Building a real-time embedded audio sampling application with MicroPython

Alan Christie

europython

9-16 JULY 2017 Rimini

This session…

- …we'll construct a continuous “listen, repeat” audio application
- …is for beginners, but we're going to cover a lot in 25 minutes
- …will introduce…
    - MicroPython (the parts we need anyway)
    - The PyBoard
    - The PyBoard audio skin

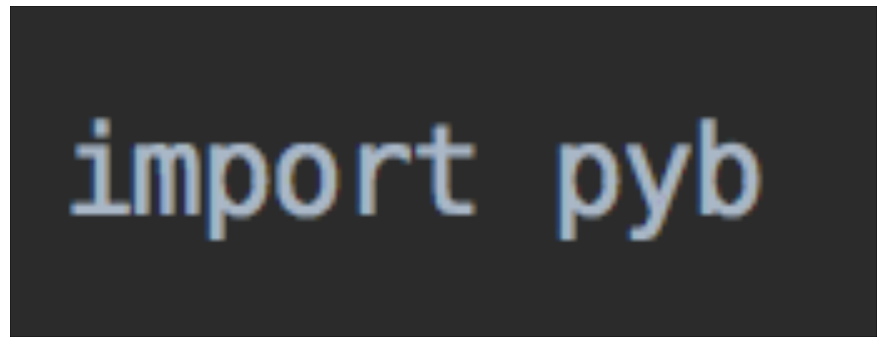
MicroPython

- Lean implementation of Python 3 (256kB)
- Optimised for micro-controllers
- Numerous modules for hardware control

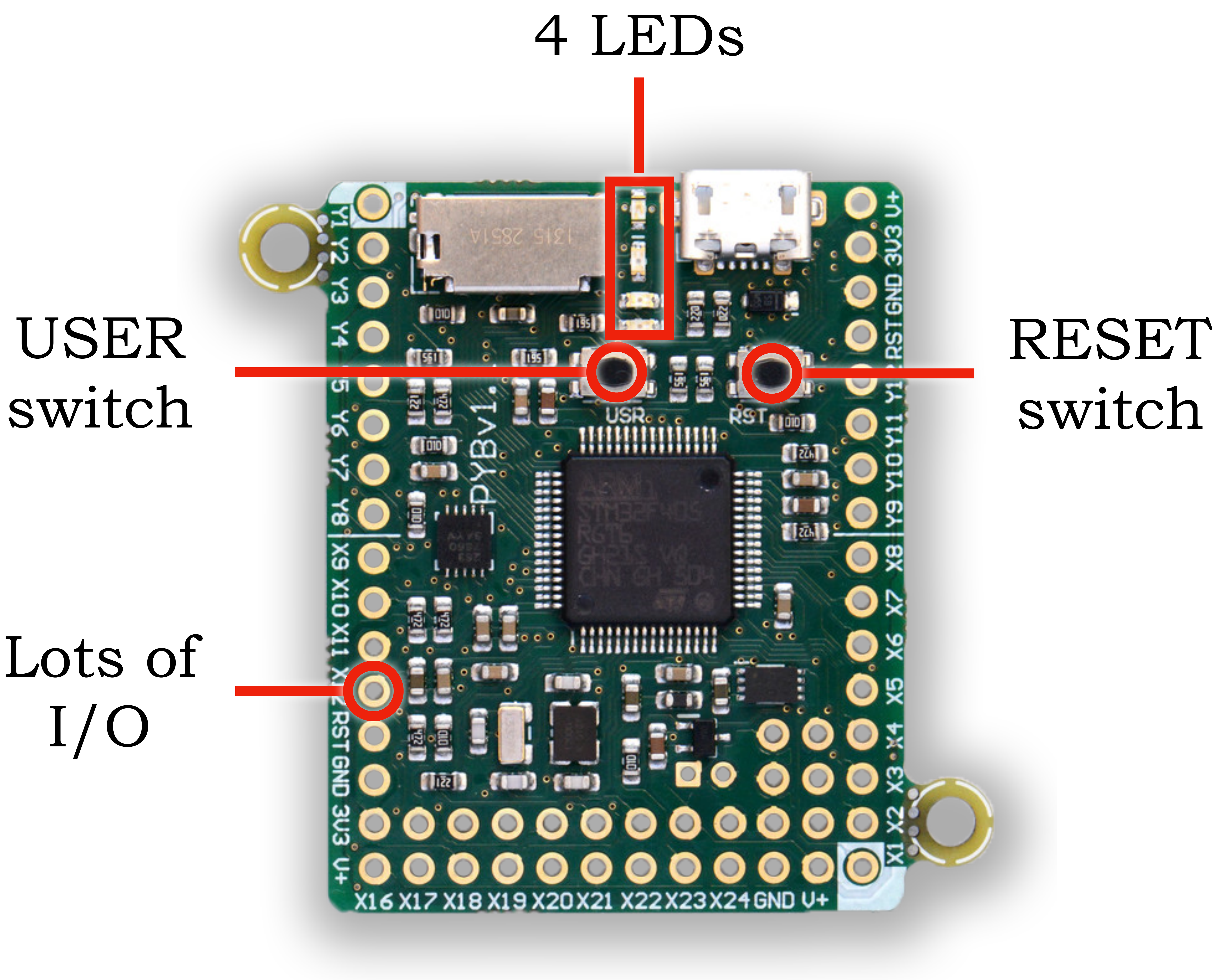
- Originally created by Australian programmer and physicist Damien George
- Backed by Kickstarter campaign in 2013

The PyBoard (v1.1)

- Just one of a number of hardware architectures
- 192K RAM (100K heap)
- 48-168MHz ARM
- 12-bit digital resolution (**DAC** & **ADC**)



```
import pyb
```



4 LEDs

USER switch

RESET switch

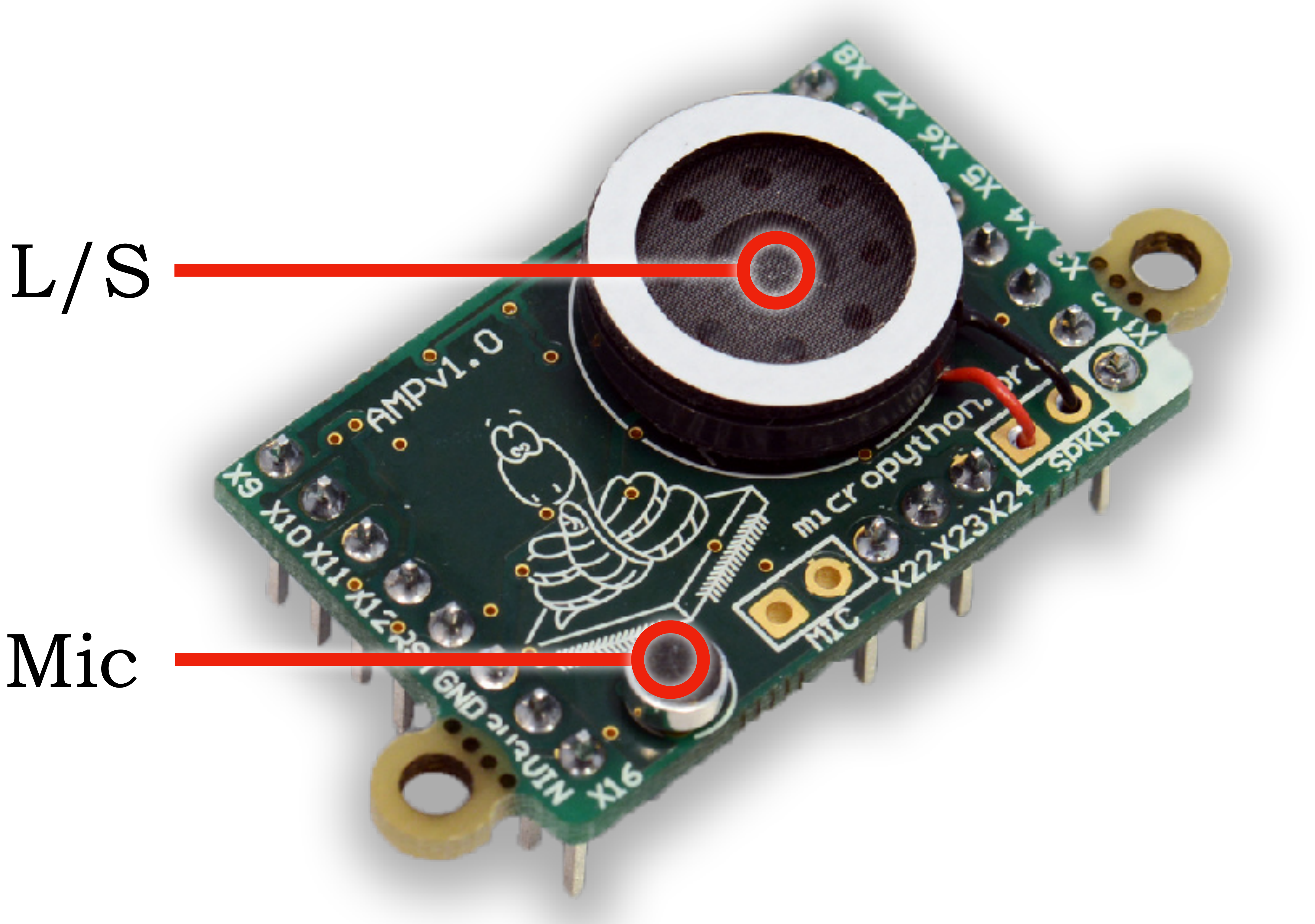
Lots of I/O

USR
RST
PYBv1.
Y1 Y2 Y3 Y4
X16 X17 X18 X19 X20 X21 X22 X23 X24 GND V+
X1 X2 X3 X4 X5 X6 X7 X8 Y9 Y10 Y11 Y12 RST GND 3V3 V+

