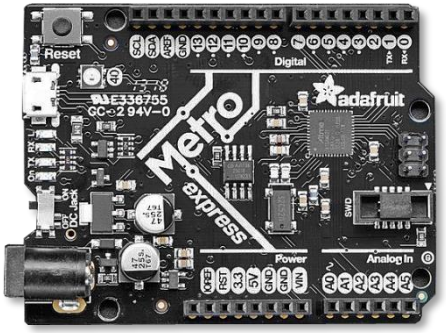
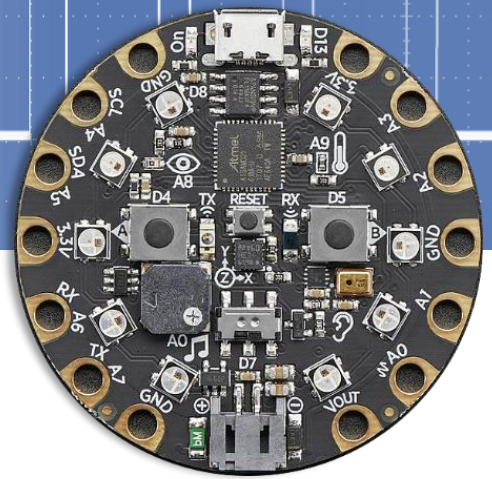
Circuit Python Beispielprogramme

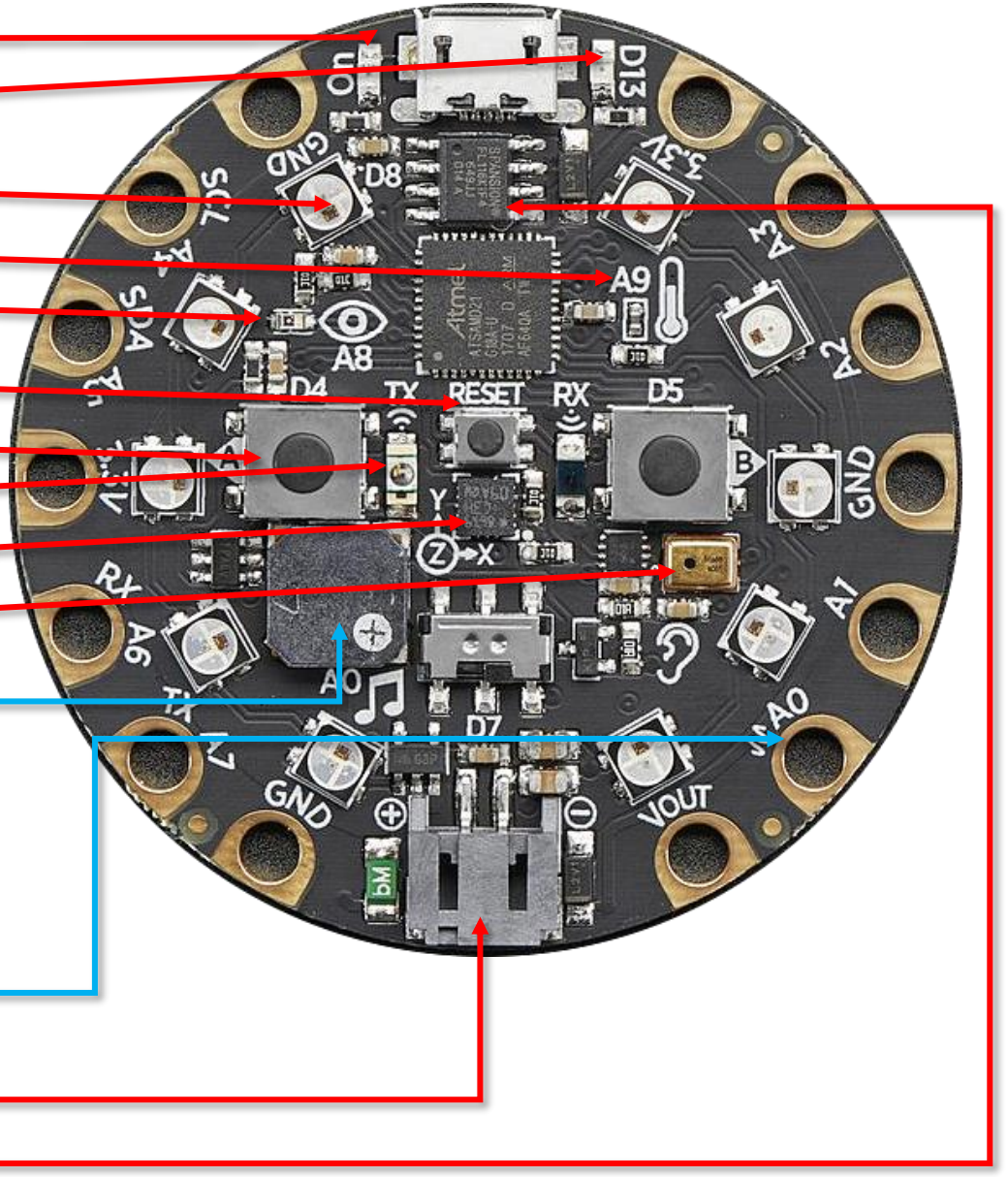
Johannes Kepler Gymnasium Weil der Stadt | Thomas Jörg | Stand: 03. September 2018 | Version 0.7



