UC Berkeley, ME100, Spring 2021
Lab 6: Analog/digital conversion and pulse-width modulation

Plan to complete these tasks during the week of March 29
Checkoff due on Gradescope by Friday April 12, 11:59pm

Introduction and objectives

The purpose of this lab is to become familiar with some features of the ESP32 that enable interfacing with the outside, analog world. In Lab 5 you used the built-in digital-to-analog converters to generate an amplitude-modulated signal. In this lab you will gain experience of using:

- The pulse-width modulation (PWM) outputs to control the apparent brightness of an LED and the tone of a loudspeaker;
- The built-in timers to deliver PWM signals with specified frequency, duty cycle, and duration;
- An external op amp as an output amplifier for a loudspeaker;

Step 1: PWM with internal LED

Pulse-width modulation is a technique whereby varying the fraction of time for which a pulsing binary voltage signal is high creates the effect of being able to vary an output continuously. This effect is particularly useful for varying, e.g., the apparent brightness of a light source or the warmth of a heater. If the pulsing signal is so fast that its individual oscillations cannot be discerned by the user, the output device appears to be continuously on but with a variable intensity.

First you will become familiar with the PWM syntax by using it to vary the brightness of the built-in LED.

- Configure the board LED pin as a standard output, `led`, as you did in `blink.py` in Lab 1.
- Initialize PWM timer 0 for the Pin `led` with frequency 500 Hz and 50% duty cycle:

  ```
  L1 = PWM(led,freq=500,duty=50,timer=0)
  ```

- Confirm that by varying the duty cycle between 0 and 100 (e.g. with the command `L1.duty(5)`), you can get the LED to go from appearing fully off to fully on.

Why is the PWM approach better than varying the voltage across the LED to vary the brightness continuously?

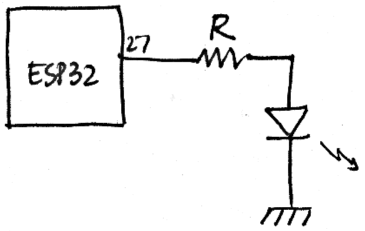
- Now add a second timer and callback function to increase the LED brightness from 0 to 100% of its maximum value linearly over a period of 10 seconds, then go back to a value of zero and repeat. Skeleton code to get you started is on bCourses. You will need to alter the period of the timer to achieve intensity variation over 10 seconds (what are the units for specifying the timer period?)
- You may find that increasing the duty cycle linearly with time does not give a particularly “linear” feel, perhaps because the eye’s response to intensity is nonlinear. If you like, you could experiment with increasing duty cycle nonlinearly with time to improve the effect.

You could conceivably just have written a loop to vary the brightness in a series of steps. Using a timer and callback function instead has the advantage that it frees up the ESP32 to run other code simultaneously, while the LED will continue to pulsate at a predictable rate.

Step 2: PWM with external LED

Now we will drive an external LED with varying apparent brightness. Any pin that can be configured as an output can be used as a PWM pin (GPIO pins 34–39, which are input-only, are excluded). An example of a pin you can use is GPIO pin 27, which is pin 23 on the development board and called pin A10 in Micropython.

Install your LED with a series resistance between the output pin and ground as shown below. Select the resistance *R* so that the current through the LED does not exceed its rated value or the maximum source current of the ESP32 pin. For example, the red LED in your kit has a rated forward current of 30 mA and a typical forward voltage of 1.7–2.2 V. Meanwhile, the ESP32 datasheet (Table 13, p 36) shows a maximum source current of 40 mA and sink current of 28 mA for GPIO pin 27. Allow a safety factor so that the current does not exceed, say, 15 mA for 1.7 V across the LED.



What value of *R* did you select?

Update your code so that the external LED, instead of the board LED, is varied in apparent intensity by the PWM signal, run the code, and capture a video of the LED brightness varying. Upload this version of your code to Gradescope.

