Lab 1

Re-submit Assignment

Due No Due Date **Points** 0 **Submitting** a file upload **Available** after Jan 28 at 12am

Welcome to the first lab of this course!

This lab will introduce you to the hardware we are using, and to the SDK provided by the vendor Nordic Semiconductor.

Requirements

For this lab, you will need the **development board** that you got at the beginning of the course. In addition, you will need some **software** to actually program your board. For this lab you will need:

- C compiler for the ARM platform: Please follow the instructions given **here** (https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm). Remember the location you used for installing. After you have installed the compiler, make sure that you can use the command "**arm-none-eabi-gcc**" by typing it into your command line.
- The make tool. If you are using a *nix system, it should already be installed, or will most likely be in your package manager. For Windows, you can either install it directly, or use a tool such as Cygwin or MinGW.
- nRF5 SDK: This consists of two parts. One is the SDK itself, which is distributed in source code, and you can download it **here** (https://www.nordicsemi.com/Software-and-tools/Software/nRF5-SDK). The second part consists of the nRF command line tools, which you need to actually flash software to your board. You can find those **here** (https://www.nordicsemi.com/Software-and-Tools/Development-Tools/nRF-Command-Line-Tools/Download#infotabs).
- (Optional) To use the logging output you will need the Segger RTT Viewer, which can be downloaded **here** (https://www.segger.com/products/debug-probes/j-link/tools/rtt-viewer/).

You are now ready for doing the first lab!

Template

To get you started, please download the template for lab 1 **here**. Look through the code and try to understand it. You will not have to make any changes to the makefile, all your changes will happen in main.c and sdk_config.h.

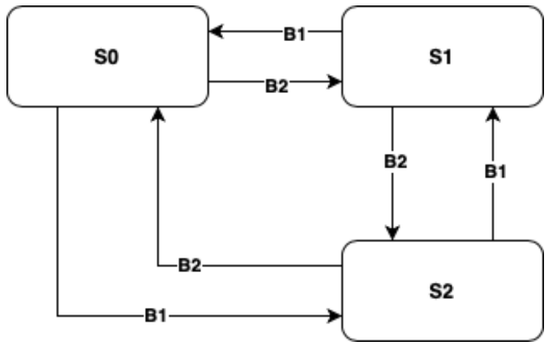
Inside the template, you must adapt the SDK path to point to the folder where you have put the SDK when you downloaded it. You will most likely also have to adapt the compiler path in the makefile.

For this, inside the SDK folder, navigate to components/toolchain/gcc and open the Makefile.posix or Makefile.windows depending on the operating system you are using. Here you have to adapt the compiler to have the correct version number and location according to your installation.

You may use some of the code you have already seen during the lecture! It is located under Files/Code here on Studium.

Goal

At the end of this lab, you will have implemented a state machine with various states, each of which is determined by a custom light pattern shown through the LEDs on the development board. State changes are triggered by button presses. The overall state machine is illustrated below:



As can be seen, there are three different states. The first and second button are used to switch between these states in a clockwise (respectively counter clockwise) manner. The different states are defined as follows:
When the system is in state **S0** it will display a clockwise turning light rope, e.g. LED1 will light up, after a short while it will turn off and LED2 will turn on. Next will be LED4 (observe the LED layout on the board), and so forth.
In **S1** all LEDs will turn on for a while, and then all will turn off, this behaviour is repeated (i.e. you will blink all LEDs).
S2 is the same as **S0**, but counter-clockwise and faster.

After each part, get one of the teaching staff members to check your solution!

If you need help, or want a teacher to look at your solution, write your group number on the document **here** **(https://docs.google.com/document/d/1d1YYCWQmm_XQuUdFsMeFQxzr_kEusym4z5lWBx19ZAc/edit?usp=sharing)** .

Part 1

If you inspect the code, you will find, that in the beginning we define some custom types, which we will use for our application. In part 1 of this lab we will only deal with the first state, **S0**. If we look at the type specification for a state

```
typedef void (*state_func_t)( void );

struct _state
{
    uint8_t id;
    state_func_t Enter;
    state_func_t Do;
    state_func_t Exit;
    uint32_t delay_ms;
};
typedef struct _state state_t;
```

we can see, that each state has an ID, three different functions that are called when entering, executing, and exiting the state, and a delay, which will determine how often we check for a button input. To make it easier for you, in the template we have already provided you with the function signature for the executing state function (**void do_state_0(void)**), and also included parts of the execution routine in the main function. As a first step, implement the three different functions for the first state. When you have implemented your state functions, fill out the **const state_t state0** declaration given further down in the code. Think of a good value of the delay, so that you can clearly see the light rope (also consider that when implementing **S2**, you will need to make that one faster, so you should give yourself some margin here). When you have implemented the state0 variable, you can turn to the main function, and finish it in a very basic implementation, that calls the appropriate state functions at appropriate times.
You can now compile your code by typing "**make**" in the terminal in the project folder. When your program has compiled you can upload it to your connected board by typing "**make flash**".
Hints:

You might not have to implement all state functions. Maybe there is already a good fit in the SDK?

Part 2

For this part you will implement the other two states and also the state changing mechanism. For this, start by implementing the state functions, and also the state declarations according to how you've done it in the first part. Once you are finished, fill out the state table **const state_t state_table[][]** in the code. Observe, that you need to include a possible transition for each event type that you have defined, even if in your current state, you do not adhere to certain inputs. In those cases, the state should remain the same, i.e. you can put the current state in those columns (one such case is for the no_evt, which signals that no input was received).
Now that you have implemented the states, you also need to implement the state transitions. For this, make use of the button library from the SDK. The communication between your button event handler and your main function can be done in different ways. We suggest to use the atomic FIFO library of the SDK as seen during the lectures. Make sure to use a proper button debounce delay! For better readability you can also implement a function **event_t get_event(void)** that looks if there is a new data element in the FIFO and returns the appropriate event. This function can then be used in the main function to determine if a state change needs to happen.
You will also have to update the execution mechanism in the main function. For reference, you might want to consult the lecture slides about the traffic lights example again.

Part 3

By now you have a functioning state machine, which allows you to switch between different light patterns by pressing the first and second button on your board. Maybe you think, that implementing the state machine in the table driven way was a bit of an overkill for this example, and you are right, this could have also been done in a different (probably easier) way. However, to see why table driven implementations are quite powerful, we will now introduce a new state **S3**. This state is entered whenever you are in any of the other states, and press button 3. In this state, all four LEDs will slowly fade between full brightness and being turned off for 10 seconds, before automatically switching to **S0**. To implement this, you will need to make use of the **LED softblink library**, and also program a custom application timer, which will trigger an interrupt and put a timeout event in your event FIFO.

Hints:

You will need to include the correct header files for the softblink library. To learn how to use this library correctly, refer to the SDK manual online, and/or look at the example application provided by the SDK.

You will see now, why using an atomic FIFO was a good choice from the beginning. We can use the same FIFO in a different interrupt service routine, since all accesses are atomic, and therefore use the same data structure to forward commands between interrupts and the main application.

When you are finished with the lab, and have shown your solution to one of the teachers, please upload your code as well! Only one member per group has to upload the group's code.