Lab 5 - Audio and Servo

Goals

- Interact with audio on the Microbit
- Play arbitrary tones using PWM
- Control a servo using PWM
- Play analog waveforms using PWM

Equipment

- Computer with build environment
- Micro:bit and USB cable
- Servo and Microbit breakout

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/UNi8tVNT

Table of Contents

Lab 5 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: Audio**
    a. Demonstrate the `pwm_square_tone` code and application
    b. Demonstrate the `pwm_servo` code and application
    c. Demonstrate the `pwm_sine_tone` code and application
    d. Demonstrate the `record_and_play` code and application
    e. Check your understanding

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 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 lab5 <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 lab5`
        - This gets the most recent commits from the new remote source
    - `git checkout lab5/main`
        - This changes your current commit to the remote source's main branch
    - `git switch -c lab5-code`
        - This makes a new branch for your lab code
    - `git push -u lab5 lab5-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 lab5`
    - `git switch lab5-code`
- **BOTH STUDENTS** should do this
    - `git submodule update --init --recursive`
        - Makes sure all git submodules are initialized and updated

Part 2: PWM Control

Advance warning: more than any of the other labs, this lab is **conceptually** difficult, rather than a lot of code. In fact, you won't write that many lines for the entire lab, and there are only a few checkoffs. So, **what's really really important for this lab is attention to detail**. It's very easy to miss a step and have the whole system not work. So be careful with the instructions, and make sure you understand what's going on (and ask if you don't).

1. Play square wave tones over speaker with PWM

- cd into `software/apps/pwm_square_tone/` You'll be editing `main.c` here

- Initialize the PWM peripheral

  Use the `nrfx_pwm_init()` within the `pwm_init()` function

  - [Documentation for the nrfx_pwm driver]()

  - You will need to make an `nrfx_pwm_config_t` local variable to pass into the initialize function
    - `output_pins` is an array of four GPIO pin numbers, use SPEAKER_OUT for the speaker pin and NRFX_PWM_PIN_NOT_USED for unused pins
    - `base_clock` should be 500 kHz
    - `count_mode` should be Up
    - `top_value` is the default value for `COUNTERTOP` which we will overwrite later (pick anything for now)
    - `load_mode` should be Common
    - `step_mode` should be Auto

  - We don't care about callback events for the PWM, so passing in `NULL` for the callback is fine

- Play a tone using the PWM peripheral

  To do so, you'll need to fill in the code for `play_tone()` and call it from `main()`

  - To stop the PWM, use `nrfx_pwm_stop()`

  - Use `NRF_PWM0->COUNTERTOP` to access the `COUNTERTOP` register
    - `COUNTERTOP` is the number of ticks in a PWM square wave cycle (ticks/cycle) so you'll pick a value based on the PWM input clock

(ticks/second) and the desired output frequency (cycles/second)

- The PWM sequence data array has already been created for you globally. You will need to edit the value in order to set a 25% duty cycle
  - Take a look at `sequence_data` at the top of the file
  - Note: you don't care if it's left or right alignment here, just set a value so that the toggle occurs at 25% of the way counting up to `COUNTERTOP`
  - 25% will make it audible but quiet. Going to 50% will make it louder.

- To start the PWM, use `nrfx_pwm_simple_playback()`
  - The instance is defined for you at the top of the file
  - You'll want to give it a flag so it plays the note indefinitely.

- Here is a list of [frequencies for the tones of a piano]

- Play multiple tones to form music

  - You can play whatever you want as long as it consists of at least four distinct notes, so feel free to play a short jingle or song

  - If you don't have any particular music in mind, the comments in main right now guide you through the A major arpeggio scale (A_4, $C\#_5$, E_5, A_5), which is sufficient

  - You almost certainly want to stop the PWM after the last note so it doesn't continue playing it forever

  - Note that `nrfx_pwm_simple_playback()` is non-blocking. It doesn't wait for the note to finish (the note will play forever anyways), it just sets it up and returns. So you might want to delay some time before changing the note.

