Lab 3 - LED Matrix

Goals

- Use timers and GPIO to control the LED matrix
- Display text on the LED matrix

Equipment

- Computer with build environment
- Micro:bit and USB cable

Documentation

- nRF52833 datasheet: https://docs-be.nordicsemi.com/bundle/ps_nrf52833/attach/nRF52833_PS_v1.7.pdf
    - Online version: https://docs.nordicsemi.com/bundle/ps_nrf52833/page/keyfeatures_html5.html
- Microbit schematic: https://github.com/microbit-foundation/microbit-v2-hardware/blob/main/V2/MicroBit_V2.0.0_S_schematic.PDF
- Lecture slides are posted to the Canvas homepage

Github classroom link: https://classroom.github.com/a/JAZlYMhD

Table of Contents

Lab 3 Checkoffs

You must be checked off by course staff to receive credit for this lab. This can be the instructor, TA, or PM during a Friday lab session or during office hours.

- **Part 2: LED Matrix**
    a) Demonstrate that you can enable several LEDS in a single row
    b) Demonstrate that you can modify several LEDS in a single row using the timer
    c) Demonstrate the ability to draw a simple pattern on the LED matrix
        i) Also explain the code that accomplishes this
    d) Demonstrate the ability to display a single character on the LED matrix
    e) Check your understanding
    f) Demonstrate your final application displaying multiple strings over time
        i) Also explain the code that accomplishes this
- **Part 3: Snake Game**
    a) Demonstrate your working game
        i) Also explain the code that accomplishes this

Lab Steps

Part 1: Setup

1. Find a partner

- Rule: you can pick any partner you want, but you can't pick the same partner twice
- You MUST work with a partner
    - If you can't find someone, ask course staff for help

2. Create your Github assignment repo

- There is a github classroom link on the first page of this document. Click it!
- Pick a team name
- Pick your partner
- Generally, do what it says
- At the end, it should create a new private repo that you have access to for your code
    - Be sure to commit your code to this repo often during class!
- That link might 404. If it does, you first have to go to https://github.com/nu-ce346-student and join the organization
- **Important: both of you should join the repo before you can do the next step**

3. Set up an additional Git remote

- Open a terminal if you haven't yet
- `cd` into your "nu-microbit-base" repo
- At the top right of your shiny new private repo on the Github website, there is a green button that says "Code". If you set up an SSH key, you can click the SSH tab to get that URL, otherwise you should get the HTTPS URL. Either way, copy the URL so you can enter it into terminal
- `git remote add lab3 <YOUR-REPO-URL-HERE>`
    - This adds a "remote" repo hosted on github as a source for this repo
    - (Both of you should still do this step too)

4. Individual Setup Portions

- **ONLY ONE OF YOU** should do the following steps
  - `git fetch lab3`
    - This gets the most recent commits from the new remote source
  - `git checkout lab3/main`
    - This changes your current commit to the remote source's main branch
  - `git switch -c lab3-code`
    - This makes a new branch for your lab code
  - `git push -u lab3 lab3-code`
    - This tells the new branch to push code to the new remote source
    - From now on, you can just pull, commit, and push as normal
- **THE OTHER STUDENT** should do this **AFTER** the first student finished the above steps:
  - `git fetch lab3`
  - `git switch lab3-code`
- **BOTH STUDENTS** should do this
  - `git submodule update --init --recursive`
    - Makes sure all git submodules are initialized and updated

5. Find the app starter files for this lab

- `cd software/apps/led_matrix/`
  - This lab will use the files in this directory. Your changes will be in `main.c`, `led_matrix.c`, and `led_matrix.h`
  - The last section of the lab will update `snake_game.c` as well.

