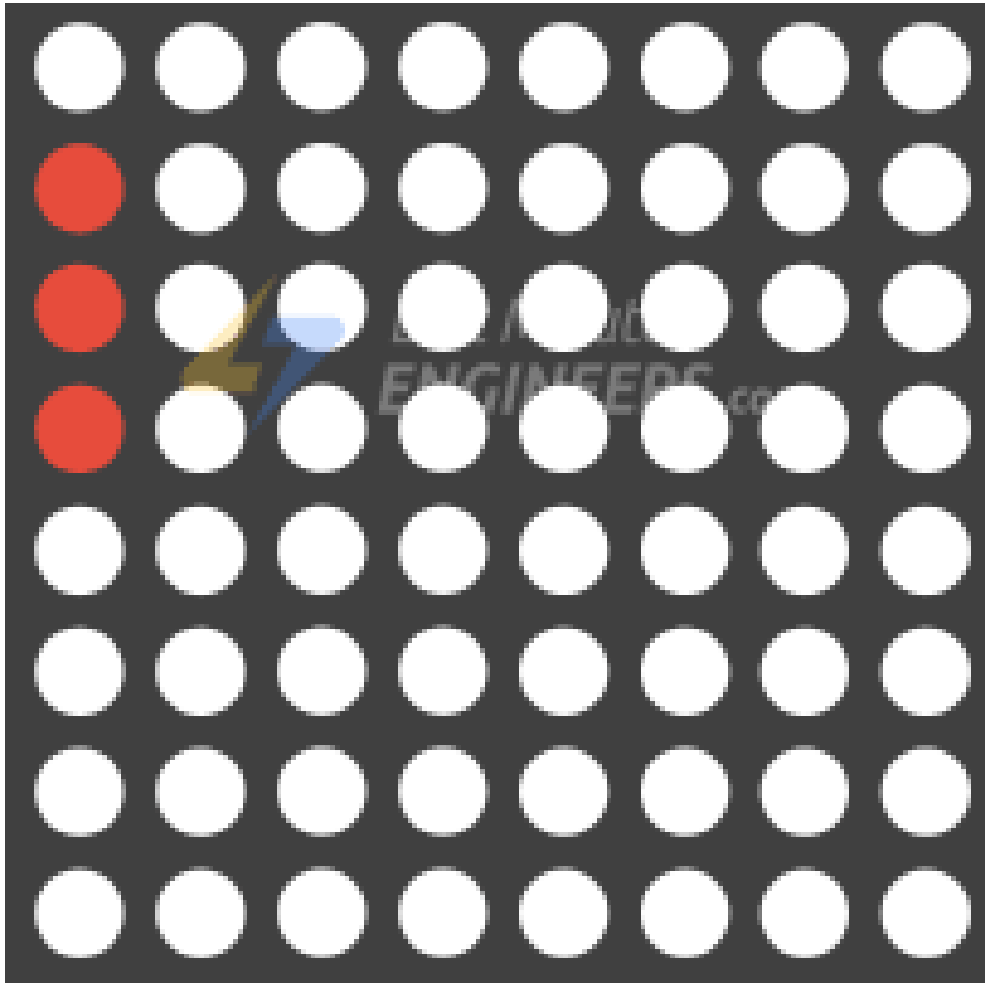


Persistence of vision

- The solution here is to abuse how human eyes work

- Eyes can’t detect changes in light that are going faster than a certain speed
  - Or if they do at all, it’s interpreted as slightly dimmer light
  - Any given LED should be above ~100 Hz to keep humans from noticing the flicker