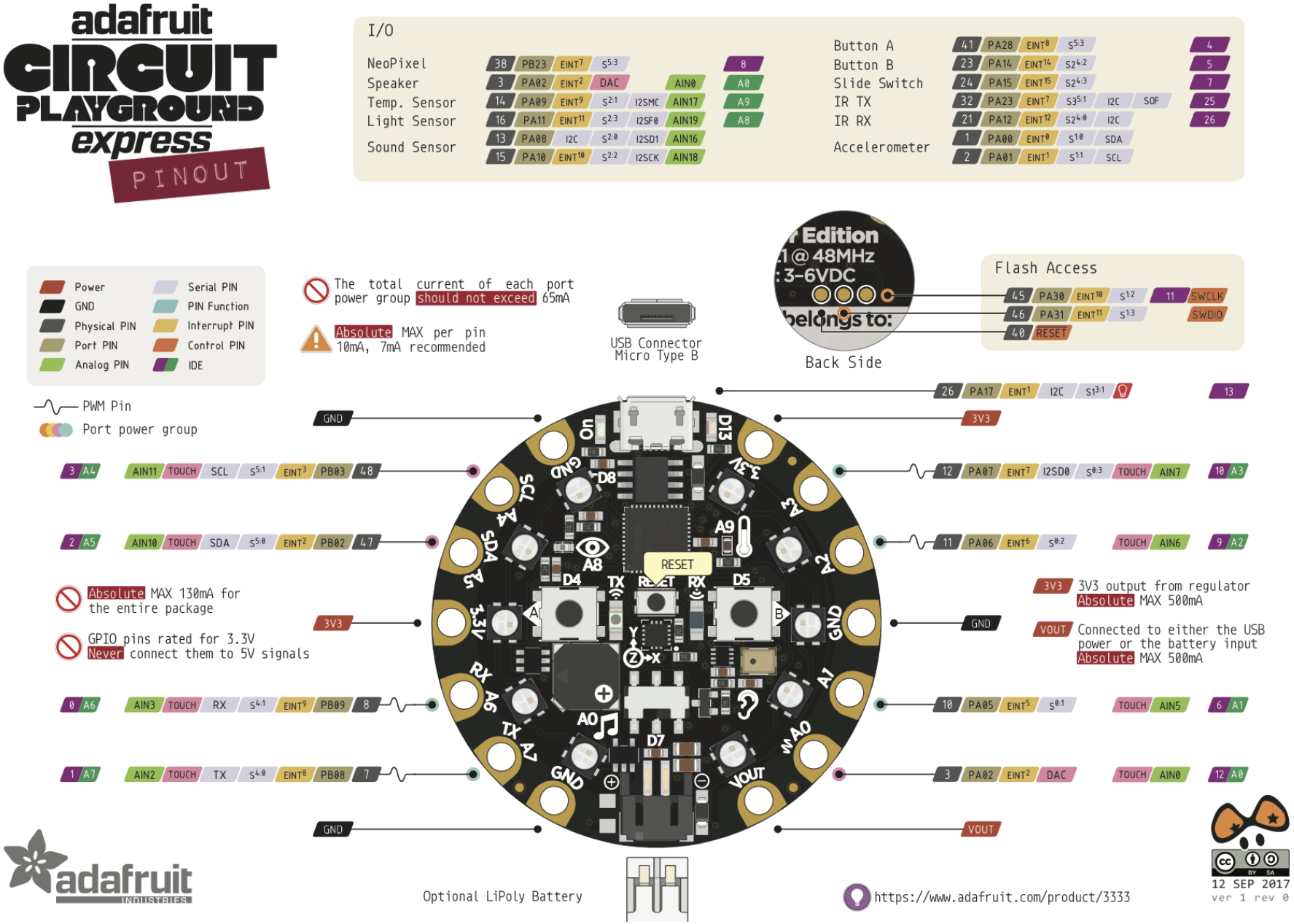


Aufbau des Circuit Playground Express (CPX)

- 1x grüne "ON" LED als Power-Indikator
- 1x rote "#13" LED, frei programmierbar
- 10 x NeoPixel
- 1 x Temperatur-Sensor, an A9
- 1 x Licht-Sensor (Phototransistor, ALS-PT19, verbunden mit A8)
- 1 x Reset-Knopf
- 2 x Drucktaster, gelabelt mit A und B
- 1 x Infrarot Sender/Empfänger, gelabelt mit RX/TX
- 1 x Beschleunigungs-Sensor (LIS3DH)
- 1 x Mikrofon mit Verstärker
- 1x Lautsprecher, verbunden mit dem DAC an A0
- 8 x Input/Output Pins
  - alle mit Analog-Inputs,
  - 1 x I2C (Anschlüsse SCL / SDA an Pins A4 und A5)
  - 6 x PWM Output (Pulsweiten-Modulation, A1, A2, A3, A6, A7)
  - 7 x Touch-Inputs (A1, A2, A3, A4, A5, A6, A7)
  - 1 x DAC („Digital-Analog-Converter, Analog Output an A0)