https://store.micropython.org/#/features

The audio skin

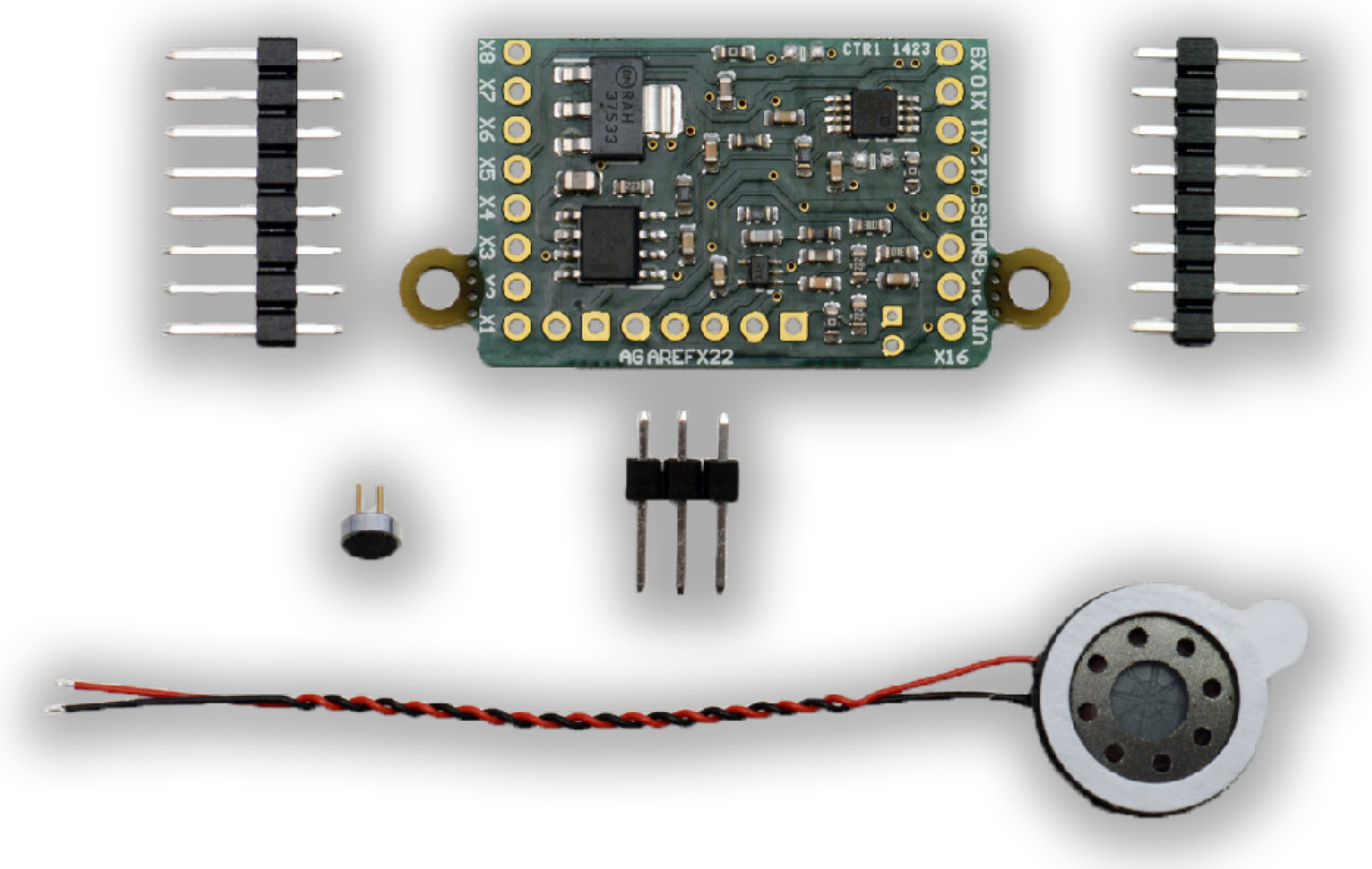
- Plugin for PyBoard
- Loudspeaker with small power amp
- Built-in microphone with pre-amp
- Option of external mic


L/S

Mic

Be prepared !

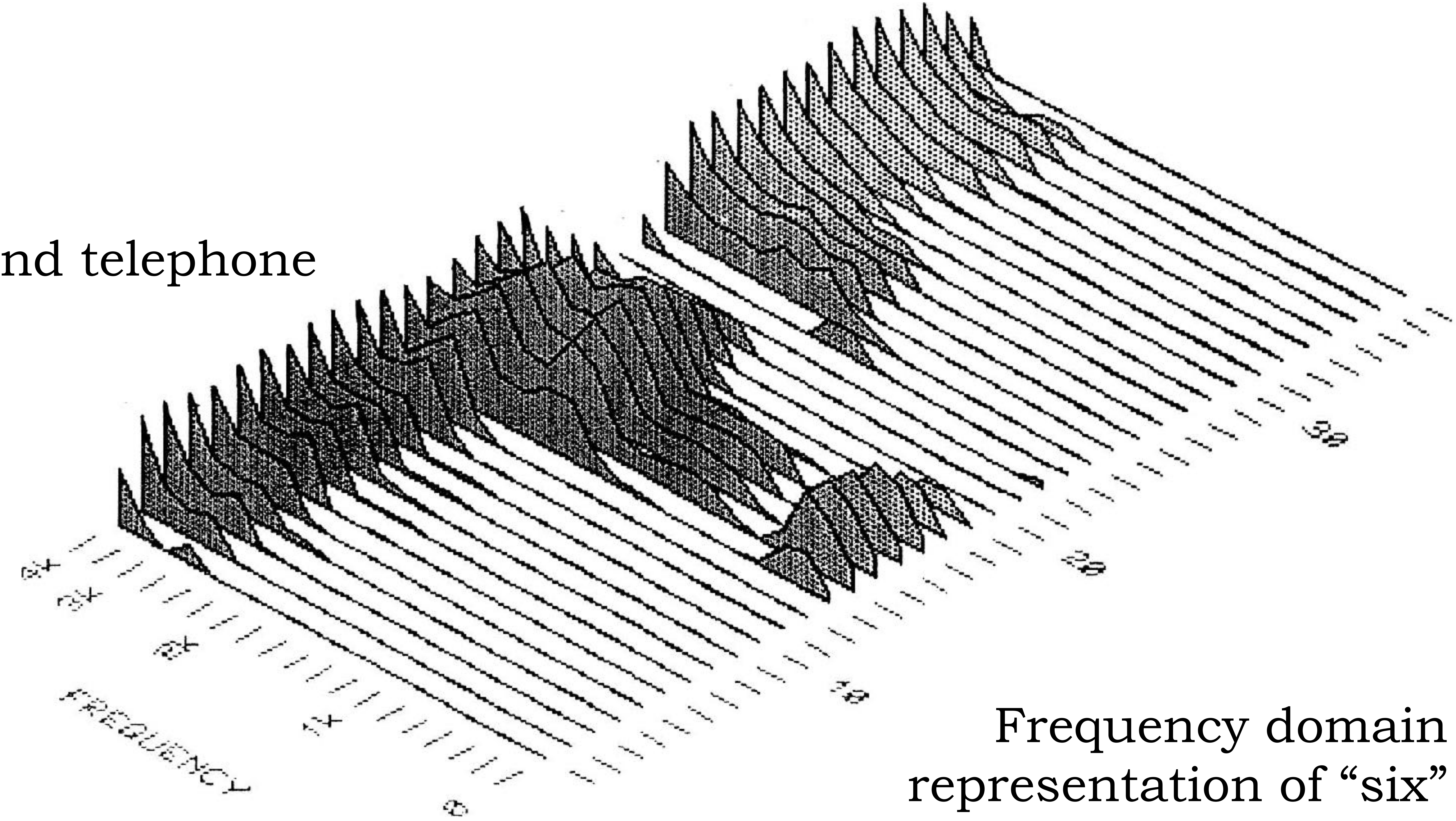
- The audio skin does require some assembly…



Speech

- Hi-fi
  14kHz

- Narrowband telephone
  3.4kHz

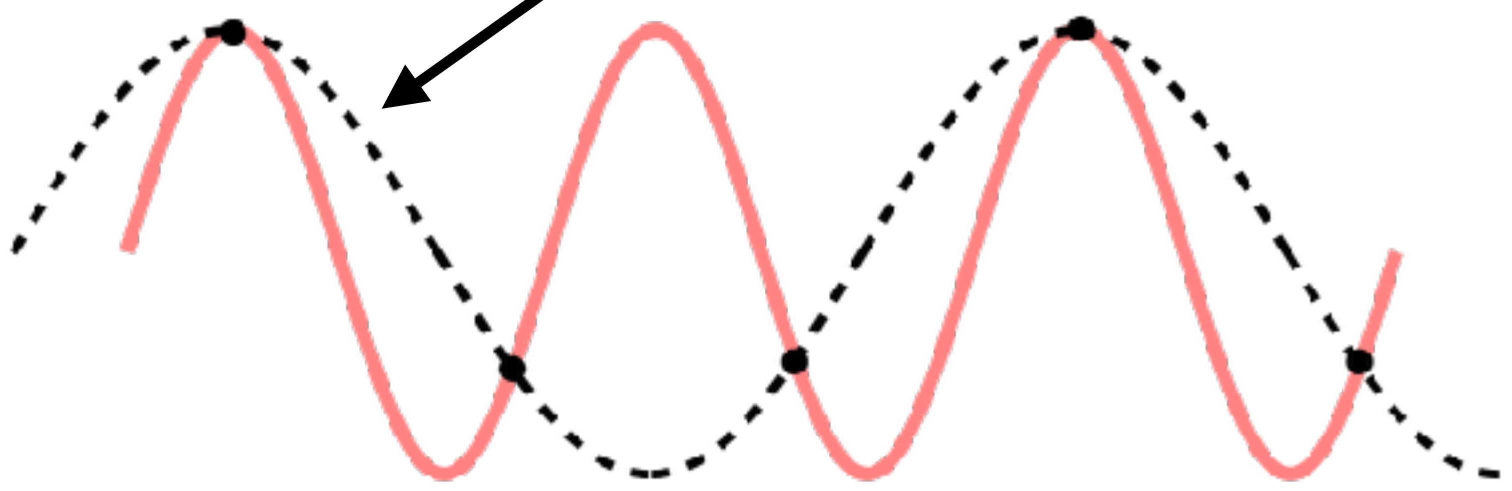


Frequency domain
representation of “six”

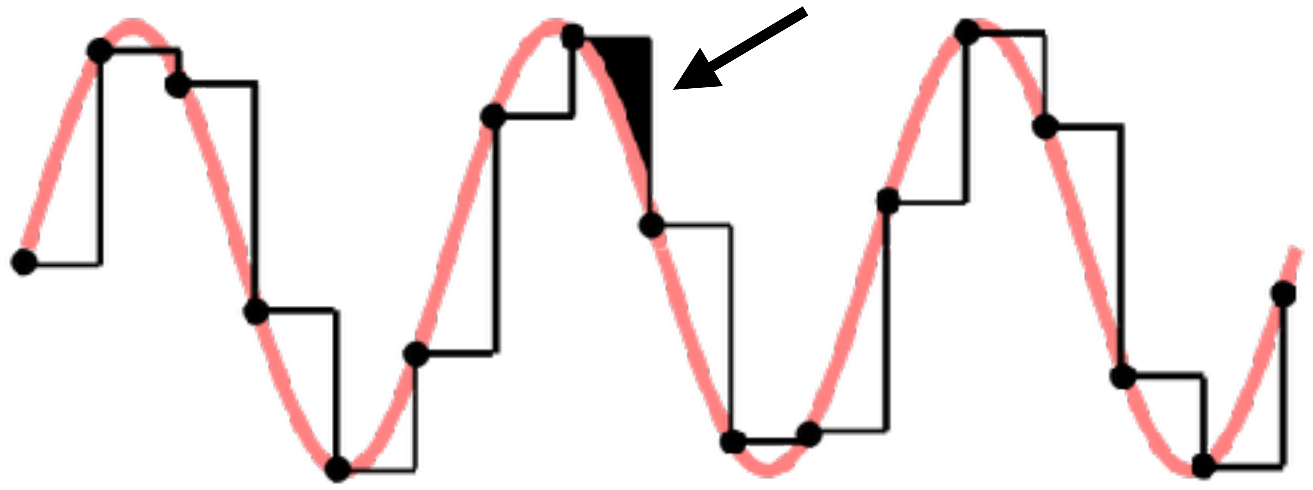
From analogue to digital (the ADC)