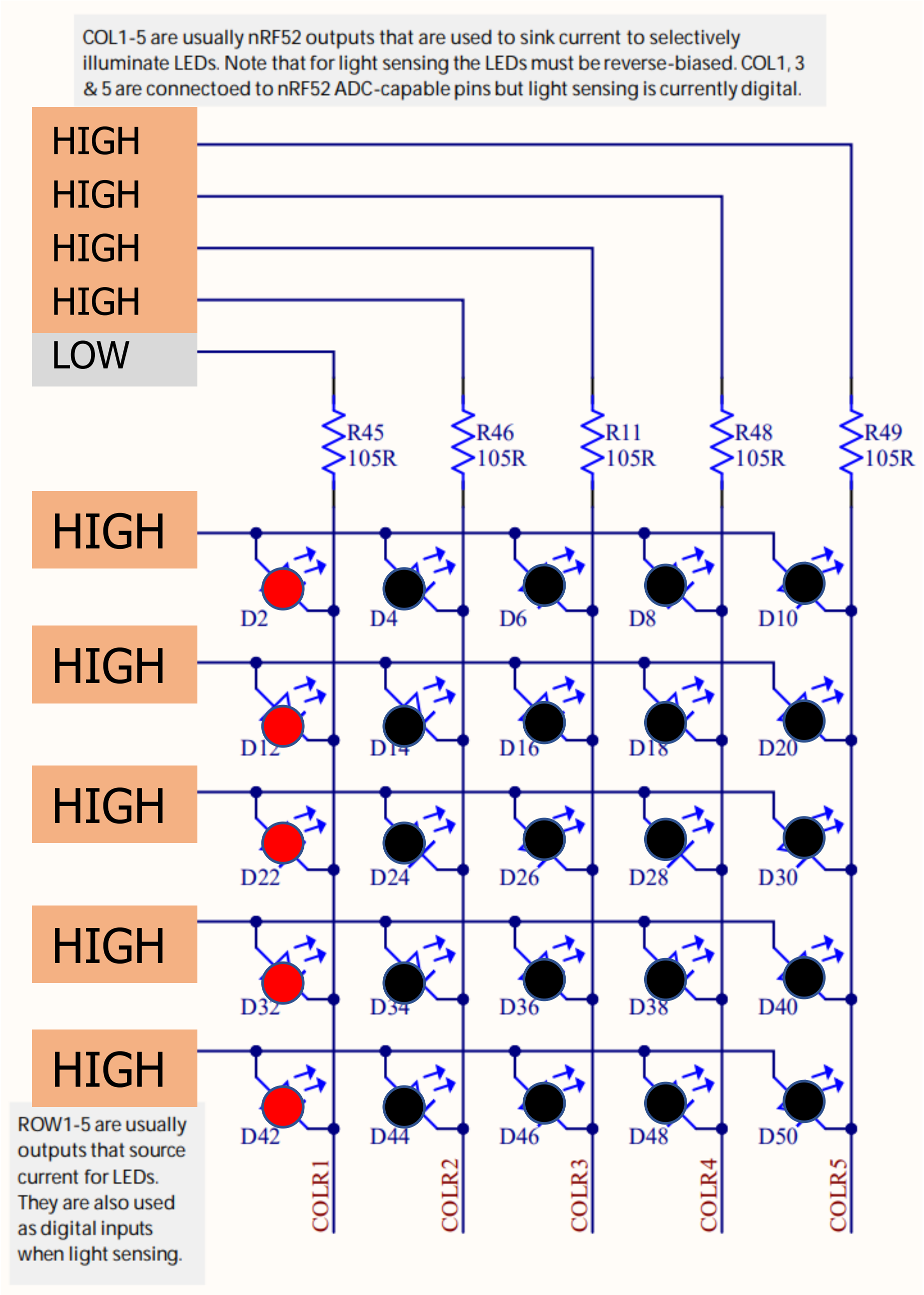
Persistance of vision on an LED matrix



One column at a time