  - 1x UART (RX / TX an A6 / A7)
- Batterie-Anschluss
- 2 MB Speicher!!



Pinouts des CPX



Ergänzende Erläuterungen:

Der CPX besitzt insgesamt 14 Pins. Die wichtigsten Pinouts sind:

2 x 3,3V:	Ausgangsspannung von 3,3V für die permanente Versorgung von LEDs oder Sensoren durch das Board.
VOut:	Die Eingangs-Spannung von USB oder Batterie wird hier durchgeschleift, also unverändert durchgeleitet
3 x GND:	„Ground“, „Minuspol“
GPIO:	„General Purpose Input Output“, grundsätzlich alle als Digital Input oder Output mit max. 7mA Ausgangsstrom, bzw. alle als 12 Bit Analog-Digital-Converter.
TX / RX:	Serielle Kommunikationsschnittstelle, TX -> „Transmit“, senden / RX -> „Receive“, empfangen

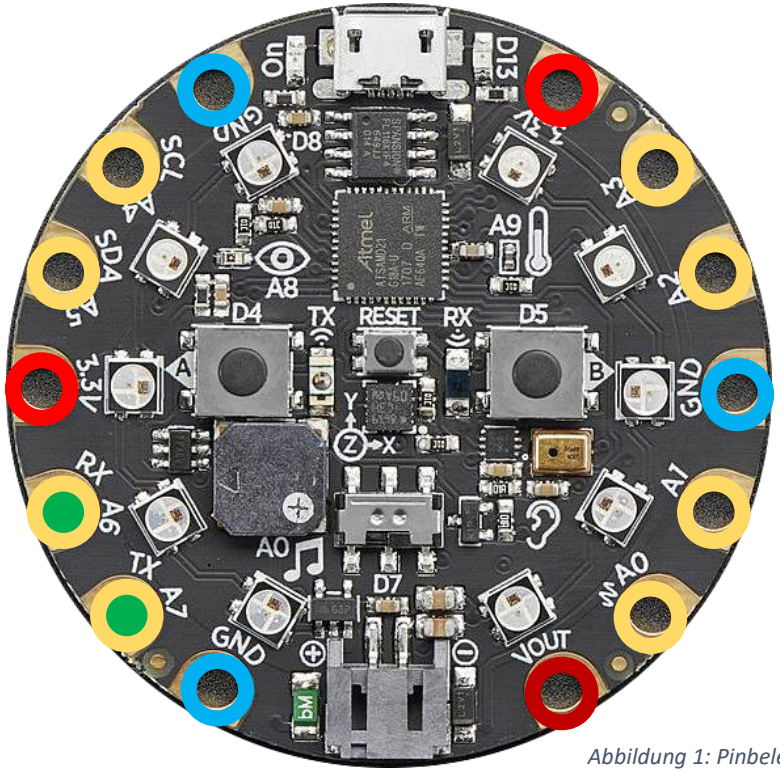


Abbildung 1: Pinbelegung des CPX

„Blink“: eine LED blinken lassen:

