Exercise no.: 5

User menu on OLED

Objectives

In this exercise you will create an application with a simple user menu displayed on the OLED screen and navigated using 3 user buttons (up, down and OK). This application will serve as a framework for future exercises, which will just add new options to the menu, so write it well.

1. Connect 3 buttons and both I^2C sections: OLED and I^2C Devices. Use separate I^2C buses for them (i2c0 and i2c1). It might be a good idea to connect a user LED as well for debugging.
2. Using BeamKit board implement a program with 3 tasks executing the following behavior:
    a. Task 1
        i. Poll the accelerometer for fresh XYZ data with at least 10 Hz frequency and make it available to other tasks through global variables.
    b. Task 2
        i. Detect button presses for all 3 buttons, taking care to debounce them properly.
        ii. On detected presses send that information into a button event queue.
    c. Task 3
        i. Render a menu with the following positions:
            1. Home screen containing the name of your device. It can be funny but make it not controversial.
            2. Accelerometer option (no data displayed at this level).
            3. System info (no data displayed at this level).

        Buttons Up/Down are used to switch between these positions. Button OK is used to select a position, which changes the contents of the display in the following way:

            1. Home screen can't be selected. Ignore OK button.
            2. When accelerometer screen is selected display the current accelerometer XYZ data. Make sure it is dynamically refreshed while the option is selected. Use proper formatting through the *sprintf()* function.

3. When system info screen is selected display the pico SDK version (three digits, like: SDK 2.2.0). Use the proper SDK macro to access that information.

While a screen is selected Up/Down buttons are ignored and a single press on the OK button takes the user back to the screen selection.

ii. Details of the application architecture are up to you but in such a simple application it makes sense to consume button events in the same task that is responsible for drawing the data on the screen.

Feel free to add more tasks as you see fit but keep in mind that simple solutions tend to save time. Your code should be readable but don't waste time on making it "perfect" ;) Avoid copy-pasting too much and name entities logically.

Keep all your .c files shorter than 1000 lines of code. Create new files if you need to but remember to add them to you cmake. It makes sense to create separate .c and .h files for accelerometer and OLED drivers.

Description

Embedded devices often need to contain a user interface and there are many ways to implement it in practice. In this exercise you will implement a simple physical user interface based on a small 128x32 OLED screen that can fit 2 or 3 lines of legible text and simple graphics. It features a SSD1306 driver which is a very popular option for cheap, low resolution, monochromatic OLEDs. Such screens can serve as a decent option for the primary user interface on simple devices and can be used as secondary displays for maintenance information like error codes, IP address, status etc. in larger systems.

With this exercise we change our approach from doing small, self-contained projects to writing a large application to which more functionalities will be added later. This is more representative of typical embedded projects and will help you learn RTOS mechanisms.

Hints

1. This task requires you to write a lot of code and can easily take 2 lessons to complete even for competent coders. If you feel like you are falling behind consider writing some code at home and test it during lab hours.
2. Reuse your old code for LIS3DH accelerometer from previous exercise.
3. Both LIS3DH and SSD1306 drivers are included in SDK examples. Refer to them to make you work faster but remember you can be asked about that code when submitting your solution so analyze it.
4. Getting everything to work at once is difficult in large projects. Divide your work into manageable chunks, for example: first write and test initialization of the OLED screen (it will display noise after initialization if you don't provide any data), then try to draw a simple line and only then try displaying strings. When you make sure your low-layer drivers work, move to writing the high-level logic of your application.
5. To make your code elegant and easier to work with in future consider making separate .c/.h files for SSD1306 and LIS3DH drivers. The most convenient way to organize a small number of files is to keep them in two directories: 'src' for source files and 'inc' for header files, directly in the root directory.
   You are free to organize files as you see fit, but the scheme described above requires the following changes to your cmake:
   - Add this line: *include_directories(inc)*
     for example below sdk_init. That will add your header files to the build.
   - Add manually your .c files to the section where you define your executables, for example like this:

     add_executable(exercise_menu
     exercise_menu.c
     src/ssd1306.c
     src/lis3dh.c)

   Note that *add_executable* section already exists in your cmake file so don't create another.
6. Some modifications to the code from SDK examples might be required to get your project to work. Don't hesitate to make these changes. You may consider using Git to be able to easily revert unsuccessful changes, although that isn't required.
7. Make sure fresh image is rendered on the screen whenever needed. You can either implement a periodic full refresh or limit yourself to manually refreshing whenever its required. If you decide to go for the periodic refresh, make sure you avoid glitches resulting from sending a frame buffer to the screen while the buffer is being edited by another task.

8. Use *typedef enum* in your code instead of magic numbers whenever possible to make it more readable and easier to debug.
9. Use *sprintf()* function to prepare strings with embedded numbers when dynamic data (for example, from the accelerometer) is to be sent to the display.
10. The more complicated your project becomes, the more sense it makes to separate functionalities into independent abstraction layers connected through well-defined interfaces like queues, shared memory etc. For example: one task handling sensors, second caring about display contents and another one for internal logic. While this separation is beneficial from the point of view of the future extension and maintenance of the application, introducing it into small programs usually results in performance penalty and a significant increase in so called "boilerplate code" which handles internal interactions between different software components without contributing to the target functionality of the device. Finding the right balance for a given project is difficult and depends on project requirements like: testability of the code, project lifecycle, company procedures etc. During this course, this decision is up to you, as long as the exercise objectives are met.

Links

FreeRTOS Beginner's Guide: https://www.freertos.org/Documentation/01-FreeRTOS-quick-start/01-Beginners-guide/00-Overview
"Mastering FreeRTOS" book and Reference Manual download page: https://www.freertos.org/Documentation/02-Kernel/07-Books-and-manual/01-RTOS_book
Official RP2350-compatible FreeRTOS kernel: https://github.com/raspberrypi/FreeRTOS-Kernel/