Lab 1 - Memory-Mapped IO and Interrupts

Goals

- Create a GPIO driver using memory-mapped I/O
- Explore interrupts

Equipment

- Computer with build environment
- Access to a USB-A port on your laptop (if you only have USB-C, bring a dongle)
- 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 schematics: Version 2.0 and Version 2.21
- Lecture slides are posted to the Canvas homepage

Github classroom link: https://classroom.github.com/a/4nanuUZB

Lab 1 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 1:** Setup
    a. Demonstrate the error app running and the message it prints

- **Part 2**: Interrupts
    a. Demonstrate triggering an interrupt with GPIO
    b. Show your application code in main.c
    c. Demonstrate your application showing preemption of interrupts

- **Part 3**: Using Memory-Mapped IO to control GPIO
    a. Demonstrate code that controls the Microphone LED with raw MMIO addresses
    b. Demonstrate that you can turn on all the LEDs on the back of the Microbit
    c. Show your MMIO struct and library code in gpio.c
    d. Show your application code in main.c
    e. Demonstrate your application controlling the LED with buttons

Also, don't forget to answer the lab questions assignment on Gradescope.

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, talk to Branden

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 github classroom 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 so, you first have to go to https://github.com/nu-ce346-student and join the organization
- **Before you clone your repo**: you must also create a personal access token or use an SSH key. SSH keys are preferred and well-explained here. You should really just do an SSH key.

  If you really don't want use an SSH key, below are instructions for personal access tokens instead:
    - Go to your github profile -> Settings
    - Then Developer Settings on the left
    - Then Personal Access Tokens on the left
    - The click the Generate New Token button on the top right
    - Add a note that is the name of this token (not important, type anything)
    - **IMPORTANT:** set the expiration time to 90 days so it doesn't expire during the quarter
    - **IMPORTANT:** check the repo checkbox below the name of the token
    - Then scroll to the bottom of the screen and click the Generate Token button
    - This will create a password that allows you to clone repos
        - It will only show this once, so copy-pasting it into a google doc in your personal drive would probably be useful
        - It will be in gray at the top of the screen

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 lab1 <YOUR-REPO-URL-HERE>`
    - This adds a "remote" repo hosted on github as a source for this repo1
- **ONLY ONE OF YOU** should do the following steps
    - `git fetch lab1`
        - This gets the most recent commits from the new remote source
    - `git checkout lab1/main`
        - This changes your current commit to the remote source's main branch
    - `git switch -c lab1-code`
        - This makes a new branch for your lab code
    - `git push -u lab1 lab1-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 lab1`
    - `git switch lab1-code`
- **BOTH STUDENTS** should do this
    - `git submodule update --init --recursive`
        - Makes sure all git submodules are initialized and updated

4. Get a Microbit

- Both you and your partner should come up to the front and claim Microbits
    - We'll write down which Microbit you take
    - It will be yours for the entire quarter!
    - Take a USB-Micro cable for the quarter too, if you need one

5. Program a board

- Plug the board into the computer
    - WARNING: if you haven't loaded code on it before, the default app makes noise
        - And is rather annoying
    - You plug into the USB on the top of the board
- If you have a VM, attach the board to the VM
    - A pop-up might appear asking you where to attach the device. Attach it to the VM. If nothing pops up, attach it manually:

        - In the menubar, click Devices->USB->something something BBC Microbit
        - If you look at Devices->USB again, it should be checked
        - You'll have to check this button each time you plug in a board. There will be a separate one for each board you have attached to the computer.
- In the blink app folder
    - `make flash`
    - It should pop up a window with a loading bar that uploads the code
    - Things like "** Programming Started **" and "** Programming Finished **" are good
    - Things like "Error: unable to find CMSIS-DAP device" and "Error: No Valid JTAG Interface Configured." are bad
        - Possibly your board just isn't attached to the VM if you're on Windows?
    - Also, the board should start blinking the red microphone LED if it works

6. Get some apps working

- There are three good starter apps:
    - blink - blinks the microphone LED
    - printf - periodically prints a message from the board
    - error - demonstrates a hardfault and error messages on the board
- Commands to control them. Use these in a terminal
    - `make flash`
        - To build code and load it onto the board over JTAG
    - `pyserial-miniterm /dev/ttyACM0 38400`
        - To listen to serial output
        - **MacOS users**: change that second argument to `/dev/tty.usbmodem<SOMETHING>` hit tab to autocomplete the something part. Remember to still have the `38400` at the end
        - If you have an older version, it might just be called `miniterm`
        - (Any other serial console would work too)
        - Note: it doesn't buffer output. Anything that happened before you opened it won't appear. Hit the "Reset" button at the top of the Microbit to start the currently loaded program again.
        - Also note: you don't have to close this when programming a board. Just leave it open in another terminal window. It should only stop working if you unplug your Microbit.
    - If the response to `pyserial-miniterm` is "command not found", you either didn't install it or didn't add it to your path
        - Go back to the Personal Lab Setup instructions and find the line explaining how to install pyserial for your system
        - Ask for help if things still aren't working
- Take a look at the code for each of the starter apps and try modifying board behavior