Schritt 1: Die interne rote LED an Pin 13 zum Blinken bringen:

```
import board          #Zugang zur Hardware
import time           #Zeitintervalle messen
import digitalio      #Pin In- und Output

led = digitalio.DigitalInOut(board.D13)
led.direction = digitalio.Direction.OUTPUT

while True:
    led.value = True
    time.sleep(1)
    led.value = False
    time.sleep(1)
```

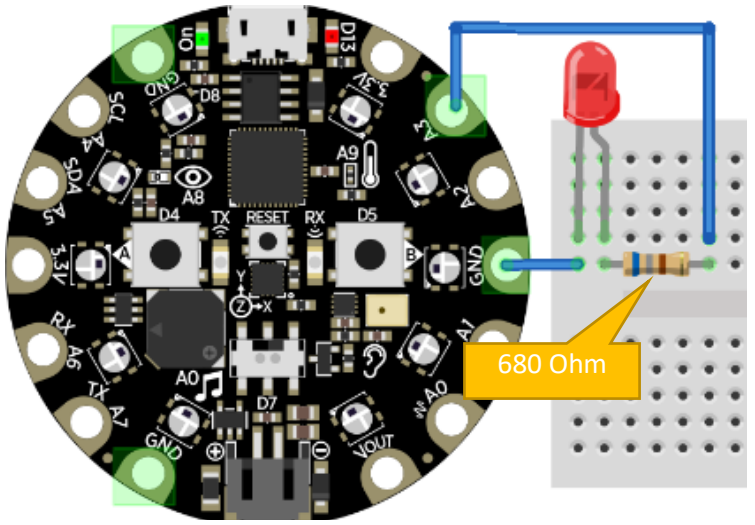


Abbildung 1: Ansteuerung einer LED

Schritt 2: Externe LED

Als Beispiel wird an den CPX-Pin A3 eine LED zusammen mit einem ca. 680 Ohm-Widerstand angebunden. Damit der Stromkreis geschlossen ist, verbindet man mit ‚GND‘. Setzt man den Pin A3 nun auf ‚True‘, dann wird er angeschaltet und liefert Spannung, um die LED zum Leuchten zu bringen.

Der Quelltext ändert sich nur in einer Zeile. Aus

```
led = digitalio.DigitalInOut(board.D13)
```

wird

```
led = digitalio.DigitalInOut(board.A0)
```

NIEMALS die LED ohne Widerstand mit dem CPX verbinden!

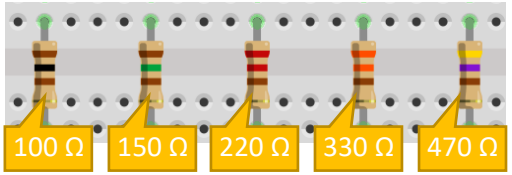


Abbildung 2: Beispiele für LED-Vorwiderstände.

Widerstandswerte werden durch die Farben der Ringe codiert:

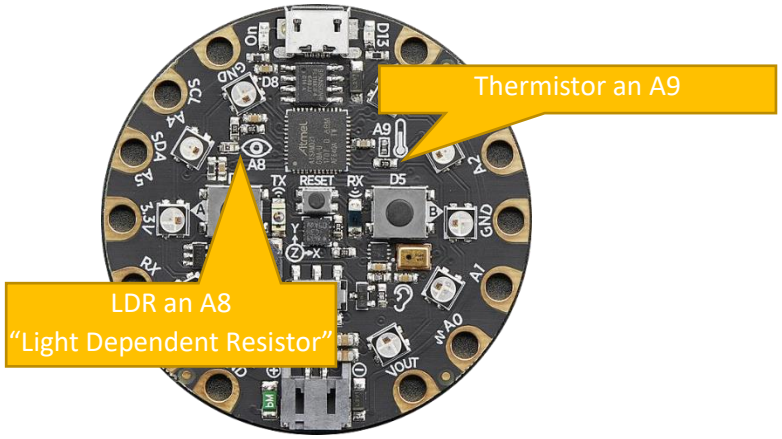
Die oberen beiden Ringe entsprechen einer Zahl, der dritte Ring einem Multiplikator (z.B. “mal 10” oder “mal 1000”. Der letzte Ring gibt die Genauigkeit wider.