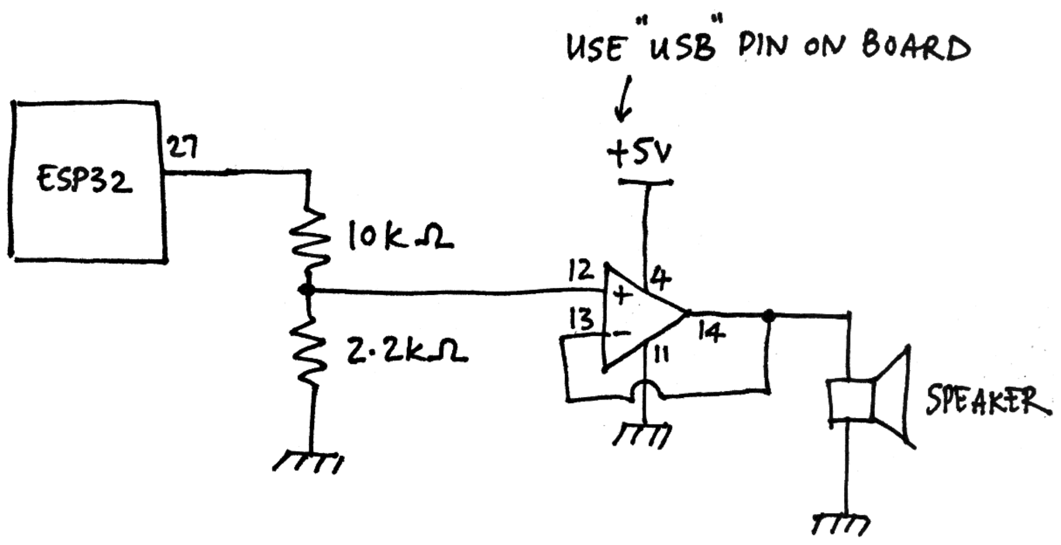
Also connect your oscilloscope between your output pin and ground, and capture an image of the voltage waveform, showing the pulses of the output voltage. The "Auto" button may be helpful for finding a suitable time base.

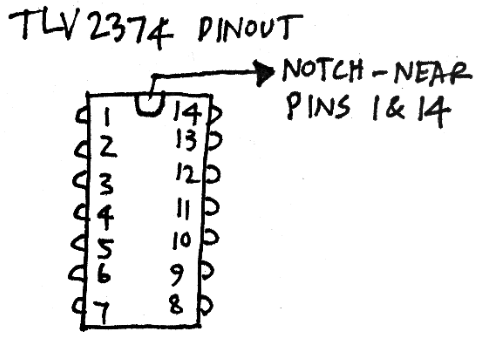
Step 3: Connecting audio amplifier and speaker

In the next steps, we will take advantage of the ability to vary the frequency of a PWM output signal. We will vary the frequency to generate a tune on a loudspeaker.

First, we need to set up an amplifier to drive the speaker contained in your kit. The resistance of this speaker is 8 ohms and it is rated 1W, which means that if we attempted to drive it directly from the ESP32 it would either likely be too quiet or we would exceed the rated current output of the ESP32 pin.

One option is to use the TLV2734IN operational amplifier in your kit. Its absolute maximum output current is 100 mA, which is only a factor of 2.5 improvement over the 40 mA source current of the ESP32, but it provides some improvement. You can set it up in a simple voltage-follower mode as follows (op amp circuits will be discussed in lecture this week):



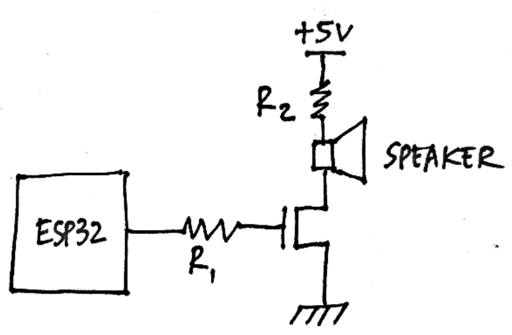


Note: in the diagram above, the numbers surrounding the op amp symbol represent pin numbers on the integrated circuit package. The pinout diagram below the circuit shows how these pin numbers are defined. This is an example of a "dual inline" package (DIP). A notch in the plastic package indicates the end of the package that is closest to pins 1 and 14. This notch convention is used throughout DIP integrated circuits. A datasheet for the TLV273* series is on bCourses.