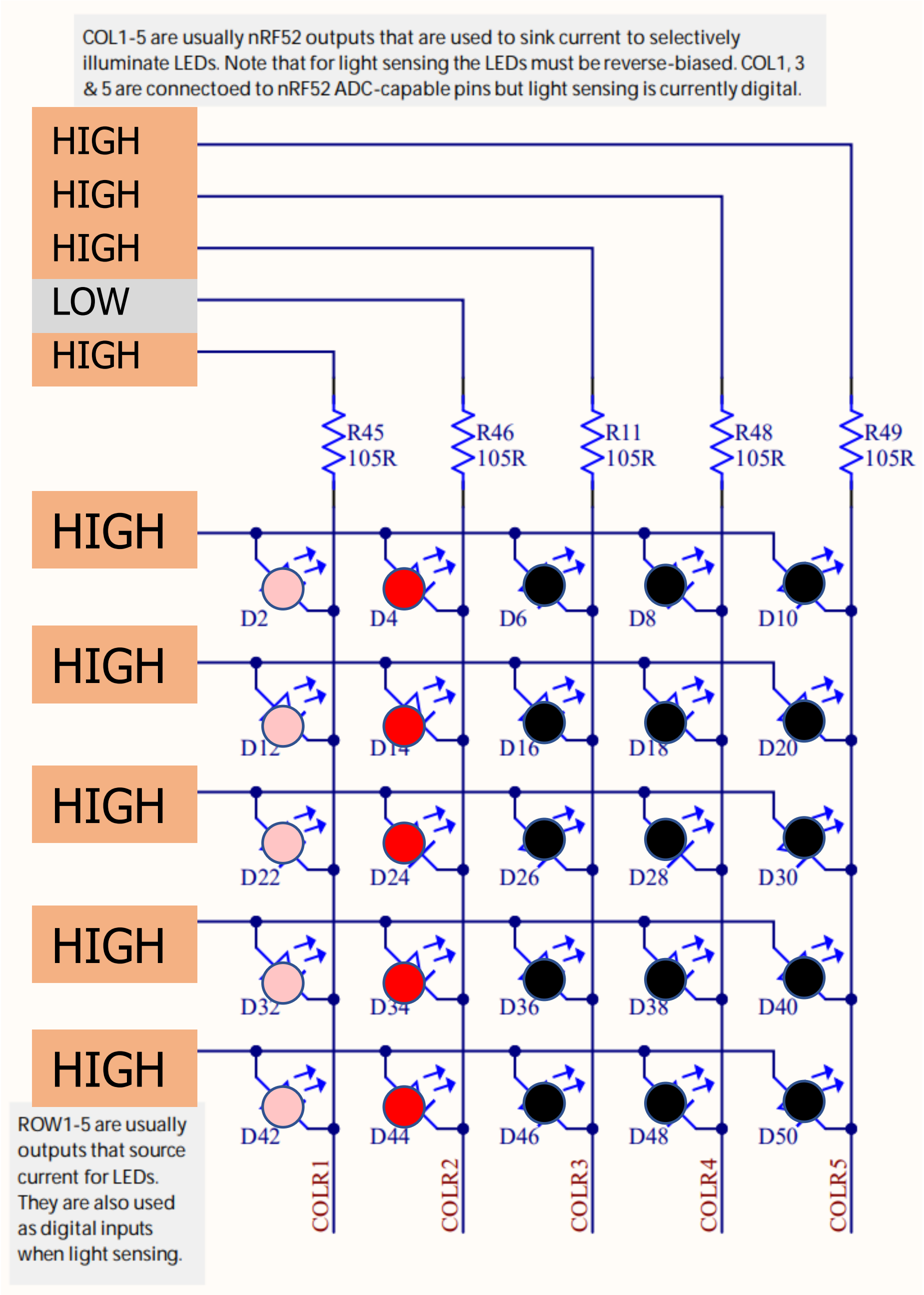
- What if we instead control a single column at a time?