https://www.elektronik-kompendium.de/sites/bau/1109051.htm

Exkurs: Berechnung Vorwiderstand für LED:

Eine rote LED benötigt etwa 1,65 Volt Betriebsspannung. Der CPX liefert 3,3 Volt an jedem PIN. 3,3 Volt – 1,9 volt = 1,4 Volt müssen „vernichtet“ werden. Zusätzlich sollten optimal 2 Milli-Ampere Stromstärke herrschen. Mit R = U/I = 1,4 Volt / 0,002 Ampere = 700 Ohm dimensioniert man den Widerstand.
https://www.elektronik-kompendium.de/sites/bau/1109111.htm

Abbildung 3: Analogsensoren des CPX



Der ‚Helligkeitssensor‘ am CPX:

Der CPX misst, wie stark das Licht eingestrahlt wird. Der LDR erzeugt jeweils zur Helligkeit einen entsprechenden Widerstandswert. Dieser Wert wird vom Mikrocontroller des CPX in einen Zahlenwert zwischen 0 (ganz dunkel) und 65535 (sehr hell) umgewandelt.

```
import board
import time
import analogio

licht = analogio.AnalogIn(board.LIGHT)

while True:
    print((licht.value,)) #ACHTUNG! Doppelte Klammer und Komma
sind kein Tippfehler. Wird benötigt zur Plotterausgabe: TUPEL!
    time.sleep(0.1)
```

Temperatursensor auslesen

Hierzu benötigt man die Adafruit-Bibliothek ‚adafruit_thermistor‘. Diese stellt die folgende Funktion zur Verfügung:

```
adafruit_thermistor.Thermistor(pin, series_resistor,
nominal_resistance, nominal_temperature, b_coefficient)
```

da am CPX der folgende Thermistor verbaut ist, ergibt sich die untere Synthax:
10K Precision Epoxy Thermistor - 3950 NTC

pin am CPX:	board.temperature
series_resistor:	10000
nominal_resistance:	10000 (10 Kilo-Ohm)
nominal_temperature:	25 *(bedeutet: 10Kilo-Ohm bei 25°C)*
b_coefficient:	3950

```
import board
import time
import adafruit_thermistor

t_sensor = adafruit_thermistor.Thermistor(
    board.TEMPERATURE, 10000, 10000, 25, 3950)

while True:
    print((t_sensor.temperature,))
    time.sleep(0.5)
```

Allgemeine Analogausgabe:

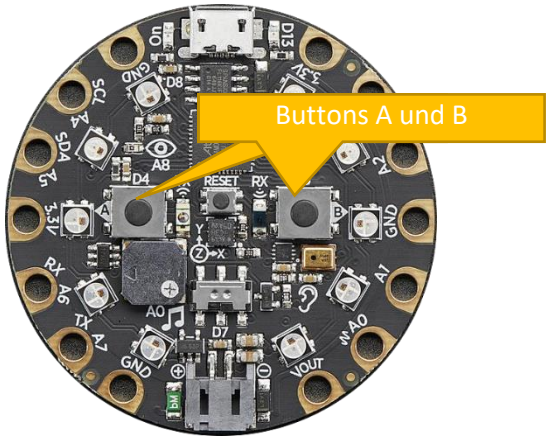
Die Bibliothek „adafruit_thermistor“ beispielsweise erledigt die Umrechnung von rohen Sensorwerten in menschlich interpretierbare Temperatur in °C.

Will man keine Bibliotheken zur Umrechnung verwenden, sondern die digitalisierten Rohwerte des Sensors nutzen, so verwendet man:

```
import board
import time
import analogio

t_sensor = analogio.AnalogIn(board.A9)

while True:
    print((t_sensor.value,))
    time.sleep(0.5)
```

Ersetzt man `board.A9` durch `board.A8`, so gibt man rohe Lichtsensorwerte aus.