A voltage follower outputs the same voltage as is placed on the non-inverting input, marked "+". Analyze this circuit to work out the peak current that will pass through the speaker. Is this current within the specs of the op amp?

Test this circuit by running the LED PWM code from above and verify that an audible signal is heard from the speaker.

Another amplifier option, which would enable larger currents and louder sound, would be to use the discrete MOSFET included in your kit; if it is a BS170 NMOS, it is capable of passing a peak current of 500 mA and dissipating 850 mW. You could use this in place of the op amp. Select an appropriate series resistance to deliver acceptable volume without exceeding the current or power capabilities of the MOSFET:



You are not required to use the MOSFET amplifier approach if you are satisfied with the op amp circuit previously introduced.

Step 4: Playing a tune

Now you will adapt your PWM code to play a tune. Each note will be played by varying the frequency of a PWM output on GPIO pin 27. Use a timer to time the transitions between notes. A listing of the frequencies associated with different notes, and an example tune, are in Appendix 2.

Upload your code and a recording (video or audio) of your tune playing on your ESP32 to Gradescope.

Some things to experiment with:

- Try different tunes, or compose your own.
- Can you vary the quality, or timbre, of the sound in any way by varying the duty cycle or other aspects of the signal that you output?
- Vary the speed of the tune until it sounds reasonable.
- Either stop when the end of the tune is reached, or pause when you get to the end of the tune for 1–2 seconds, then repeat.

Optional extension steps

The following steps are optional (they are not required to get 5/5 on this lab), but are encouraged if and when you have time, as they will provide knowledge that could be deployed in your projects. Feel free just to do the steps that interest you most.

Step 5: Compare sine and rectangular waves on a speaker

A square wave with frequency *f* actually contains frequencies other than *f*. In fact, a train of rectangular pulses with magnitude 1, period T and length t_d (such that the duty cycle is t_d/T) can, by Fourier decomposition, be written in terms of its constituent frequencies as follows:

$$g(t) = \frac{t_d}{T}\left[1 + 2\sum_{n=1}^{\infty} \frac{\sin\left(\frac{n\pi t_d}{T}\right)}{\left(\frac{n\pi t_d}{T}\right)} \cos(n\omega_0 t)\right]$$

where $\omega_0 = 2\pi/T$. So, changing the duty cycle at a given fundamental frequency ω_0 will result in the different *harmonics* (i.e. multiples of the fundamental frequency) having different relative strengths. This effect may have been audible when you ran the LED code on your speaker in Step 3.

In this step, we will compare the sound of a pure sine tone (or as close to pure as we can get) with that of a square wave. Download the script ME100_Lab6_Step5_sine_square_comparison.py from bCourses and study it. This script writes a pure 2.5 kHz sine wave to DAC2 (pin 26 of the ESP32 chip; pin 5 of the board) for 5 seconds; then it outputs 2.5 kHz rectangular waves using the PWM feature on the same pin, looping through duty cycles of 10% to 90% in 10% steps, at 1 second per step. Listen to the output and note any changes in the sound.

In fact it is very rare for a musical note to have a pure sinusoidal tone – the mix of frequencies generated by various resonances occurring in the instrument or voice is what gives it its unique quality or timbre. The mix of frequencies in a musical note is also generally far richer than that of a rectangular wave. No doubt there are some musicians in the class who may like to explore the ability of the ESP32 to synthesize particular sounds or develop novel IoT-enabled musical instruments or tools for their project. If so, you may want to read up on the Musical Instrument Digital Interface (MIDI) standard, for which packages exist in Micropython.