Part 2: LED Matrix

1. Control a single row of LEDs

- Initialize row and column pins and set default values for them in `led_matrix_init()`
    - To do so, you can use the nRF GPIO library
        - An example of using this library can be found in `apps/app_timer_example/main.c`
    - Each LED row and column pin will need to be set as an output (10 total) using the nrf_gpio_pin_dir_set() function
        - You can refer to row pins as LED_ROW1, LED_ROW2, etc.
        - You can refer to column pins as LED_COL1, LED_COL2, etc.
        - These values are defined in `software/boards/microbit_v2/microbit_v2.h`
    - Rows and columns should default as cleared
        - You can control whether leds are on or off `nrf_gpio_pin_clear()` and `nrf_gpio_pin_set()`, or `nrf_gpio_pin_write()` lets you do either with a single function
    - If you want, you could create an array of row LEDs and an array of col LEDs and use that to get the pin numbers. Something like `uint32_t row_leds[] = {LED_ROW1, LED_ROW2, LED_ROW3, LED_ROW4, LED_ROW5};`
- Set initial values for the LEDs in `led_matrix_init()`
    - To activate an LED, ensure that its corresponding row pin is high and its corresponding column pin is low
    - To inactivate an LED within an active row, set its corresponding column pin to high
- Test your code to make sure that you understand how to choose which LEDs are active in a row
    - `led_matrix_init()` is already called in `main()` for you
- **CHECKOFF**: demonstrate that you can enable four LEDs in a single row

2. Use an App Timer to modify LEDs in a single row

- **Review the example App Timer code** in `apps/app_timer_example/main.c`
  - This example uses the nRF App Timer Library
    - This is the Nordic-developed virtual timer library
  - `APP_TIMER_DEF()` is a macro that creates a global timer variable to hold your timer configurations.
    - This is how the library handles making multiple timers. It creates a variable for you. Instead of using `malloc()`, it expects users to create each timer as a separate global variable.
    - The "argument" to the macro is the name of your global variable. Name it whatever you like.
  - `app_timer_init()` initializes the timer library
  - `app_timer_create()` initializes a specific timer
    - Arguments are: a pointer to the timer variable, the timer mode (either APP_TIMER_MODE_SINGLE_SHOT or APP_TIMER_MODE_REPEATED), and a callback function
    - That first argument should be a **pointer** to your global timer variable
  - `app_timer_start()` starts a timer
    - Arguments are: the timer variable (not a pointer!), the number of ticks until expiration, and a `void*` pointer to pass to the callback function (usually just `NULL`)
    - The App Timer uses the RTC with a Prescaler of 0, so Period in seconds equals Ticks divided by 32768

- In your `led_matrix.c` in `led_matrix_init()` initialize an App Timer and start it
  - For now, you can just set it to fire once per second in APP_TIMER_MODE_REPEATED mode

- In the App Timer callback function, modify the active LEDs for the given row
  - Make sure the row pin is high, and set some column pins to high and some column pins to low so that some LEDs are active
  - There should be at least two states for the LEDs that you switch between, and you may not use the toggle functionality to do so

- Test your code to make sure that you can actually modify the active LEDs. Change which row is active and which columns are active a few times to test your understanding
  - `led_matrix_init()` is already called in `main()` for you

- **CHECKOFF**: demonstrate the ability to modify several LEDs in a single row with the timer
    - This should modify at least two GPIO pins each time. Show that you can actually control which LEDs in a row are active at any point.
    - There need to be at least two LED states that are swapped between, and you may not use the GPIO toggle functionality to do so.

3. Control arbitrary LEDs

- Keep track of the desired state for each LED in the matrix

  You can implement this any way you want to. However, here are some suggestions:

  - You probably want to keep your state as a global variable so multiple functions can interact with it

  - You can use any method you want to keep track of LED states
    - You could use a 2D matrix of boolean values, one for each LED. That would look something like `bool led_states[5][5] = {false};`
    - Or alternatively you could use a 1D matrix of row values, where each row corresponds to an 8-bit value (5 bits for LEDs and 3 bits of padding)
    - Or alternatively you could use a single `uint32_t` to contain the state of the LED matrix (25 bits for LEDs and 7 bits of padding)

- Each time the App Timer callback function triggers, modify a different row of the LED matrix.

  Only one row of the LED matrix can be active at a time, but by lighting up one row at a time and very quickly iterating through the rows, human perception will make it look as though all LEDs are active simultaneously.