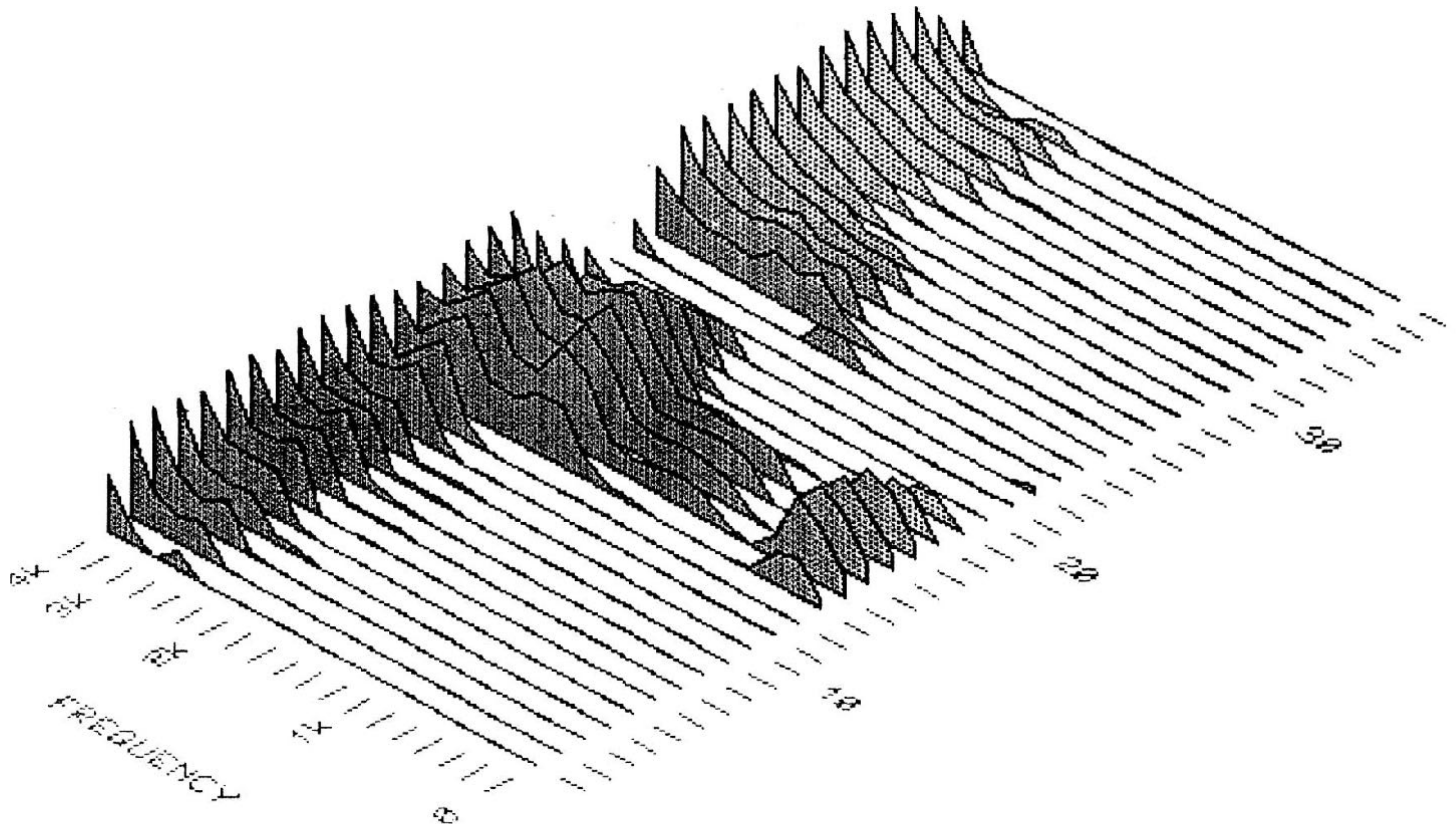
- Sampling frequency
  - To avoid “aliasing” sample at > 2x
- Resolution
  - Choice of 8 or 12 bits
  - CD is 16-bits at 44.1kHz
- Duration
  - 300-500mS per word

“Aliasing”
(Sampling at 1.5x)



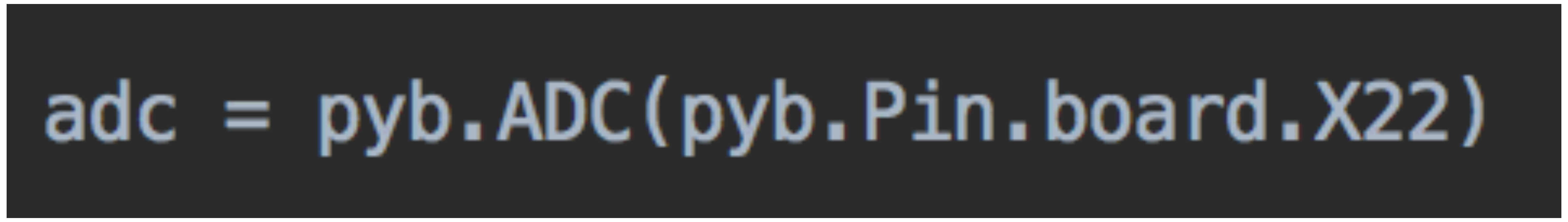
“Quantisation error”





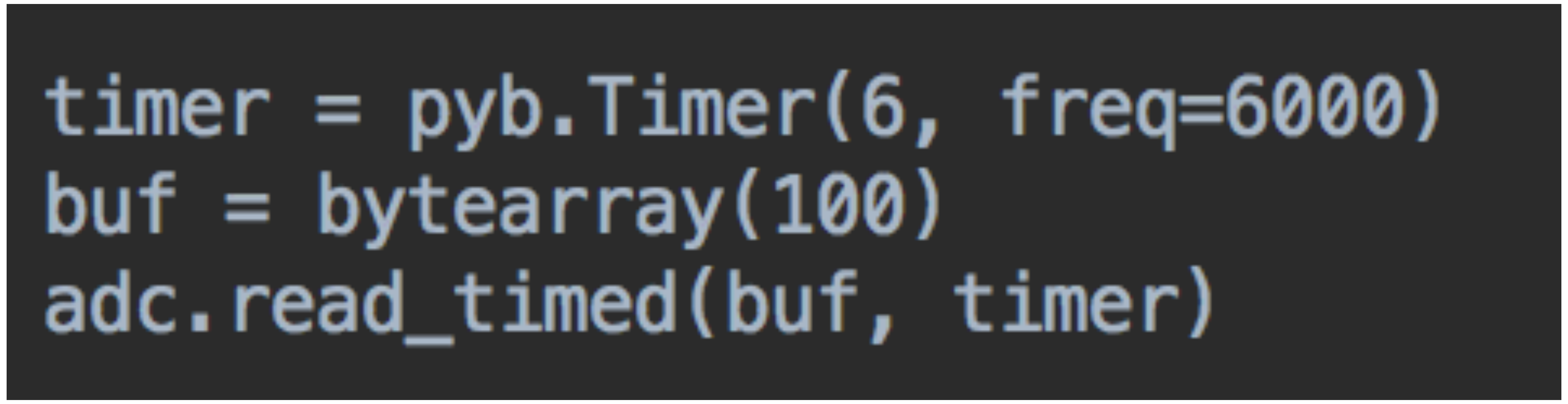
How do we record?

- Create an ADC object



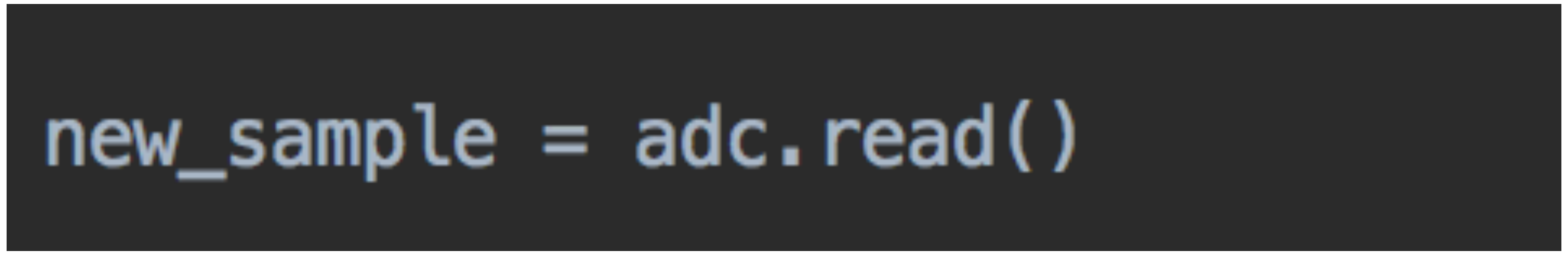
```
adc = pyb.ADC(pyb.Pin.board.X22)
```

- Then… “read_timed()”



```
timer = pyb.Timer(6, freq=6000)
buf = bytearray(100)
adc.read_timed(buf, timer)
```

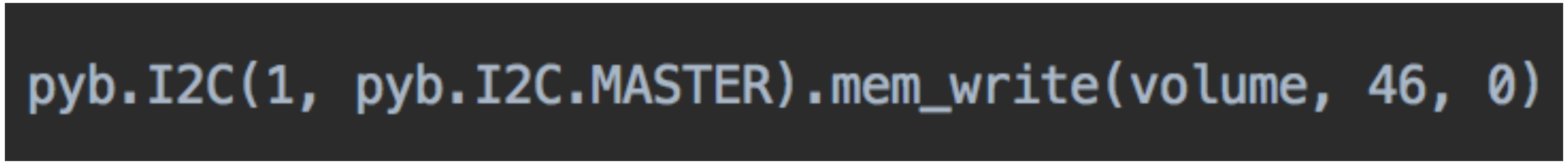
- …or “read()”



```
new_sample = adc.read()
```

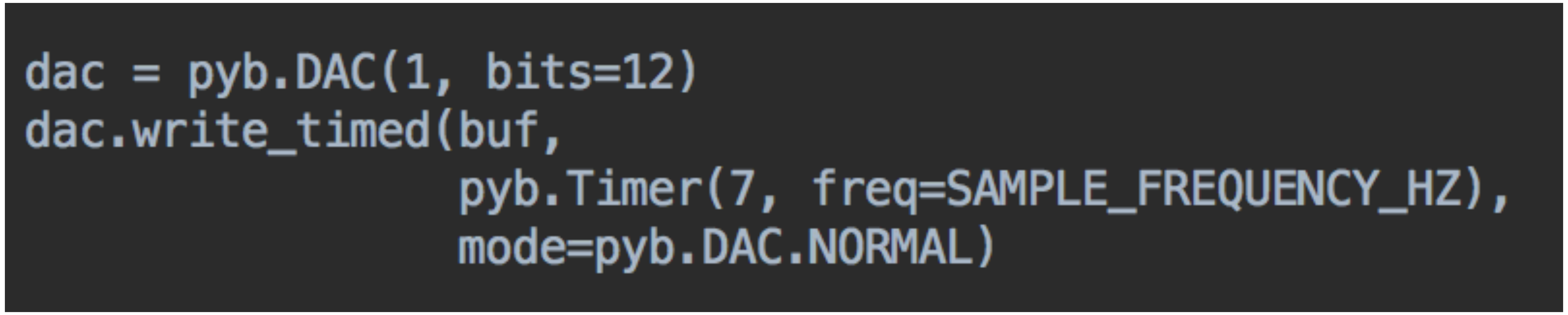
From digital to analogue (the DAC)

- Set the volume (using the digital potentiometer on the I2C bus)



```
pyb.I2C(1, pyb.I2C.MASTER).mem_write(volume, 46, 0)
```