Digitale Sensoren am CPX



Taster am CPX auslesen

```
import board
import time
import digitalio

Taste_A = digitalio.DigitalInOut(board.BUTTON_A) # BUTTON_A anschalten
Taste_A.direction = digitalio.Direction.INPUT # dann als Input schalten
Taste_A.pull = digitalio.Pull.DOWN # Grundzustand ist "DOWN", also "Aus"

Taste_B = digitalio.DigitalInOut(board.BUTTON_B)
Taste_B.direction = digitalio.Direction.INPUT
Taste_B.pull = digitalio.Pull.DOWN

while True:
    if Taste_A.value:
        print("A gedrueckt")
    elif Taste_B.value:
        print("B gedrueckt")
    else:
        print("nix gedrueckt")
    time.sleep(0.25)
```

Zum Beispiel für Button A könnte man auch kurz schreiben:

```
import board
import time
import digitalio

Taste_A = digitalio.DigitalInOut(board.D4)
Taste_A.direction = digitalio.Direction.INPUT
Taste_A.pull = digitalio.Pull.DOWN


while True:
    if Taste_A.value:
        print("A gedrueckt")
    time.sleep(0.25)
```

Touchsensoren am CPX

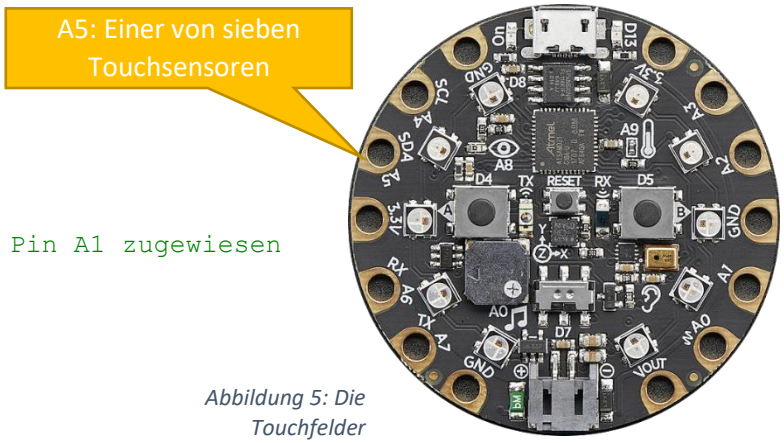
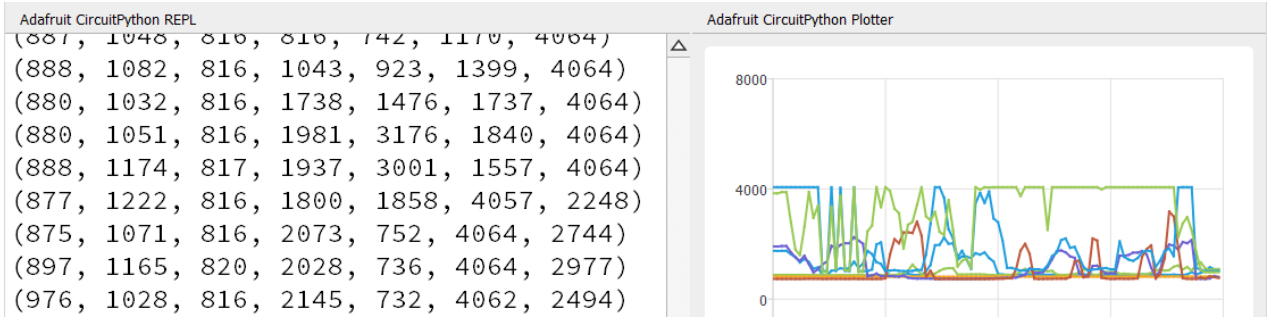


Abbildung 5: Die Touchfelder

```
import board
import time
import touchio

B_A1 = touchio.TouchIn(board.A1) # der Variable B_A1 wird ein Touch-Objekt an Pin A1 zugewiesen
B_A2 = touchio.TouchIn(board.A2)
B_A3 = touchio.TouchIn(board.A3)
B_A4 = touchio.TouchIn(board.A4)
B_A5 = touchio.TouchIn(board.A5)
B_A6 = touchio.TouchIn(board.A6)
B_A7 = touchio.TouchIn(board.A7)

while True:
    # Darstellung aller 7 touchpanel-Werte als Python-Tupel. Im Plotterfenster darstellbar
    print((B_A1.raw_value, B_A2.raw_value, B_A3.raw_value, B_A4.raw_value, B_A5.raw_value, B_A6.raw_value, B_A7.raw_value))
    time.sleep(0.1)
```



Die Ausgabe des obigen Programms im Mu-Editor, sowohl im Seriellen Fenster als auch im Plotter.