Step 6: Debouncing a switch input

The bistable circuit for switch contact debouncing that was shown in the lecture on sequential logic relied on having a *double-throw* switch – i.e. when pressed, it needed to connect one of the bistable inputs to ground, and when released it needed to connect the other bistable input to ground. However, *single*-throw switches, which can make only one connection, are much more common and simpler, so it would be helpful to have a way to debounce them.

Fortunately, there is a way to do this in software. An example is in the following script on bCourses: ME100_Lab6_Step5_sine_square_comparison.py. It relies on assuming that when a contact bounces, it does not do so for more than 20 ms (this value could be changed in the code if necessary). It makes use of the XOR function (^ in Python) to see whether the input has changed since at least 20 ms ago. If a change has taken place, the output is toggled. To test this code, connect a pushbutton between Pin A9 and ground, run the code, and confirm that the LED is on when the pushbutton is released, and off when it is being pressed.

Step 7: Capacitive touch sensing

You will be familiar with capacitive touch sensors from everyday electronic objects. Many functions are now activated by capacitive touch, rather than having a mechanical switch. Examples include the power buttons on various gadgets such as computer monitors. In addition, most touchscreens rely on capacitive sensing. The principle is that when a finger or other object gets close to or touches an electrode, the capacitance seen by that electrode changes. This capacitance change can be detected and acted upon as a trigger. Capacitive touch brings benefits of cleaner design, lower cost, and lower likelihood of failure.

Ten of the ESP32's GPIO pins can be configured as capacitive touch sensing (capsense) pins (listed in Table 1 of the ESP32 datasheet). Here we will connect two of the capsense pins to home-made electrodes to sense the proximity of a finger or other object. We choose to use GPIO pins 12 and 15, which correspond to pins 24 and 21 on the development board.

The sample code in ME100_Lab6_Step7_capsense.py sets up these pins and prints the reading from each of the pins every 0.1 s. The values are integers in the range 0 to approximately 1000. I found that recorded values equilibrated about 600–700 when open-circuit. If you connect a wire to the pin and bring your finger close to the end of it without touching it, you will see that the recorded value will reduce slightly. If you actually touch the pin, you will notice that the recorded value falls to below 100. Enlarging the area of the electrode connected to the pin (I connected a wire to a stainless steel spoon) appears to make the circuit more sensitive to proximity of a hand or finger.

Step 8: Theremin

This part is purely for fun. A theremin is a musical instrument that generates a tone whose pitch and volume can be varied by moving one's hands around near two capacitive sensing electrodes. Moving one's hands further away corresponds to higher volume and higher pitch.

Combine the elements of code provided with this lab together with the code that you wrote for Step 4 to create a simple theremin. Skeleton code is at ME100_Lab6_Step8_theremin.py. It updates the frequency of a PWM signal every 0.1 s depending on the value of a capsense pin, and that signal could be fed into the amplifier and then to the speaker. Unfortunately, I found that the transition between different frequency values sounds discontinuous and does not well approximate the sound of a real theremin (as shown in the video linked in the previous paragraph). Can you think of other approaches that would make for a more continuous-sounding variation of frequency?

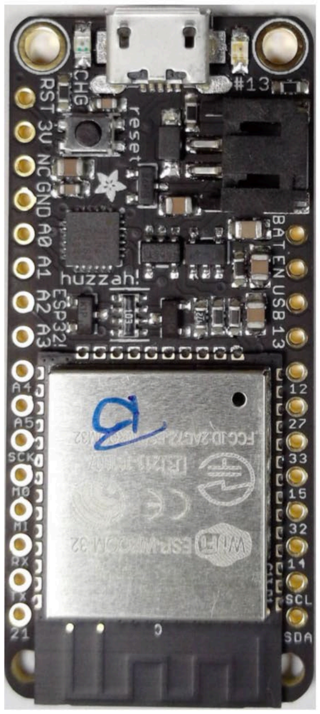
As a further step, you might turn your theremin into a true IoT device by, e.g., sending MQTT messages containing the music you have created, for sharing and later playback.

As a different approach, perhaps you could also use the accelerometer/gyroscope breakout board (more on this in later labs) to control the characteristics of sound and create a novel musical instrument.