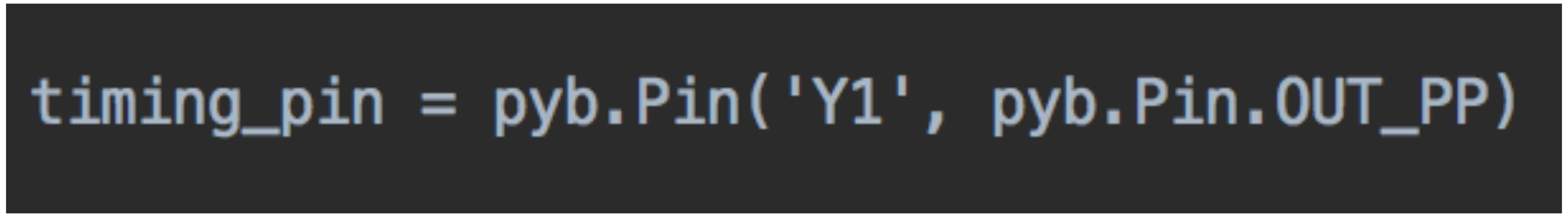
- Tell the DAC about and where to find the audio



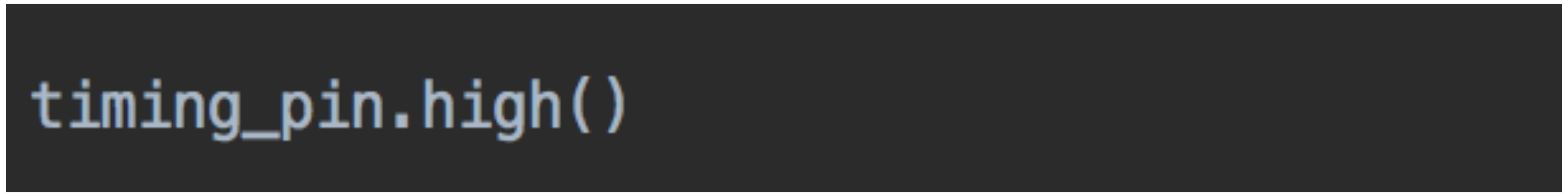
```
dac = pyb.DAC(1, bits=12)
dac.write_timed(buf,
                pyb.Timer(7, freq=SAMPLE_FREQUENCY_HZ),
                mode=pyb.DAC.NORMAL)
```

It has to be real-time - how do I know? (Pins)

- Use a pin and an oscilloscope
- Clear the pin, “do work”, set the pin



```
timing_pin = pyb.Pin('Y1', pyb.Pin.OUT_PP)
```

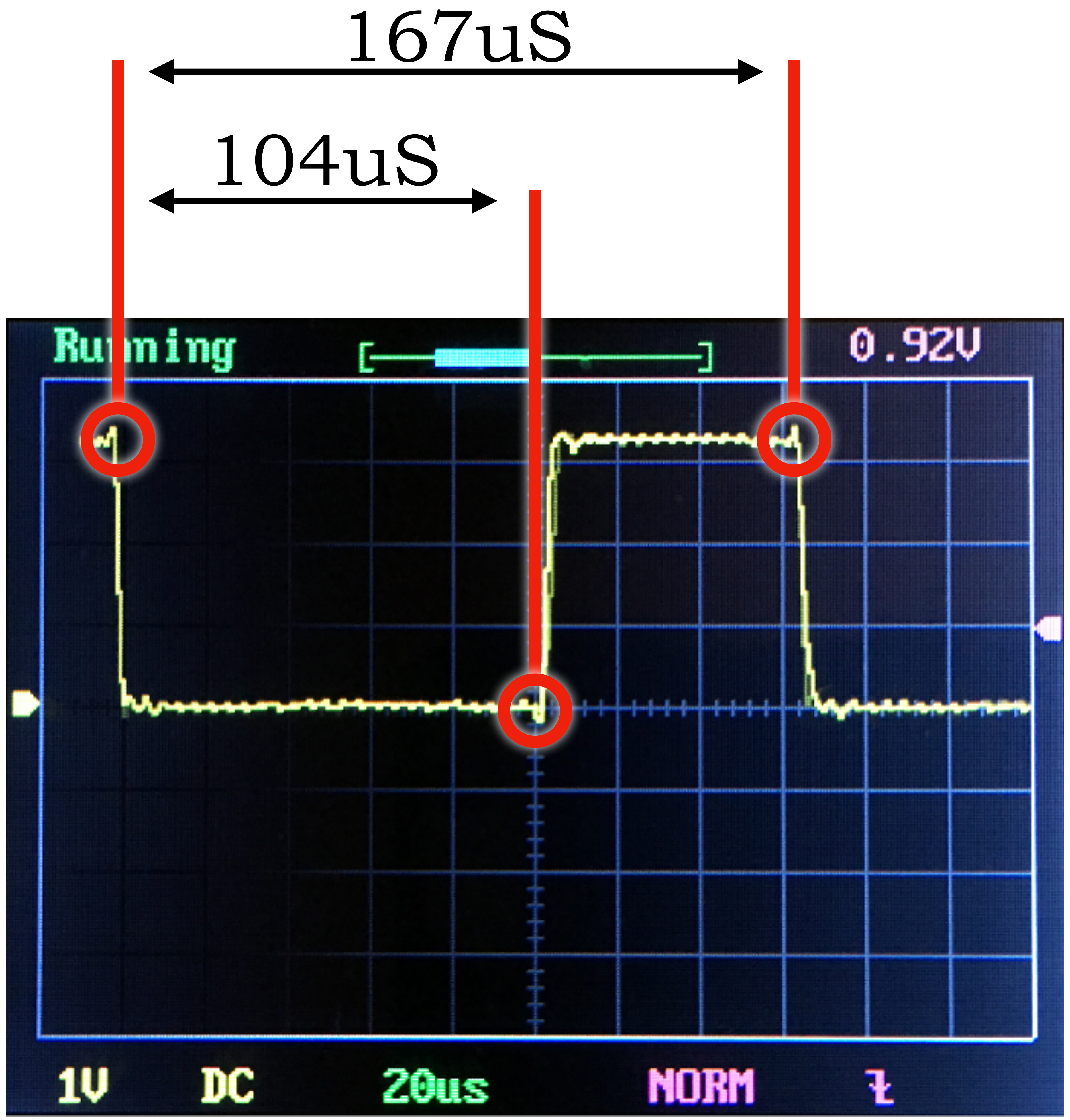


```
timing_pin.high()
```

- … or … you can use a MicroPython Timer object

Performance verification

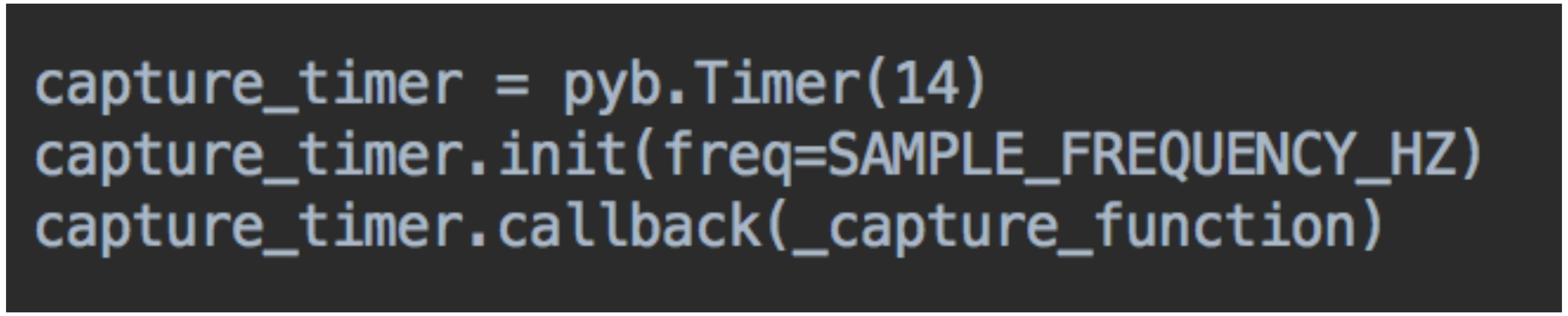
- Confirmation of 6kHz Timer (period of 167uS)
- Capture takes 104uS (9.6kHz max)
- Interestingly…
    - Clearing and reading a “Timer” takes 20uS
    - Calling “adc.read()”
      takes 50uS (20kHz upper-bound)



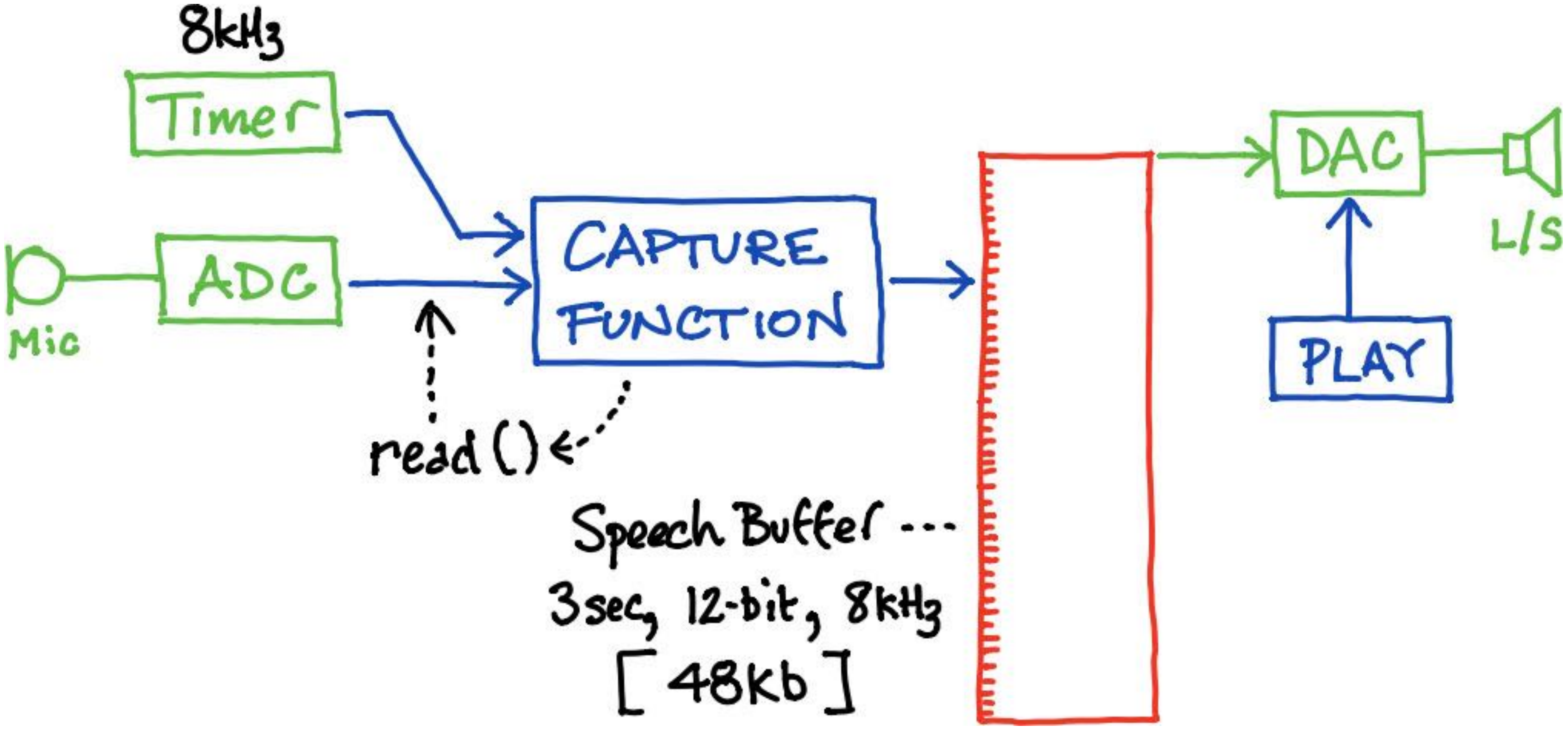
Initial setup

- Somewhere to store the captured audio (a “**sample buffer**”)
- A time-dependent “**capture function**”