Will man lediglich überprüfen, ob ein Touch-Sensor berührt wurde oder nicht, so genügt ein simples „True“ oder „False“, also ein Digitalwert. Es wird dazu nicht der „raw_value“, sondern der „value“ ausgelesen. Die „print“-Zeile ändert sich dann so:

```
    print((B_A1.value, B_A2. value, B_A3.value, B_A4.value, B_A5.value, B_A6.value, B_A7.value))
```

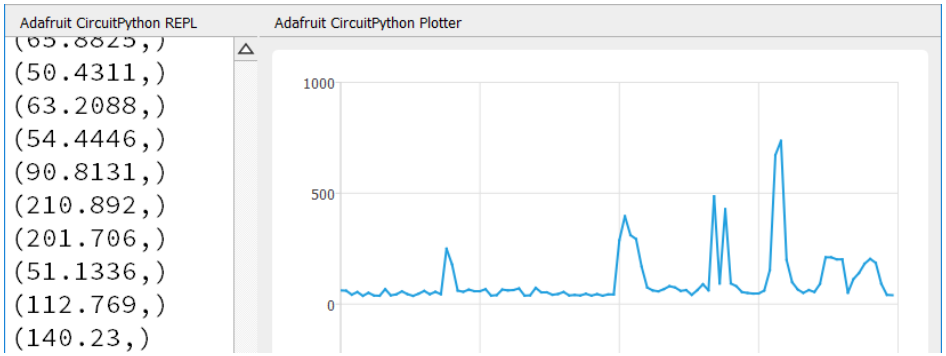
Der SoundSensor (PDM-Mikrofon)

Für die Benutzung des sogenannten PDM-Mikrofons ist die Bibliothek `audiobusio` notwendig. Ein PDM-Mikrofon ist ein sogenanntes „Puls-Dichte-Mikrofon“, welches die Tonsignale in Form einer Pulsweitenmodulation ausgibt. Diese Signale müssen per Software dekodiert werden. Daher auch die etwas „kryptische“ Programmierung.

Zur Nutzung des Programms:

Man sollte die Funktion BerechneLautstärke als Quelltext simpel übernehmen und diese im Hauptprogramm verwenden.

```
import board
import time
import array
import math
import audiobusio

def BerechneLautstaerke():
    mic.record(messwerte, len(messwerte))
    # ab hier: Berechnung der Mittleren Quadratischen Abweichung
    mittelwert = int(sum(messwerte) / len(messwerte))
    summeAbweichungen = sum(float(einzelmesswert - mittelwert) ** 2 for einzelmesswert in messwerte)
    return math.sqrt(summeAbweichungen / len(messwerte))

mic = audiobusio.PDMIn(board.MICROPHONE_CLOCK, board.MICROPHONE_DATA, sample_rate=16000, bit_depth=16)
messwerte = array.array('H', [0] * 160)

while True:
    lautstaerke = BerechneLautstaerke()
    print((lautstaerke,))
    time.sleep(0.1)
```



Die Ausgabe des obigen Programms im Serial-Fenster und im Plotter

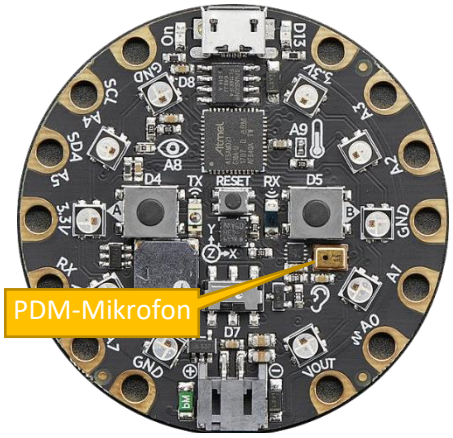


Abbildung 6: Das Mikrofon

Aktorik I: Neopixel-Ansteuerung

Der CPX besitzt insgesamt 10 sogenannte ‚Neopixel', das sind jeweils drei LEDs zu einem Leuchtmittel zusammengefasst: Ein Neopixel besteht aus je einer roten, einer grünen und einer blauen LED. Man kann also Lichtfarben mischen; es lassen sich so alle Regenbogenfarben erzeugen. Jede einzelne LED ist mit einem Helligkeitswert zwischen 0 und 255 ansteuerbar.