- First column, all LEDs on



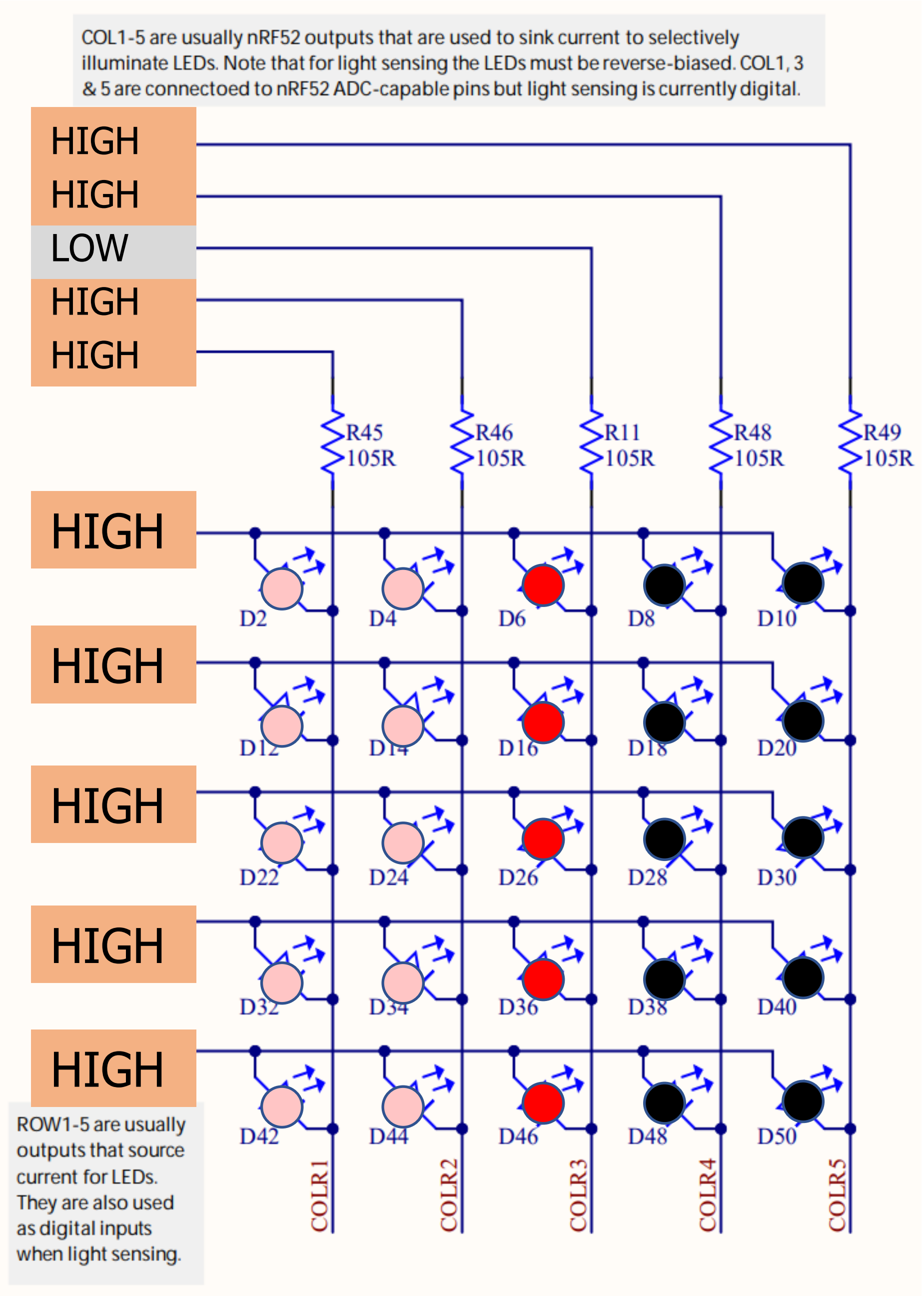
One column at a time

- What if we instead control a single column at a time?
- Same for second column through fourth column