  No object creation
  No Python floats
- Simple “**play**” function

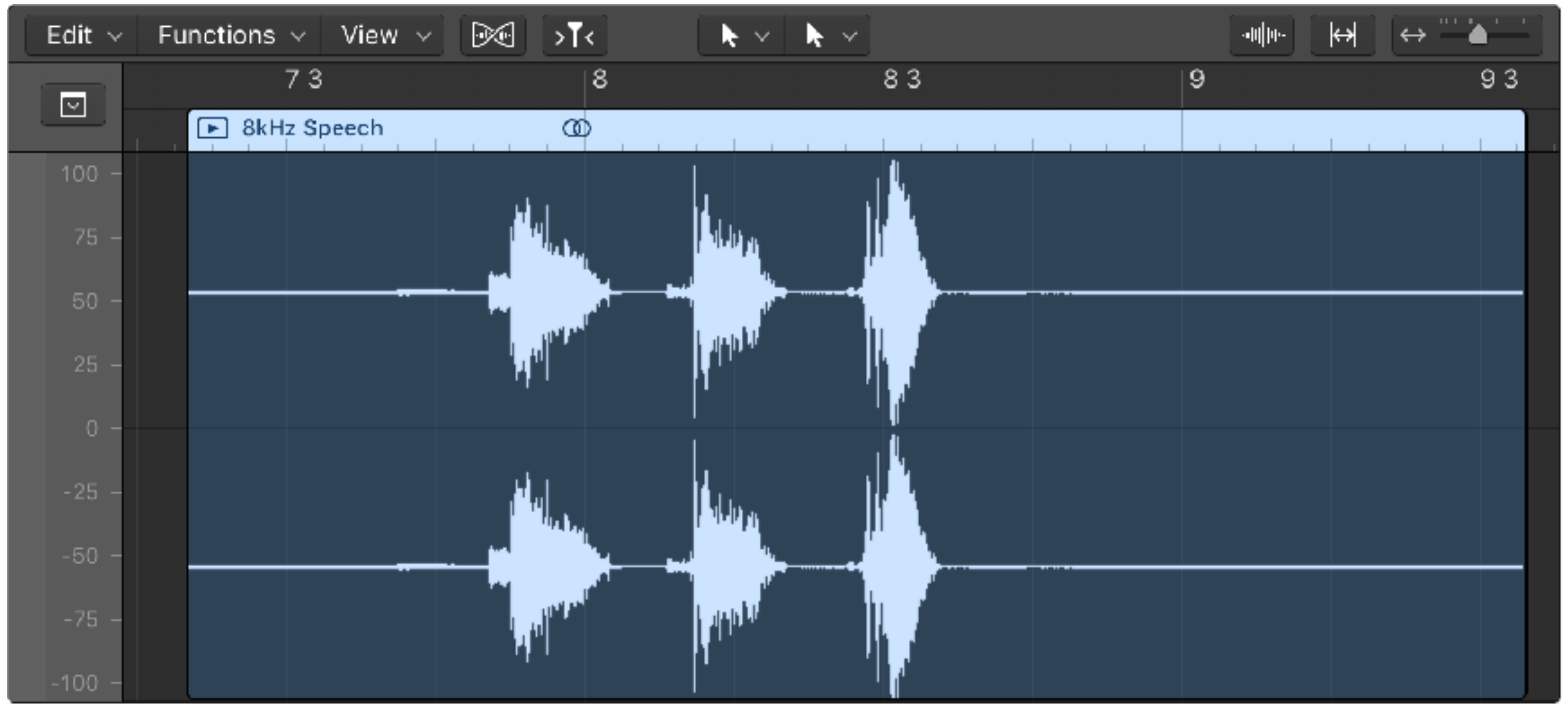


```
capture_timer = pyb.Timer(14)
capture_timer.init(freq=SAMPLE_FREQUENCY_HZ)
capture_timer.callback(_capture_function)
```



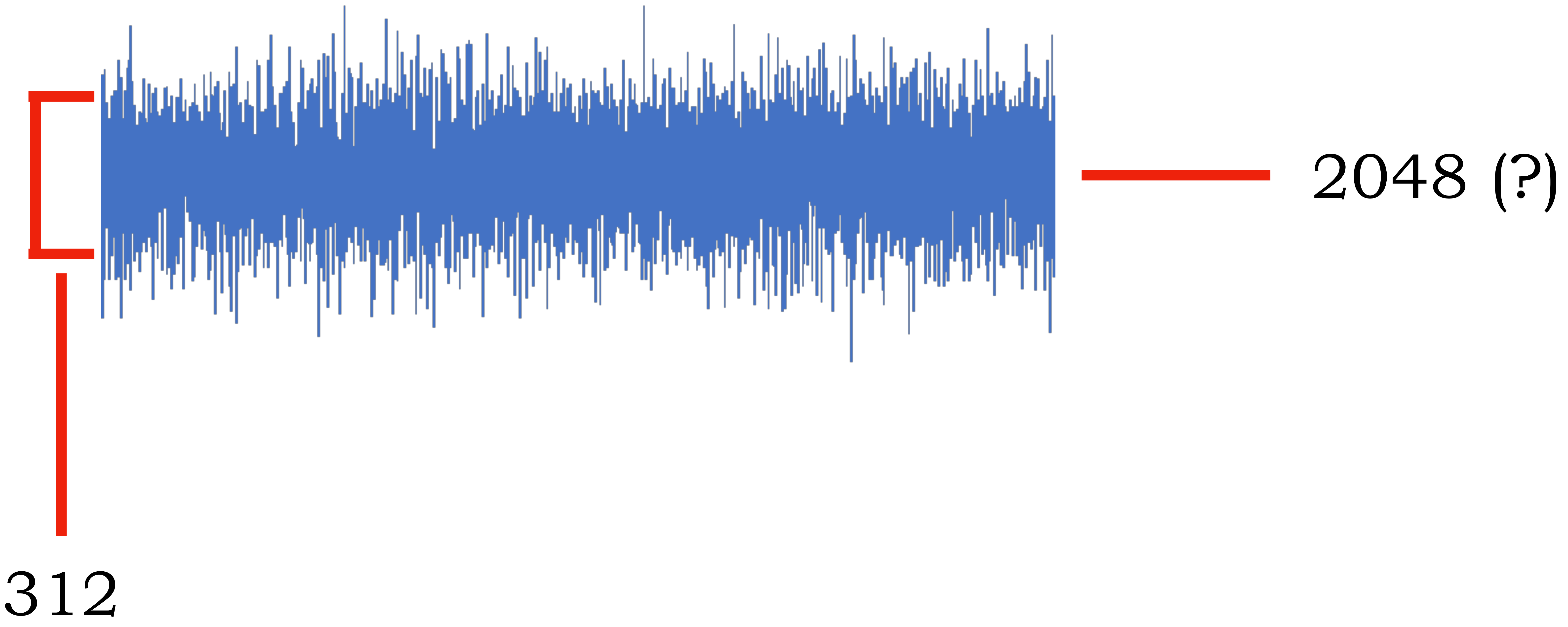
Initial recordings

- “Uno Due Tre”
  - 8kHz at 12 bits



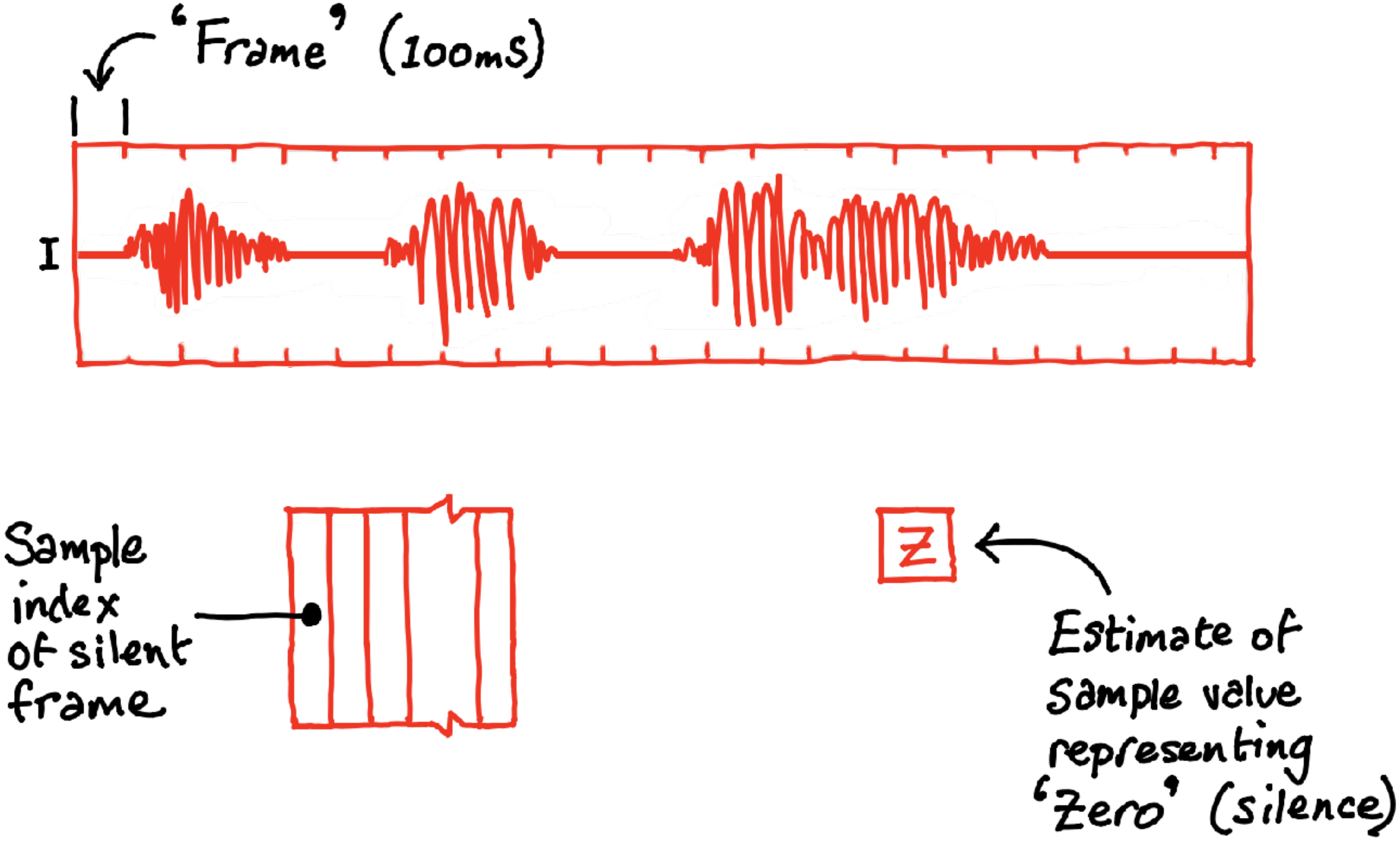
Noise

- Recording at highest resolution of 12-bits (0..4095)
- 95% of noise samples occupy 312 values around “zero”
- 8-9 bits consumed