  - You might need a global variable (or static function variable) to keep track of the current row being displayed

  - Inactivate the current row, then change the column pin states, then enable the next row
    - If you do this in a different order, it will briefly light the wrong LEDs

  - First get writing each row working at **low speed,** and then when you can see rows are changing correctly, increase the frequency in the next sub-step

  - `nrf_gpio_pin_write()` may be a useful function here

- Modify the App Timer to run fast enough to refresh the LED matrix imperceptibly

  - Each LED row needs to refresh at 100 Hz or higher in order to not flicker when viewed. Because only one row is active at a time and there are five rows, this means the App Timer should fire at least 500 times per second.

    - Don’t go below ~20 ticks for the interrupt period, or else your callback handler might take long enough that you will miss interrupts.

- Test your code to make sure you can write an arbitrary pattern to the LED matrix

    - An X pattern is a good test to ensure that everything is working
        - It would be impossible to draw without going row-by-row as we are doing
        - You can make this test part of `led_matrix_init()` so it runs immediately and automatically

    - Only the LEDs you selected should light up
        - If other LEDs are very very dimly lit up, you probably enabled the next row before modifying the columns

    - The refresh rate should be fast enough that you cannot detect the LEDs changing state
        - You might need to increase the configuration of the App Timer
        - Be sure not to waste tons of time in the interrupt handler with a `printf()`

- **CHECKOFF**: demonstrate the ability to draw a simple pattern on the LED matrix
    - Also explain the code you wrote that accomplishes this
    - You MUST be using the `led_states` variable to control what is displayed

4. Display a single character

- Create a function that takes in a character and displays it on the LED matrix

  This will need to access the font array in `font.c` in order to determine which LEDs to light for a given character. It contains LED active/inactive states for the first 128 ASCII characters including uppercase and lowercase letter, numbers, and various punctuation. The first 32 ASCII characters that are not representable are left blank. Here is a visualization of the font.

  The font array is a 2D array, indexed first by character (for any character from 0 to 127). The second dimension has 5 values, one corresponding to each row (1-5). The value at a combination of character index and row index is an 8-bit value, corresponding to the 5 LED states, padded with 3 zeros (in the most-significant bits).

  For example:

  `font[82][0]` corresponds to the first row for the letter `'R'` and has the value of 0x0F, which means the LEDs in column 1, 2, 3, and 4 should be active (and column 5 should be inactive).

  `font[82][1]` corresponds to the second row for the letter `'R'` and has the value of 0x11, which means the LEDs in column 1 and 5 should be active.

  Generally, 0x1F means all LED columns are active while 0x00 means all LED columns are inactive

  - The font array is already included in `led_matrix.c` via `font.h`
  - You will need to decode the 5 rows of the character and use them to modify your LED state (which will then lead to the display changing when the timer next fires)
    - Beware: the font row and bits are zero-indexed, while the LED matrix rows and columns are one-indexed

- Test your code to actually display the character
  - You will need to add the function to `led_matrix.h` in order to call it from `main()`
  - If everything from before is working, that should be sufficient to draw the character on the display
    - Be sure to test with characters that are not horizontally symmetrical and also with characters that are not vertically symmetrical

- **CHECKOFF**: display a single character

5. Check your understanding

The font data structure we're using in `font.c` is not necessarily the most efficient way to store the data. Instead, it focuses on ease of use by reducing the number of bit manipulations necessary.

- What is the minimum possible size in Bytes to store a font of 128 separate 5x5 characters?
- What is the actual size used to store the characters in the `font.c` file?
- **CHECKOFF:** How many bytes are wasted by the representation in `font.c`?

6. Display arbitrary strings

- Create a function that takes in a string and displays it on the LED matrix

  You must have a function that takes in a string, and you must use a timer (usually a second timer) to display characters (rather than calls to `delay_ms()`), but otherwise I'm going to leave the implementation here up to you. As long as the string is displayed in a readable fashion, you'll get credit for this lab.

  Clarification: you MUST make a library function that takes in a string and outputs it to the screen. You can't just call the character display function in a while loop with delays.