One column at a time

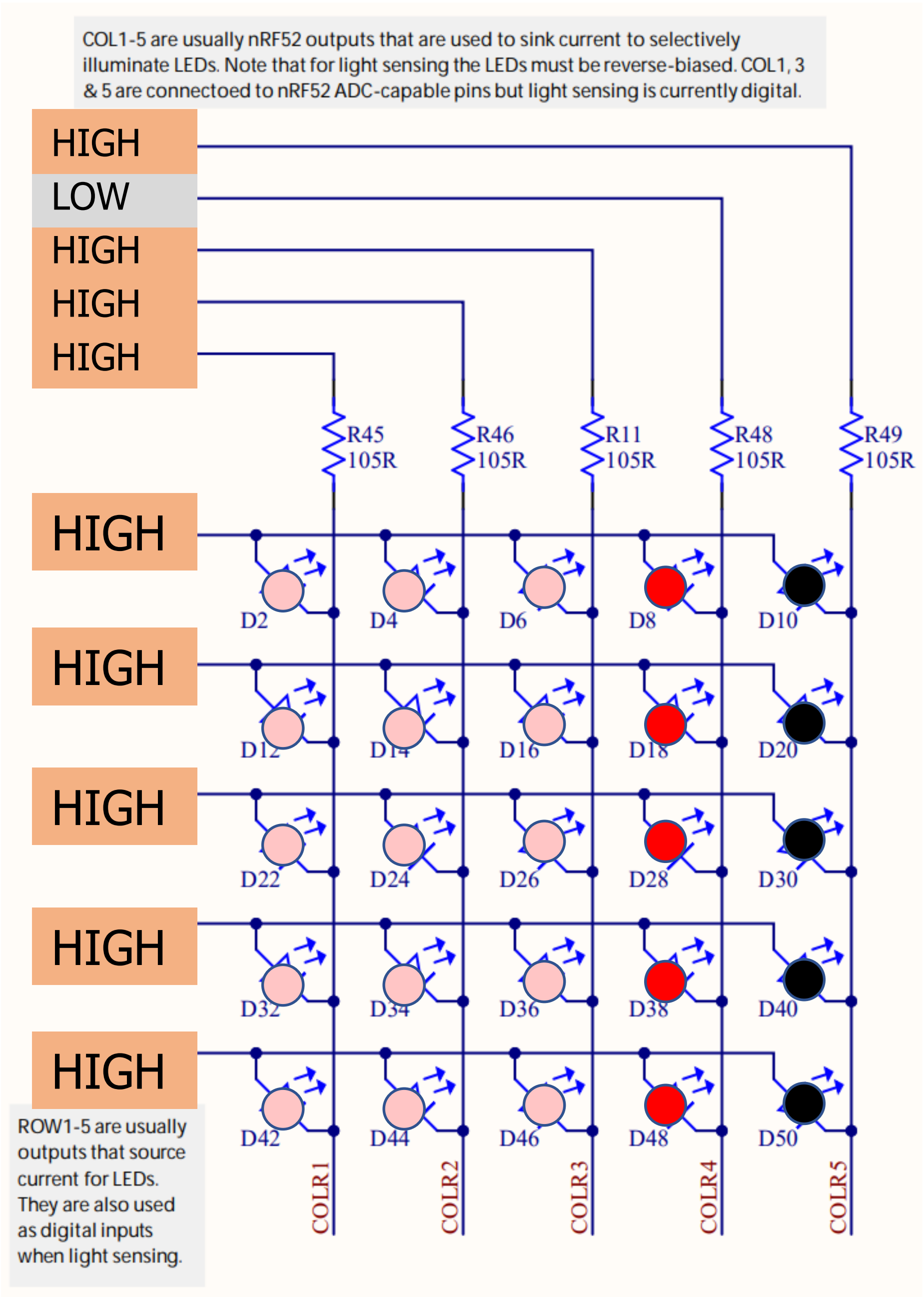
- What if we instead control a single column at a time?
- Same for second column through fourth column