- **CHECKOFF:** demonstrate the error app running and the message it prints
    - *Question:* what causes that app to error?
    - Remember this blink pattern for the future!! If you ever see it, it means your code crashed

Part 2: Interrupts

The goal here is to trigger interrupts and see interrupt handler functions get called automatically. We also want to demonstrate the ability to “nest” interrupts, such that a higher priority interrupt “interrupts” a handler that’s already running for a lower priority interrupt.

You’ll need to reference the nRF52833 manual for much of this.

7. Find the app starter files for this lab

- `cd software/apps/interrupt/`
    - This lab will use the files in this directory. All of your changes will be in `main.c`

8. Trigger an interrupt with GPIOTE

- Find the documentation for configuring the GPIOTE
    - The GPIOTE register definitions can be found in the GPIOTE section of the manual
    - The GPIOTE peripheral has 8 “channels”, each of which could be configured to generate an interrupt event. In our case, we’ll only need to use channel zero to start, which corresponds to `CONFIG[0]`
    - The CONFIG register has five different fields that can be set: MODE, PSEL, PORT, POLARITY, and OUTINIT
- Configure the input pin with GPIOTE using the CONFIG register
    - The MMIO struct is already made for you. Access it as `NRF_GPIOTE->REGISTER`
        - For example: `NRF_GPIOTE->INTENSET` or `NRF_GPIOTE->CONFIG[0]`
    - We’re going to connect this channel to one of our buttons on the Microbit
        - Button A is P0.14 and is active low (pushing the button generates a low signal)
        - Button B is P0.23 and is active low
    - The configurations you’ll need to set are:
        - MODE: Event
        - PSEL: the pin number you want
        - PORT: the port you want
        - POLARITY: HiToLo (pushing the button makes the signal go low)
        - OUTINIT: irrelevant in Event mode (probably leave in default state)
    - Tip: there are multiple ways to set bits in C.
        - You could use hexadecimal
            - `0x11` would set bits zero and four
        - You could use bit shifts and bit-wise OR operations
            - `(1 << 4) | (1 << 0)` would set bits zero and four
- Enable the interrupt event from the GPIOTE peripheral
    - This is the INTENSET register. Go find the documentation for it in the manual

    - Make sure you are setting the `INTENSET` register correctly.
        - Enable interrupts only for `IN[0]` (the event for channel 0 of the GPIOTE) and leave all other interrupts disabled
- Enable the interrupt in the NVIC and set its priority
    - Functions for interacting with the NVIC:
        - `void` **`NVIC_EnableIRQ`**`(uint8_t interrupt_number);`
        - `void` **`NVIC_DisableIRQ`**`(uint8_t interrupt_number);`
        - `void` **`NVIC_SetPriority`**`(uint8_t interrupt_number, uint8_t priority);`
    - Interrupt numbers are defined for you in headers and you can use the names in your code. Relevant names:
        - `GPIOTE_IRQn`
        - `SWI1_EGU1_IRQn`
        - For example: `NVIC_EnableIRQ(GPIOTE_IRQn)`
    - Make sure to set a priority too. Priority is a number from 0 to 7 where a lower number is higher priority (pick anything for now)
- Do something in the handler to show that you're there
    - For this step, the `GPIOTE_IRQHandler()` will be what runs
    - I recommend `printf()`. Loops and `nrf_delay_ms()` can also be used
- Trigger a GPIO interrupt
    - Upload the code that you've written to the board
    - If everything is configured correctly, pressing the Button should trigger an interrupt and cause the code in the `GPIOTE_IRQHandler()` to run
    - **Debugging:** if things don't work, the most common cause is not setting all the right bits in the `NRF_GPIOTE->CONFIG[0]` register, so double-check those.
- **Checkoff**: demonstrate that you can trigger an interrupt with GPIO via a button-press

9. Trigger a software interrupt

- Use the functions `software_interrupt_init()` and `software_interrupt_trigger()` to do this
    - They are already written for you in the starter code
    - They trigger interrupts through the Event Generation Unit (EGU) peripheral
- You will also need to set the priority of the software interrupt as previously done for GPIO
- **No checkoff**: continue to the next step

10. Nested interrupts

- Make the GPIO interrupt preempt the software interrupt
    - Lower priority numbers take precedence over higher priority numbers

    - Use some combination of a for loop, `printf()`, and `nrf_delay_ms()` to make the software interrupt handler run for long enough that you can press a button and observe the effect
- **Checkoff**: demonstrate preemption occurring to the course staff
    - Also show your code in main.c
    - *Question:* how do you know it's preempting?