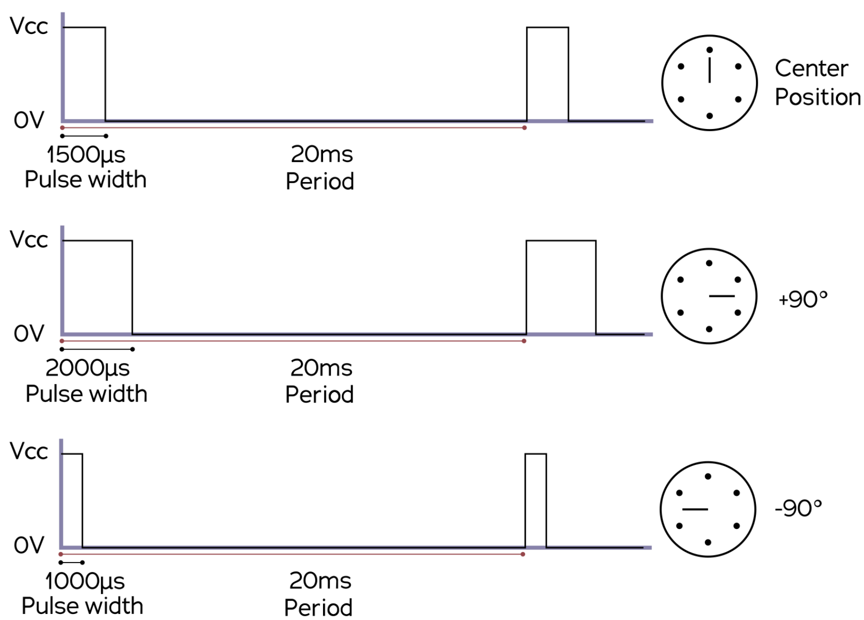
- **CHECKOFF**: demonstrate your code and app to course staff

  - If it is too quiet to hear, you might have to bump duty cycle back up to 50% for the demonstration
  - *Question:* how did you determine COUNTERTOP, and what are the units of each part of that equation?

2. Control a Servo with PWM

For this section, we're going to switch things up for a bit and control a servo using PWM. This isn't audio like the sections before and after, but should be a smoother transition into using the PWM as an analog output (rather than as a waveform like the previous section did).

Here's how a servo works: we feed it a constant PWM signal and it holds a specific position. When the PWM duty cycle changes, the servo will change its angle to match it. Here's a great diagram from Wikipedia: (https://en.wikipedia.org/wiki/Servo_control) Although the exact numbers vary a little for our specific servo.



Details for our servo: SG92R

- Pinout:
  - Brown: Ground
  - Red: VCC (3.3 Volts)
  - Yellow: Signal, connection to EDGE_P0
- PWM Signal requirements:
  - Frequency: 50 Hz (20 ms period)
  - Duration mapping:
    - 2.250 ms high -> 180 degrees
    - 1.375 ms high -> 90 degrees
    - 0.500 ms high -> 0 degrees

- For this lab you'll need a servo and a Microbit breakout to attach it to
    - You won't need a breadboard though, as we can hook the servo up directly to the breakout
    - Come grab it from the course staff

- Attach the servo to the breadboard breakout so the Brown wire connects to ground (GND), the Red wire power (3v3), and the Yellow wire connects to EDGE_P0 (pin 0 on the breakout)
    - Double-check that you did this right before connecting the Microbit to it. That way you don't damage the servo

- Your code will be in the `pwm_servo` directory. Inside `main.c` you'll need to implement `pwm_init()`, `set_servo()`, and `main()`.
    - You can and should copy code from the `pwm_tone()` application to do this
    - Don't forget that you'll need to modify some parts of the configuration though!

    - To implement `set_servo()`, the comments lead you through a series of transformations. Angle -> Duration -> Duty Cycle -> COMP value

    - A common mistake: the PWM output MUST be left-aligned to control a servo (it should start high and then later go low). To do this, you'll need to set the most-significant bit of the `sequence_data` value to 1. The lower 15 bits should still be your COMP value.

- To test this code, call `set_servo()` from `main()` with a specific angle. You can try changing this angle and reprogramming the board to see if the servo moves.
    - If it doesn't work:
        - Check that you did the left-aligned thing right
        - If it makes noise, you're likely outputting to the speaker instead of the servo pin

- In `main()` implement an application which sweeps the servo from 0 to 180 degrees by moving 45 degrees per second.

- **CHECKOFF**: demonstrate your code and working app to course staff

- **CHECKOFF**: Return your servo and breakout

3. Play sine wave tones over speaker with PWM

Differently in this part (and the next), we will now play samples from a buffer at a constant data rate. This is closer to how a DAC would work: there is some rate that analog samples are being produced at, with the analog value changing each time. We had to pick some sampling rate, and choose 16 kHz which is fast enough to handle *most* audible frequencies.

- cd into `software/apps/pwm_sine_tone/` You'll be editing `main.c` here

- The sine wave computation has been provided for you

  Look through the code already in the file. You shouldn't have to change any of this, but you will need to use the `sine_buffer` for creating your own output samples.

  It takes about 50 μs to calculate a single sine wave value. So for large buffer values, this can take a noticeable amount of a second to finish. We calculate the sine wave once at boot to make this more efficient, and then reuse the sine data to create the samples for our tones.

- Initialize the PWM peripheral

  Edit the function `pwm_init()` to do so

  - This will be almost the same as the initialization from the previous part of the lab except that the PWM clock should be set to 16 MHz

  - Also you will need to actually set a COUNTERTOP value here
    - In this example we have a constant frequency that we're playing PMW periods at, which is defined as the `SAMPLING_FREQUENCY`
    - For each data sample period, two PWM periods should play
    - We will later set the value of repeats so each PWM period is played twice

- Enable playing sine wave data over the speaker using PWM

  Edit `play_note()` to do so

  - To create each sample, you'll step cumulatively through the sine wave buffer and pull out the current sample value
    - Imagine a buffer of 10 samples of a sine wave for the entire sine period of 2π radians. If you want to go twice as fast as this, you step by twos and use every other sample.