One column at a time

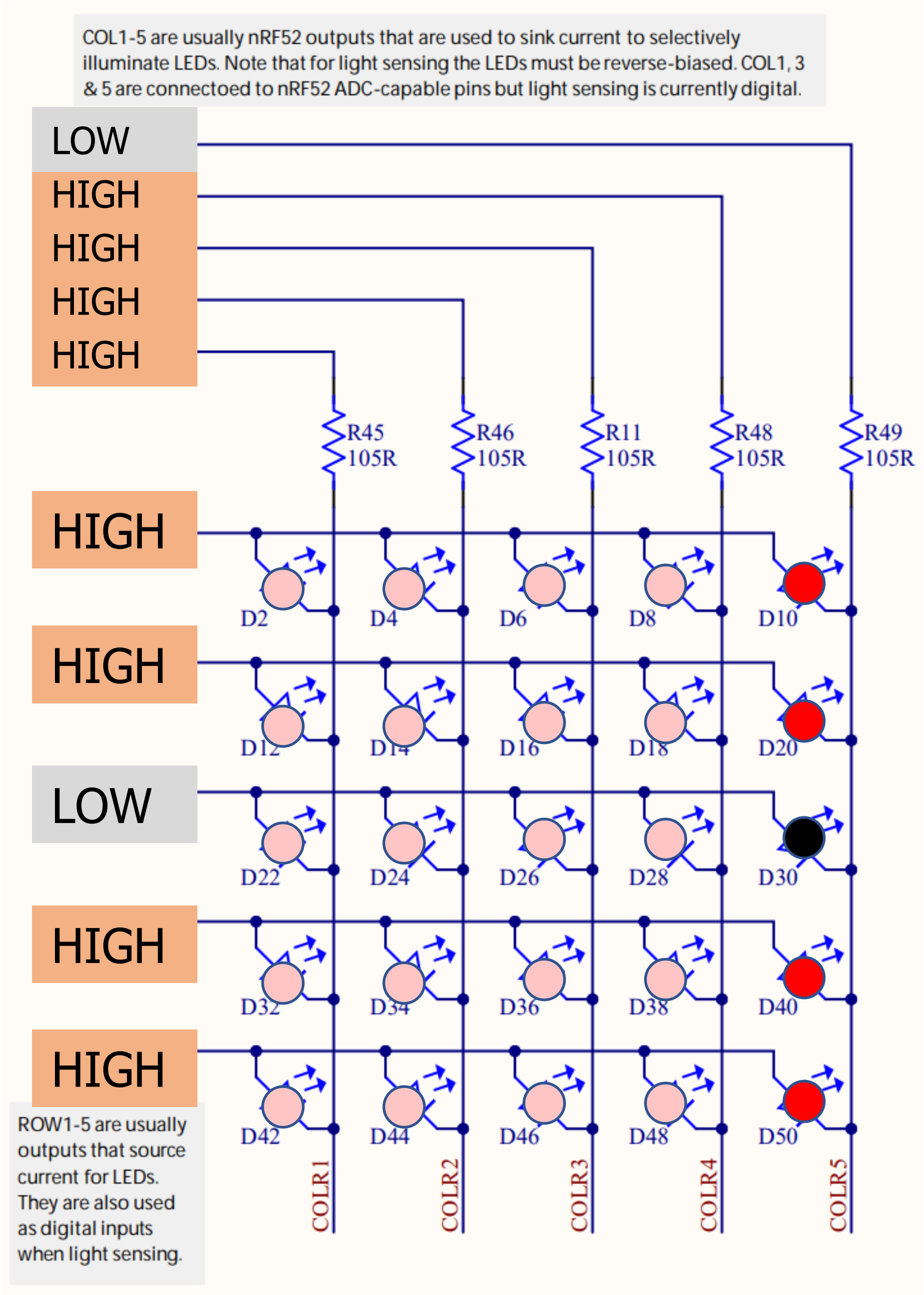
- What if we instead control a single column at a time?

- Same for second column through fourth column



One column at a time

- What if we instead control a single column at a time?
- Last column we only turn on some of the LEDs
- As long as we keep cycling through columns fast enough, the whole thing becomes a display



LED matrix full design

- Requires GPIO and a Timer

- When the Timer fires
  - Change which column you are displaying
  - Update the row pins based on this new column
    - Read row data from a 5x5 array that models what the screen should show

- When the user wants to change the display
  - Update that 5x5 array in memory
  - It’ll start getting drawn on the screen the next time the Timer fires

Outline

- Driver Interfaces (Blocking and Non-Blocking)
- Event-driven Model
- Continuous Operation