Part 3: Using Memory-Mapped IO to control GPIO

11. Use raw pointers to control an LED

- Look through the section on GPIO in the nRF52833 manual.
    - Particularly take a look at the registers for the GPIO peripheral
- Start with the application at `software/apps/gpio/` in the `main.c` file
- Enable the Microphone LED with raw memory-mapped IO addresses
    - The Microphone LED is Port 0, Pin 20 and is active high
    - You will need to write to the DIR and OUT registers (in that order)
        - Alternatively, the SET/CLR versions of those
    - To write an individual bit, you'll need the bit shift operator `<<` https://www.arduino.cc/reference/tr/language/structure/bitwise-operators/bitshiftleft/
    - This should only take two lines of code
    - Take a look at the `apps/temp_mmio/` example app for syntax
- **CHECKOFF:** demonstrate this code to course staff

12. Implement GPIO library

- Code for the GPIO driver library goes in gpio.c and gpio.h.
- First, create a struct GPIO MMIO registers
    - The GPIO register definitions can be found in the GPIO section of the nRF52833 datasheet.
    - Each type should be a `uint32_t`
    - You can use arrays of `uint32_t` to specify gaps in the address space
    - You can also use arrays of `uint32_t` to specify repeated registers (such as PIN_CNF)
    - Be sure to use the `volatile` keyword when actually instantiating your structure pointer as a global variable.
    - You'll need two struct pointers, one for each port
        - Alternatively, an array of two struct pointers
- To test that your GPIO MMIO register struct is correct, print out the address of a few registers and double-check against the datasheet
    - You can print them inside the `gpio_print()` function
    - You can print pointers with the format specifier `%p`
    - The following code takes the address of a struct member: `&(struct->member)`
    - To see the print output run the following in a terminal:
        - `pyserial-miniterm /dev/ttyACM0 38400`
        - You can leave it running to keep seeing output throughout the lab!
- Implement the functions in gpio.c using your MMIO struct.
    - `gpio_config()` sets the the given pin to be an input or an output
    - `gpio_set()` and `gpio_clear()` set the pin to high (set) or low (clear)
    - `gpio_read()` reads the value from the pin

    - Configuring a pin as an input requires both setting its direction and connecting the input buffer. Both can be done with the appropriate PIN_CNF register.
        - Be careful to pay attention to ALL of the fields of PIN_CNF and make sure you're setting them correctly.
            - The PULL, DRIVE, and SENSE fields can all be left as their default zero values
            - You'll need to configure both DIR and INPUT
    - Each GPIO pin number is a combination of Port `(0 or 1) << 5` and pin number (0 to 31)
        - You'll need to determine which struct pointer to use based on the port
    - To set individual pins, you'll need to use bit masks using a combination of the **&**, **|**, and **~** operators https://www.arduino.cc/en/Tutorial/Foundations/BitMask
- On boot, turn on all the LEDs on the back of the board
    - You'll write code in `main.c` to do this
    - To do so, you'll need to set 10 different GPIO pins to be outputs
    - Then you'll need to configure the output value:
        - Set (high) all the rows
        - Clear (low) all the columns
    - The relevant pins are:
        - ROW1 - P0.21
        - ROW2 - P0.22
        - ROW3 - P0.15
        - ROW4 - P0.24
        - ROW5 - P0.19
        - COL1 - P0.28
        - COL2 - P0.11
        - COL3 - P0.31
        - COL4 - P1.05 (i.e., pin number 37)
        - COL5 - P0.30
    - Hint: if the fourth column fails to turn on (but everything else works) then you probably aren't handling Port 1 correctly
- **Checkoff**: demonstrate that you can turn on all the LEDs on the back of the Microbit
    - Also show your code in main.c

13. Control LED with buttons

- Use Button A and Button B to control the Microphone LED. One should turn the LED on and the other should turn the LED off
    - Use your GPIO library to read the buttons and control the LED
    - Button A is P0.14 and is active low
    - Button B is P0.23 and is active low
    - If code isn't working, it's time to debug your GPIO library
        - Are the MMIO registers mapped to addresses correctly?
        - Are there additional fields that you do need to write to?

        - Are there additional fields that you shouldn't be writing to but are?
- **Checkoff**: demonstrate your working application to the course staff
    - Also show your code in main.c and gpio.c
    - *Question:* how do you handle both GPIO ports?

Heads up: this lab can't be used in your final projects. To give you access to raw Interrupts and MMIO, we've removed the SDK libraries that you will want to use. So this lab is really stand-alone for learning purposes.