  Some things to consider:
  - You should support the user modifying the text that is being displayed, likely by calling this function a second time
    - You could provide a callback function when the text is finished
    - You could turn this function into a blocking function that doesn't return until the text is complete
    - You could just return automatically after being configured
  - You likely need to keep an additional global variable or two (or possibly a struct) to contain information about the string currently being displayed and your offset within it

    - You almost certainly want a second timer to control when to move to the next character in the string

    - You can decide if you want to make the speed at which text is displayed publicly configurable
        - You could include the speed at which characters move as part of the public interface for the function if you want to
        - Or alternatively you could include the total duration in which to display the entire string
        - Or alternatively you could choose one default speed for moving between letters and not give the user any control

    - You can decide if you want to make the string repeat once it is complete
        - You could stop displaying anything once the string is completed
        - You could repeat the string automatically whenever it completes
        - You could include a boolean flag in the public interface to choose whether it repeats

    - You can decide how to move text on the display
        - Text could move like a marquee, which should be the most readable method, but also means handling part of two letters on the display simultaneously
            - This is somewhat tricky to implement but totally possible. Feel free to ask for an implementation hint if you want to do this!
        - Text could simply be written letter-by-letter, with only one displayed at a time
        - You could add other more-interesting transforms, like fade-in or fade-out

- Make the display say “Hi CE346!”, then several seconds later write “It works!”

    - You will need to add the function to `led_matrix.h` in order to call it from `main()`
    - In `main()`, you may call `delay_ms()` or use other methods to change which string is printing at which time
        - Just don’t use delays in your `led_matrix.c` file. Use `app_timer` there

- **CHECKOFF**: demonstrate the working application to course staff
    - Also show them your code and walk through the interesting parts
    - How does it move letter by letter through the string? And what happens when the string ends?

Part 3: Snake Game

For this last part of the lab, we're going to be implementing a game using the LED matrix. Our screen is very tiny, so we can't do a ton on it, but we could make a very simple version of the Snake game. If you haven't ever played it before, you can try it here: https://sites.google.com/site/populardoodlegames/google-snake

We'll keep our game pretty simple. The snake will initially start as one unit in size, but will grow one additional unit in size every 10 iterations. The game ends if the snake exits the screen boundaries or if it intersects with itself. Hypothetically, the maximum-sized snake would be 25 parts long and cover the entire screen.

1. Understand the existing starter code

- The existing code gives you a framework for the game.

  At the top are some types you'll need to use. There's a struct for holding row, column locations. There's an enum for tracking direction. There's also a large struct for holding all the state of the game.

  Below that is an implementation of a queue that you can push and pop `snakeloc_t` structs to/from. This will be used to hold the snake body parts. You shouldn't have to edit it.

  Below that is a callback handler for button presses. It will get called whenever BTN_A or BTN_B get pressed and it will save information in the game state.

  Finally, there are two public functions for the game. `snake_game_init()` is written for you and initializes the state of the game. You shouldn't need to edit it, but you can if you want to add or change something. `snake_game_advance_state()` is the function you'll need to implement.

- The general plan for the game is to automatically move the snake in its current direction across the screen by popping the tail from the queue and adding a new head to the queue. Button presses should change the current direction the snake is heading.

  The game keeps track of state in the `game_state.model` array which represents each pixel in the screen and is `true` at the positions the snake is currently occupying. At the end of each iteration, the game's state should be drawn to the LED matrix.

2. Setup code

- You'll need to initialize the game in `main.c` after initializing the LED matrix. Then you will need to call `snake_game_advance_state()` in the main loop. You can choose how much to delay between each iteration of the game. Shorter delays make the game more difficult. I have a hard time playing it faster than 200 ms per iteration.

- You will also need to add a function to `led_matrix.h` (and `led_matrix.c`) that allows you to write arbitrary pixels on the screen. You can implement this function however you like, but I recommend a function that takes in a 5x5 `bool` array as that's likely the easiest to both use and implement.

3. Implement the game

- This one's on you. There are a bunch of comments in the `snake_game_advance_state()` function to guide you.

  You're welcome to add additional features or modify how the game works, as long as you're keeping the general spirit of the game.

- A tip: start by implementing movement and screen display without bothering to implement collision detection or button presses. That will let you visually see if your code is working.

- **CHECKOFF**: demonstrate your working game and the code that you wrote

    - You'll need to demonstrate the snake getting long enough to intersect itself. So you might either need to slow down the game or get good.