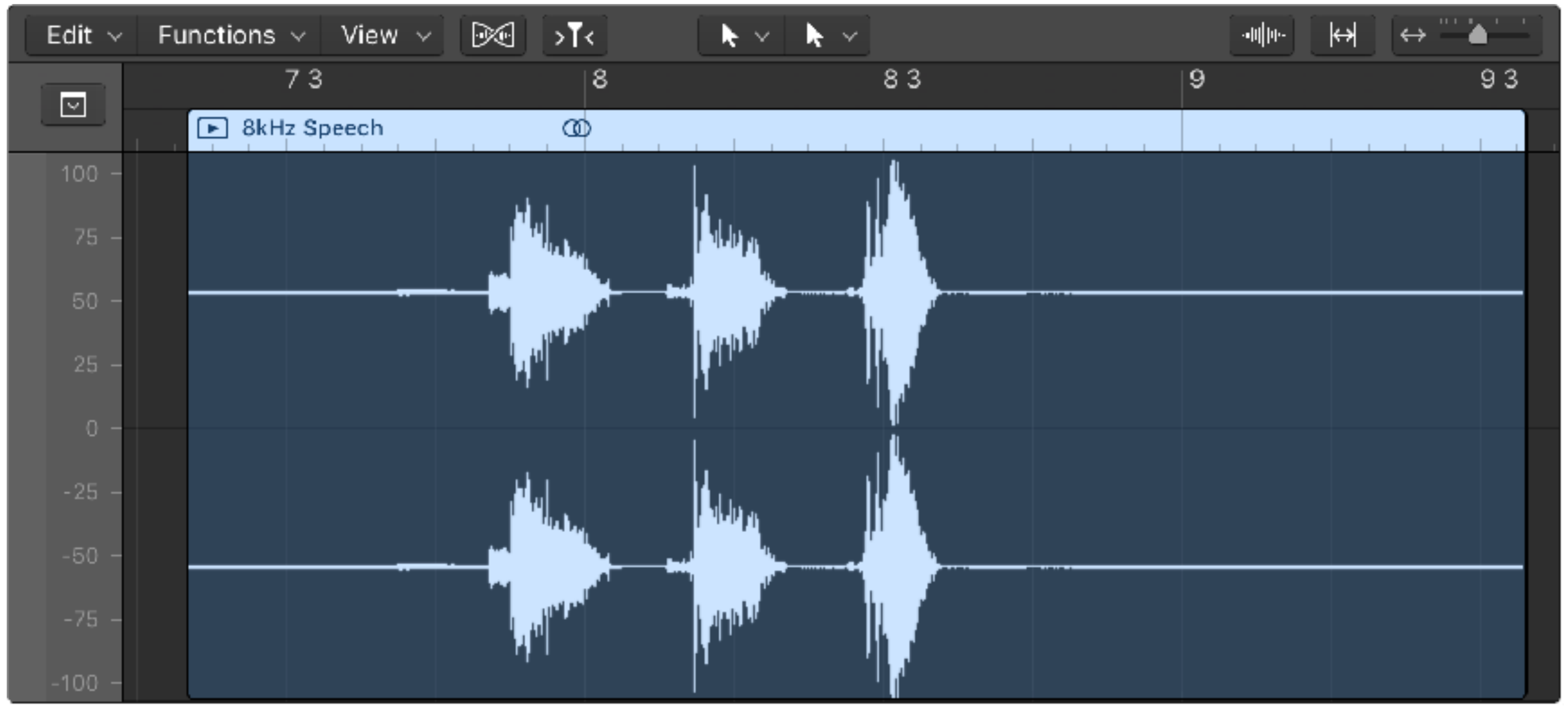
Reducing some noise (a ‘quick-win’)

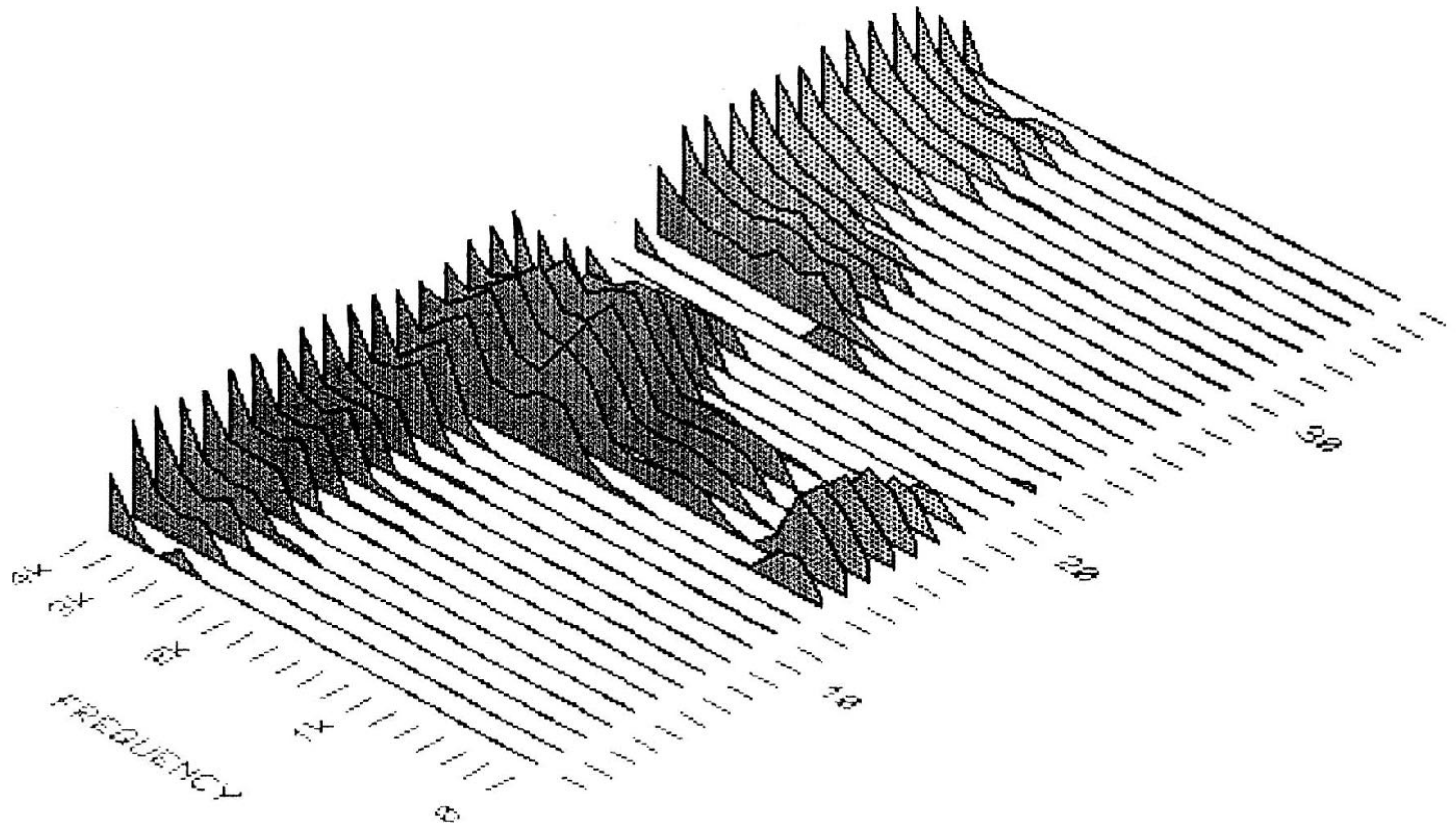
- Search for silence
- Set to silence
- Simple


‘Frame’ (100mS)
I
Sample index of silent frame
Z
Estimate of sample value representing ‘Zero’ (silence)

Noise reduced recordings

- “Uno Due Tre”
    - 8kHz at 12-bits
- “Uno” .. “Dieci”
    - 8kHz at 8-bits
    - 10kHz at 8-bits



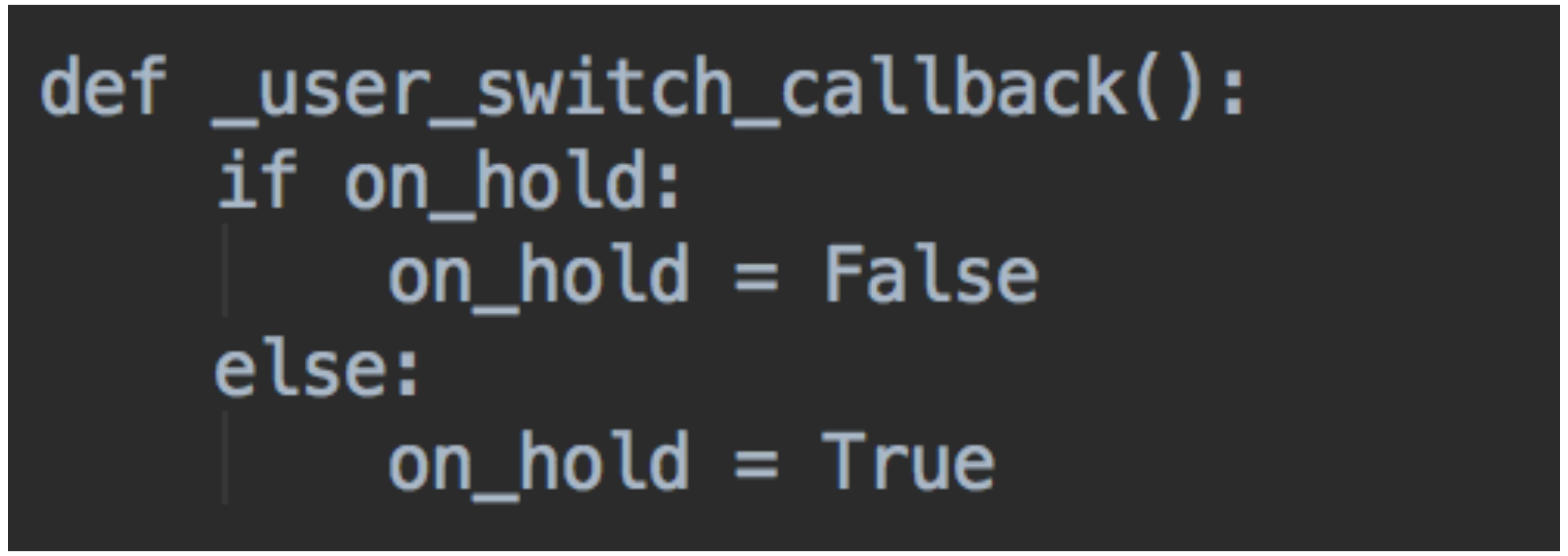


Application refinements

- Ability to “enable/disable” using the **USER switch**
- **LEDs** to indicate states
  - “Listening”, “Recording”, “Playing” and “Saving”
- Automatic recording (**speech detection**)
- Save recordings to an **SD card**
  - Built-in flash is very small