Appendix 1: Huzzah ESP32 pinout

Adafruit HUZZAH32 MicroPython

GPIO	ALT	µPy	
RESET			1
3.3V			2
			3
GND			4
26	DAC2	A0	5
25	DAC1	A1	6
34	ADC6	A2	7
39	ADC3	A3	8
36	ADC0	A4	9
4		A5	10
5	SCK	A16	11
18	MOSI	A17	12
19	MISO	A18	13
16		A19	14
17		A20	15
21		A21	16



	µPy	ALT	GPIO
28	VBAT		
27	EN 3.3V		
26	VUSB		
25	A12	LED	13
24	A11	BOOT	12
23	A10		27
22	A9	ADC5	33
21	A8		15
20	A7	ADC4	32
19	A6		14
18	A15	SCL	22
17	A14	SDA	23

Boot mode:

BOOT (A11) has a built-in pull-down
Connect to 3.3V on power-up for safe boot.

EN 3.3V: tie to GND to disable regulator

Legend:

sup	ADC	SPI
GND	DAC	I2C
BOOT	VBAT/2	LED
input only!		
VBAT/2 tied to VBAT2		

Appendix 2: Notes and example tune

C3 = 131
CS3 = 139
D3 = 147
DS3 = 156
E3 = 165
F3 = 175
FS3 = 185
G3 = 196
GS3 = 208
A3 = 220
AS3 = 233
B3 = 247
C4 = 262
CS4 = 277
D4 = 294
DS4 = 311
E4 = 330
F4 = 349
FS4 = 370
G4 = 392
GS4 = 415
A4 = 440
AS4 = 466
B4 = 494
C5 = 523
CS5 = 554
D5 = 587
DS5 = 622
E5 = 659
F5 = 698
FS5 = 740
G5 = 784
GS5 = 831
A5_ = 880
AS5 = 932
B5 = 988
C6 = 1047
CS6 = 1109
D6 = 1175
DS6 = 1245
E6 = 1319
F6 = 1397
FS6 = 1480
G6 = 1568
GS6 = 1661
A6 = 1760
AS6 = 1865
B6 = 1976
C7 = 2093
CS7 = 2217
D7 = 2349
DS7 = 2489
E7 = 2637
F7 = 2794

```
FS7 = 2960
G7 = 3136
GS7 = 3322
A7 = 3520
AS7 = 3729
B7 = 3951
C8 = 4186
CS8 = 4435
D8 = 4699
DS8 = 4978

# Bach Prelude in C.
bach = [
C4, E4, G4, C5, E5, G4, C5, E5, C4, E4, G4, C5, E5, G4, C5, E5,
C4, D4, G4, D5, F5, G4, D5, F5, C4, D4, G4, D5, F5, G4, D5, F5,
B3, D4, G4, D5, F5, G4, D5, F5, B3, D4, G4, D5, F5, G4, D5, F5,
C4, E4, G4, C5, E5, G4, C5, E5, C4, E4, G4, C5, E5, G4, C5, E5,
C4, E4, A4, E5, A5_, A4, E5, A4, C4, E4, A4, E5, A5_, A4, E5, A4,
C4, D4, FS4, A4, D5, FS4, A4, D5, C4, D4, FS4, A4, D5, FS4, A4, D5,
B3, D4, G4, D5, G5, G4, D5, G5, B3, D4, G4, D5, G5, G4, D5, G5,
B3, C4, E4, G4, C5, E4, G4, C5, B3, C4, E4, G4, C5, E4, G4, C5,
B3, C4, E4, G4, C5, E4, G4, C5, B3, C4, E4, G4, C5, E4, G4, C5,
A3, C4, E4, G4, C5, E4, G4, C5, A3, C4, E4, G4, C5, E4, G4, C5,
D3, A3, D4, FS4, C5, D4, FS4, C5, D3, A3, D4, FS4, C5, D4, FS4, C5,
G3, B3, D4, G4, B4, D4, G4, B4, G3, B3, D4, G4, B4, D4, G4, B4
]
```