        - Here, we have 500 samples for a single sine wave period, and we need to step through them to create our own output samples based on the desired note frequency. The step value has been provided for you
        - Iterate through the `samples` buffer copying over values from `sine_buffer`
        - Make sure to wrap your cumulative steps back around if it would go past the SINE_BUFFER_SIZE.
            - Don't set it to zero though! It should be a type of modulus operation (except that you can't actually modulus floats)
            - Setting it to zero would be a discontinuity in the sine wave output, which will sound sort of like a click or wobble in the sound.

    - Next, create the PWM sequence. You can copy some of this from the previous part of the lab.
        - You should set repeats to 1, for a total of two PWM periods per sample
        - Note that the value is the number of repetitions after the first play (so the total number of times a sample is played will be `1+repeats`)

    - Finally, start the playback. You should loop the playback forever with the proper flag

- Pass in a maximum scale value to `compute_sine_wave()` in `main()`
    - The sine wave values range from 0 to `max_value`
    - Use COUNTERTOP as the maximum value to make your life simpler, as it means the sample data will already be portions of COUNTERTOP

- Play multiple tones to form music

    - The easiest choice here is to do whatever you did for the previous part of lab

    - **NOTE:** this is going to be very quiet. That's okay and expected!

    - You again almost certainly want to stop the PWM after the last note so it doesn't continue playing it forever

- **CHECKOFF**: demonstrate your code and app to course staff

4. Record audio and play it back over the speaker

- cd into `software/apps/record_and_play/` You'll be editing `main.c` here

- The ADC side of the application has been provided for you

  Look through the code already in the file. You shouldn't have to change any of this, but you'll definitely have to interface with it, so understanding what it is doing will be helpful. Particularly note the sampling rate, as that will determine PWM timing.

- Initialize the PWM peripheral

  Edit the function `pwm_init()` to do so

  - This will be almost the same as the initialization from the previous part of the lab except that the PWM clock should be set to 16 MHz

  - Also you will need to actually set a COUNTERTOP value here
    - For each data sample period, one or more PWM periods should play
    - This depends on the value of repeats you choose when setting up the PWM sequence

- Play audio samples over the speaker using PWM

  Edit play_audio_samples_looped() to do so

  - First, modify each sample in place so it represents a PWM duty cycle rather than ADC counts. There is a `#define` for the maximum ADC value at the top of the file
    - Again, you don't have to worry about alignment here. Just set each value so that the toggle occurs at a duty cycle proportional to how large of an analog value you read

  - Next, create the PWM sequence. You can copy some of this from the previous part of the lab.
    - You almost certainly want a non-zero repeat count. If repeat is zero, each analog sample is represented by exactly one PWM period. Instead, we want each sample to be represented by a few periods so that there is more than a single waveform to average energy across
    - Increasing the repeat count will require modifying `COUNTERTOP` however. Otherwise your playback will be at an incorrect frequency. All repeated periods should happen within a single sample period.

        - Note that the value is the number of repetitions after the first play (so the total number of times a sample is played will be `1+repeats`)

    - Finally, start the playback. You should loop the playback forever with the proper flag

- Test the record and play application

  That should be everything needed to make the application work. You'll need to speak relatively loudly and pretty close to the Microbit in order for the output to be audible.

    - If the output is loud and the correct speed, but distorted, you may be clipping the signal (it may have reached max). Try speaking a little softer

    - If the output is slow, you likely miscalculated the `COUNTERTOP` value, leading to the output playing at a slower frequency than it was recorded

    - You might want to make it so that pressing a button stops the PWM from playing so that it doesn't keep repeating you forever

- **CHECKOFF**: demonstrate your code and app to course staff

If the instructors have a speaker setup on hand:
This app sounds WAY better if you use an external speaker. LIKE significantly better. Almost all of our problems are actually caused by the speaker on the Microbit.

- If we have an external speaker on hand for you to use, you can try out your application to see if it *really* works.

- You will need to make one change to code: have the PWM output to pin EDGE_P8.
    - You could even do this in addition to the existing SPEAKER_OUT rather than replacing it by adding it as a second output pin. That way your app works with OR without the external speaker.

- Connect your Microbit to the breadboard breakout (LED Matrix side up!!) to hear better audio quality.

5. Check your understanding

- Consider what would happen if we changed the `record_and_play` application to use a 500 kHz PWM clock, and correctly adjusted everything else in the code correspondingly including the COUNTERTOP value.

  In this case, you would almost certainly find that the audio sounds bad when played back. Why might this be?

- **CHECKOFF:** Explain what the problem most likely is. Use specific numbers.