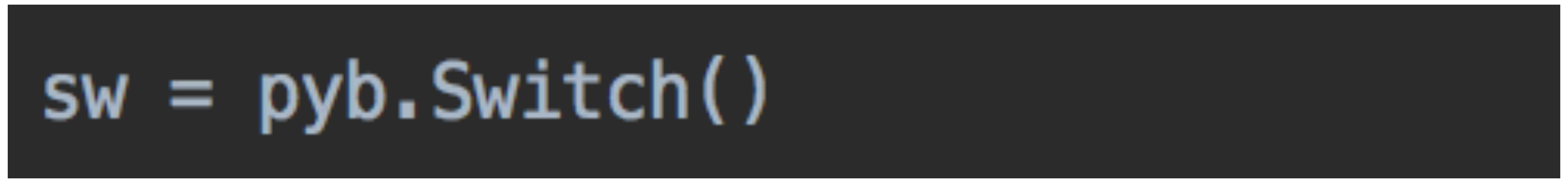
Responding to the USER switch

- Provide a “handler” function



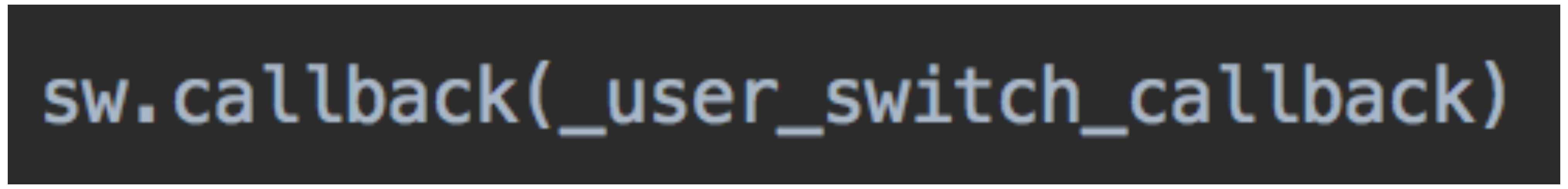
```
def _user_switch_callback():
    if on_hold:
        on_hold = False
    else:
        on_hold = True
```

- Create a “Switch” object

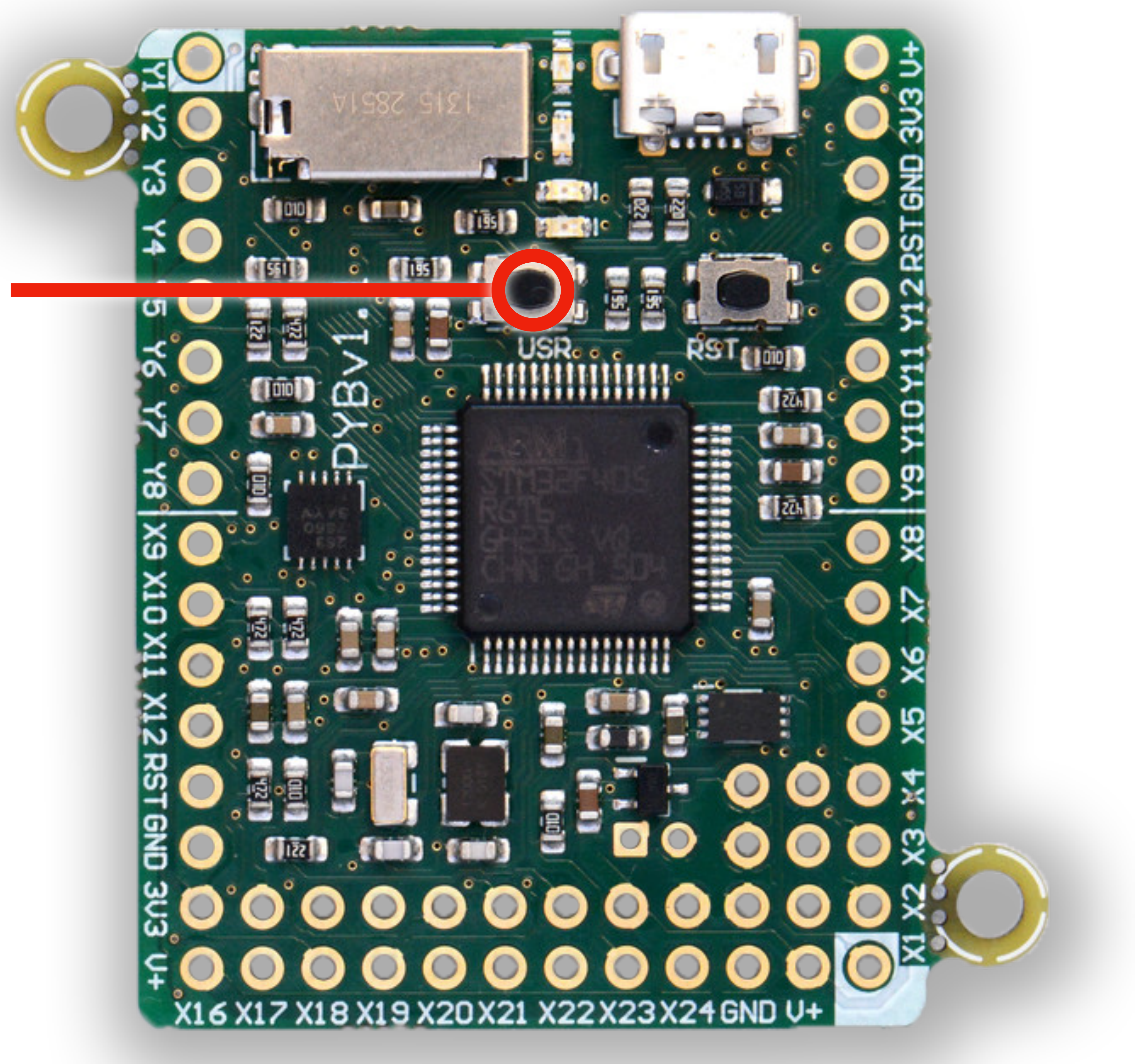


```
sw = pyb.Switch()
```

- Attach the “handler” as a callback



```
sw.callback(_user_switch_callback)
```

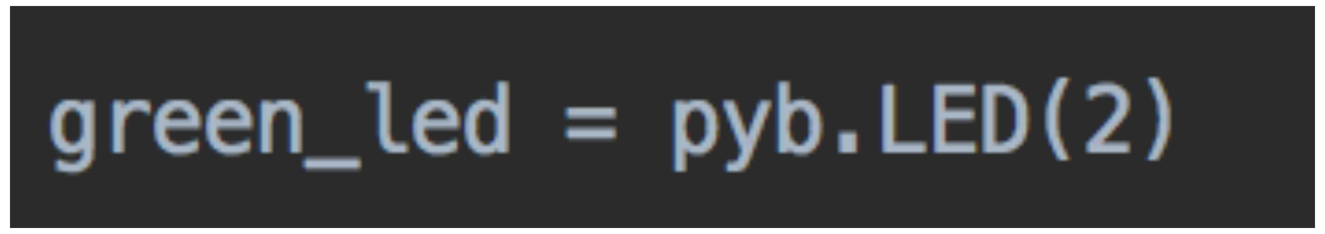


USER
switch

USR
RST
PYBv1.
Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8 X9 X10 X11 X12 RST GND 3U3 V+
X16 X17 X18 X19 X20 X21 X22 X23 X24 GND V+
V+ 3U3 GND RST Y12 Y11 Y10 Y9 X8 X7 X6 X5 X4 X3 X2 X1

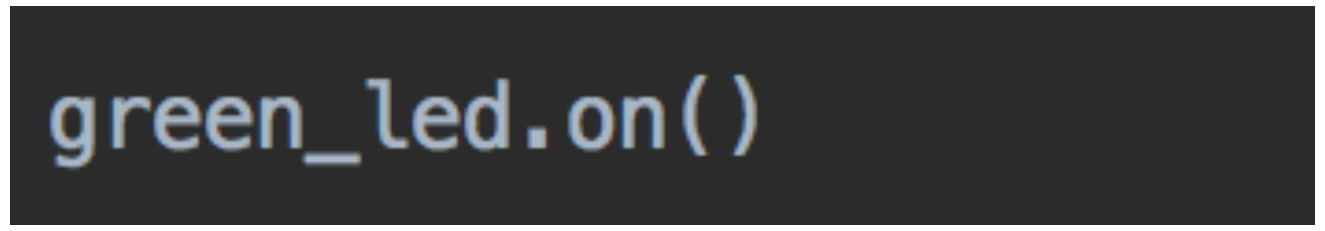
Driving the LEDs

- Create an LED object (1..4)



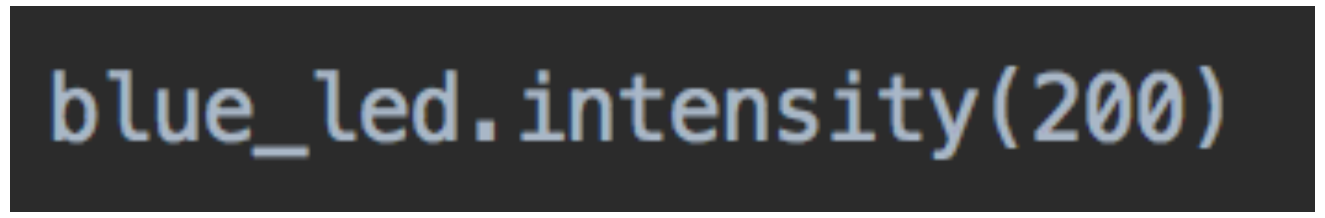
```
green_led = pyb.LED(2)
```

- Call “on()” or “toggle()”



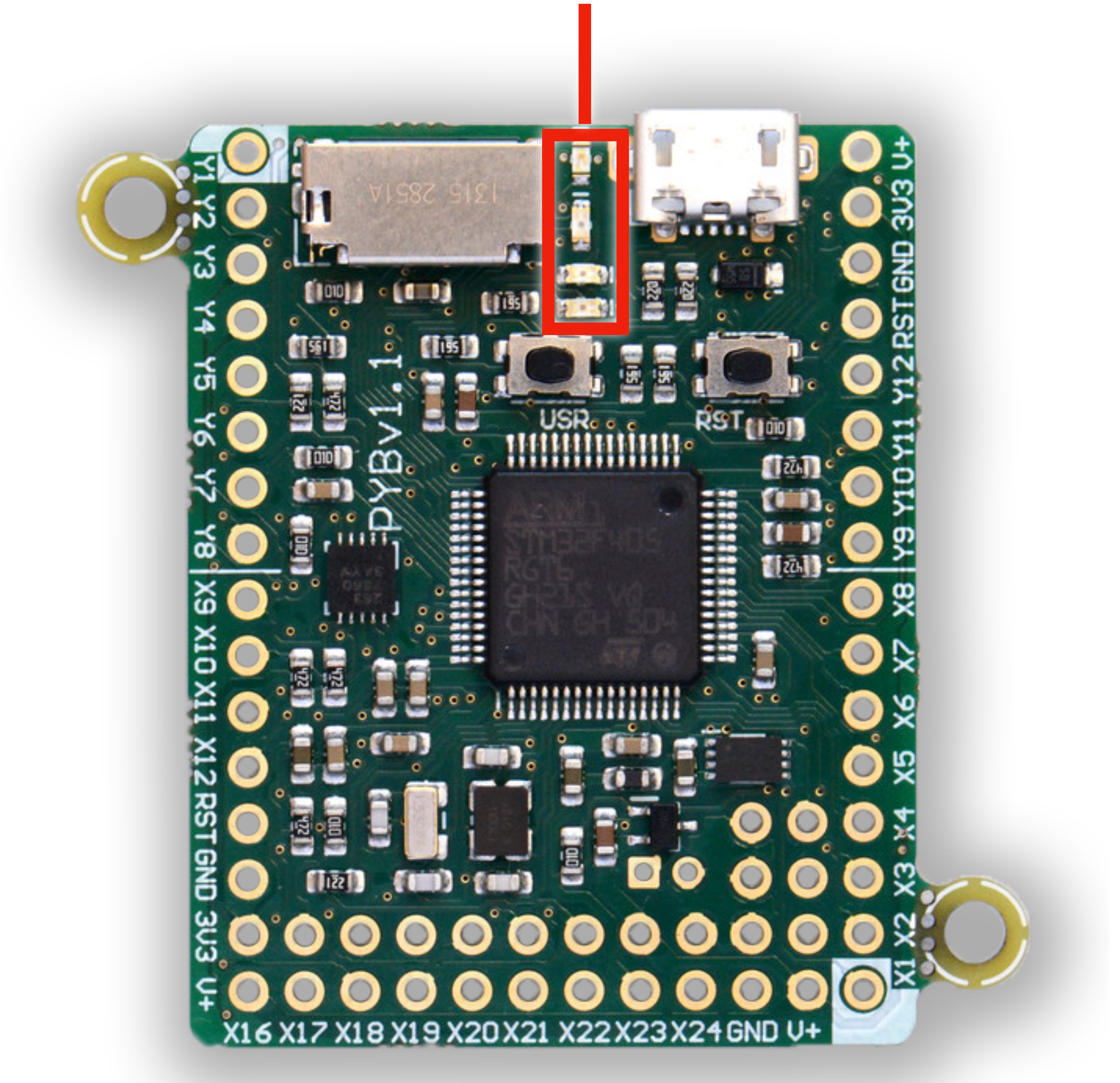
```
green_led.on()
```

- The blue LED (4) supports “intensity(n)”



```
blue_led.intensity(200)
```

4 LEDs

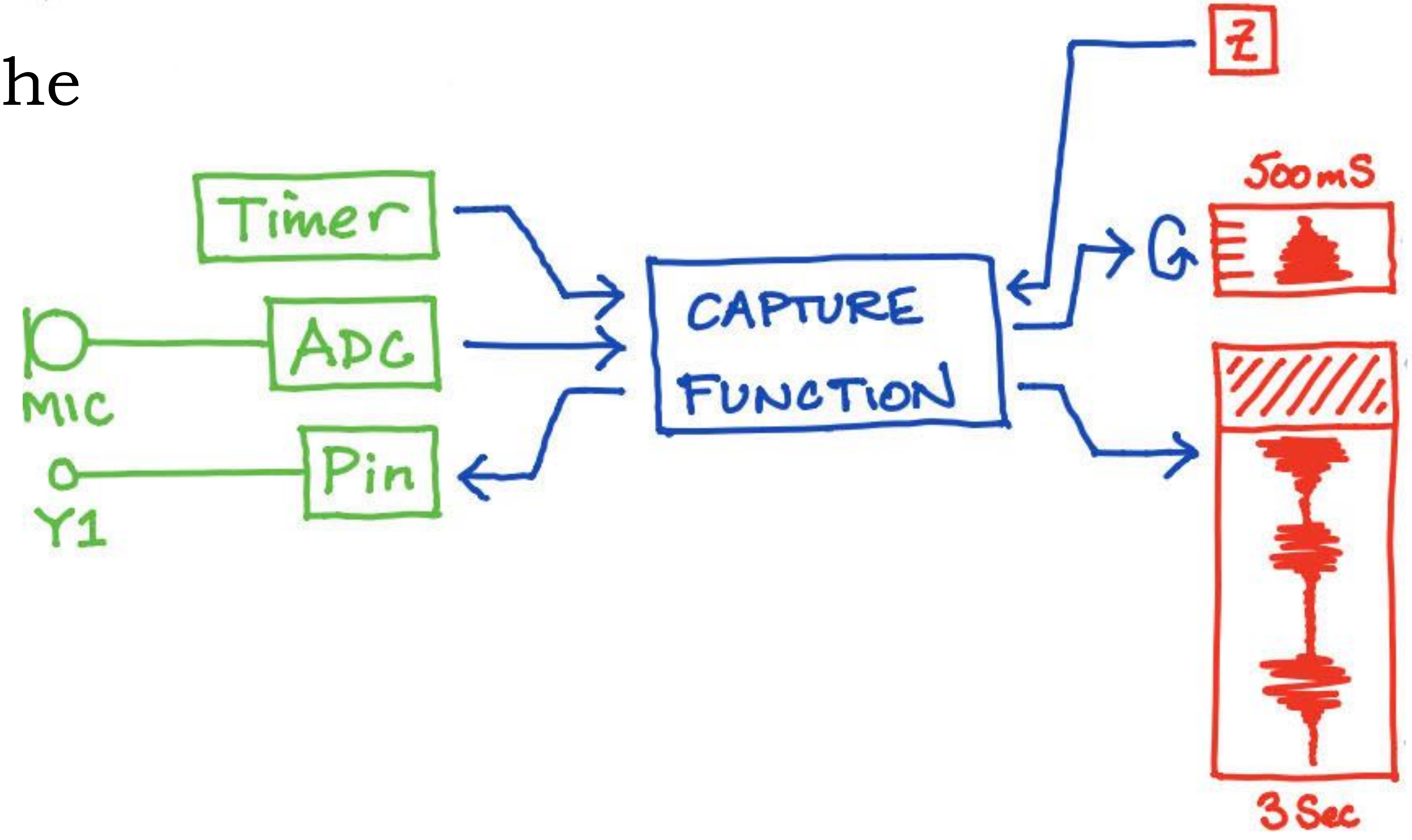

PYBv1.1
USR
RST
Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8 X9 X10 X11 X12 RST GND 3U3 V+
V+ 3U3 GND RST Y12 Y11 Y10 Y9 X8 X7 X6 X5 X4 X3 X2 X1
X16 X17 X18 X19 X20 X21 X22 X23 X24 GND V+

(Simple) automatic speech detection

- **Capture** to a small “circular” buffer prior to “recording”

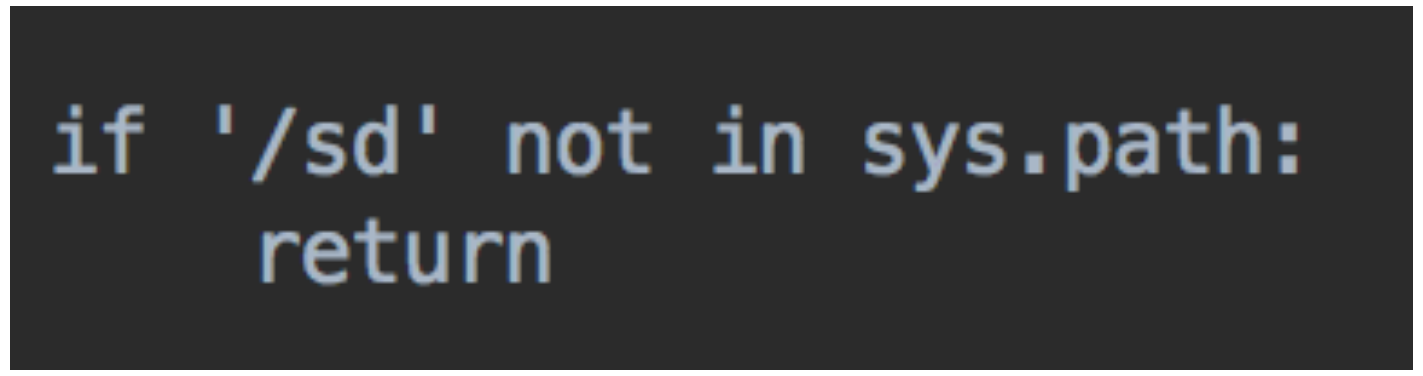
- **Analyse** samples within the “capture function”

- **Switch** to “record” on speech detection



Writing to an SD card

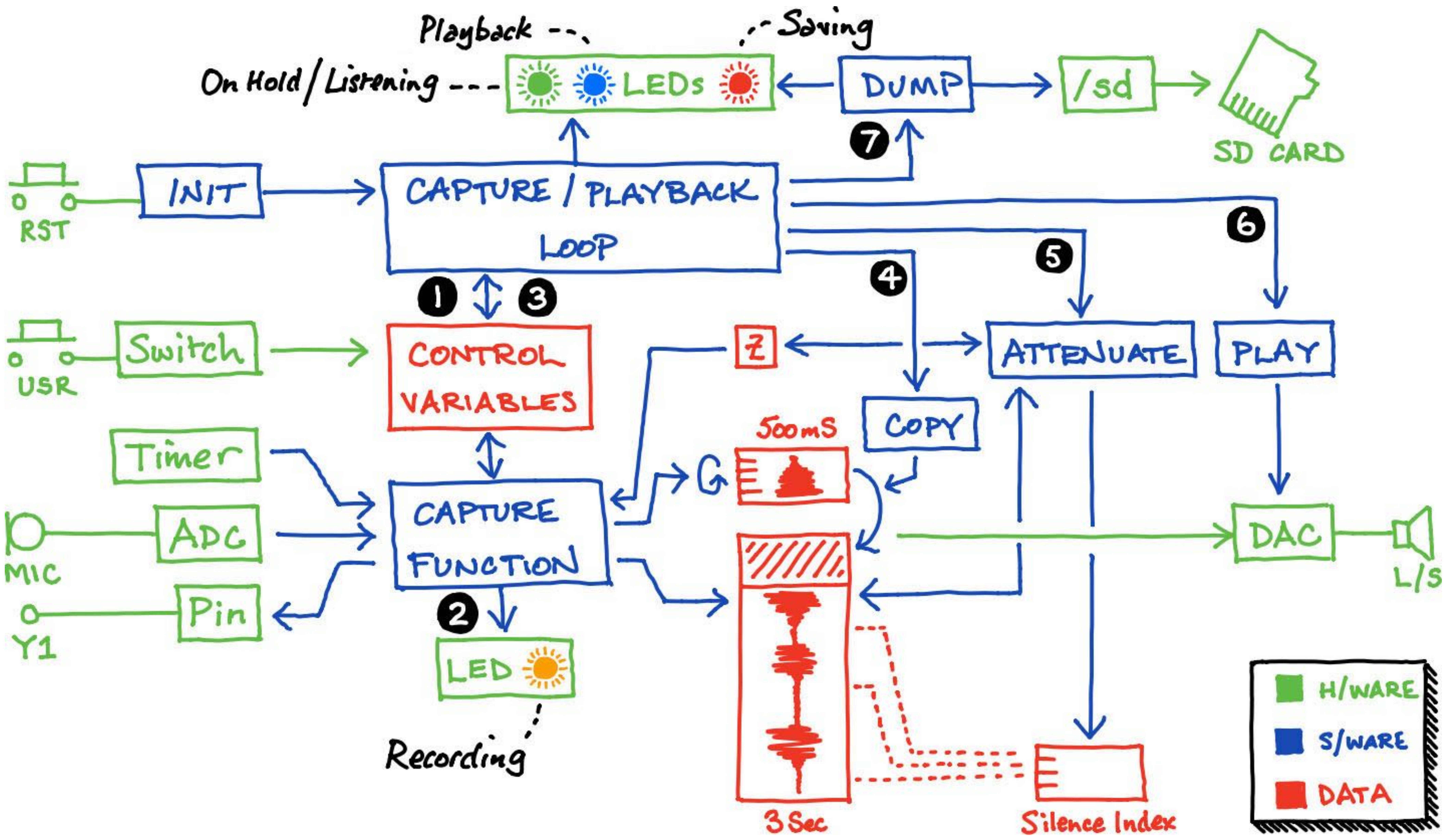
- Check if an SD card is present



```
if '/sd' not in sys.path:
    return
```

- Open, write, close

Putting it all together…



Grazie per l’ascolto Thank you for “listening”

Alan.Christie
at
MatildaPeak.com

@AlanBChristie

GitHub https://github.com/alanbchristie/**PyBdEcho**