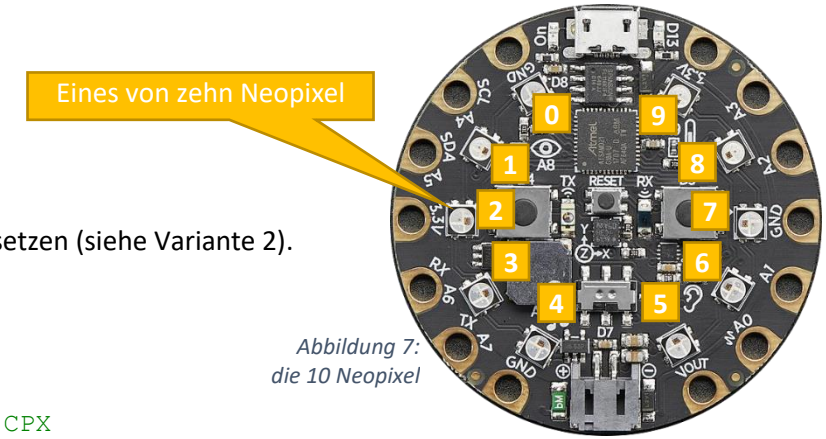


Abbildung 7: die 10 Neopixel

Wahlweise kann man die Helligkeit der gesamten 10er-Reihe auch mit der Eigenschaft „brightness" setzen (siehe Variante 2).

```
import board
import time
import neopixel

# ---------------------------------- Initialisierung der Neopixelreihe des CPX
NeopixelReihe = neopixel.NeoPixel(board.NEOPIXEL, 10)


# ---------------------------------- Variante 1, alle 10 LEDs werden gleichzeitig gesetzt. Automatisches Update
NeopixelReihe.fill((64,0,0))                    # Die  rote LED wird auf Helligkeit 0,25 (also 64/256) gesetzt
time.sleep(0.1)
NeopixelReihe.fill((0,64,0))                    # Die grüne LED wird auf Helligkeit 0,25 gesetzt
time.sleep(0.1)
NeopixelReihe.fill((0,0,64))                    # Die blaue LED wird auf Helligkeit 0,25 gesetzt
time.sleep(0.1)
NeopixelReihe.fill((0,0,0))                     # Alle LEDs werden ausgeschaltet


# ---------------------------------- Variante 2, jeweils eine LED wird gesetzt. Automatisches Update
# ---------------------------------- ausserdem:  Lauflicht vorwärts
for i in range (10):
    NeopixelReihe[i] = (0,255,0)                # bei NeoPixel Nummer 'i': Grün auf 100%
    NeopixelReihe.brightness = i/10             # Helligkeit wird heruntergesetzt
    time.sleep(0.1)


# ---------------------------------- Variante 3: LED-Automatik aus, muss durch show()-Befehl geupdatet werden
# ---------------------------------- ausserdem:  Lauflicht rückwärts
NeopixelReihe.auto_write = False                # automatisches LED-Update aus, ist schneller!
for j in range (10):
    NeopixelReihe.fill((0,0,0))                 # Alle LEDs abschalten (bzw. dazu vorbereiten)
    NeopixelReihe[9-j] = (64,64,64)             # RGB auf 0,25, also weisses Licht
    NeopixelReihe.show()                        # JETZT die LEDs updaten!
    time.sleep(0.01)
```

Aktorik II: Audiosample-Erzeugung und Ausgabe

Ein schwieriges Programm zur Tonerzeugung am DAC (Digital-Analog-Wandler)

Es wird zunächst ein Array – das ist eine spezielle Art einer Liste mit streng definierten Datentypen – erzeugt, in dem alle Zahlenwerte vorberechnet werden, welche der DAC zum Abspielen benötigt. Die Vorberechnung ist notwendig, weil der Prozessor des CPX nicht schnell genug ist, in Echtzeit die komplizierten mathematischen Berechnungen durchzuführen.

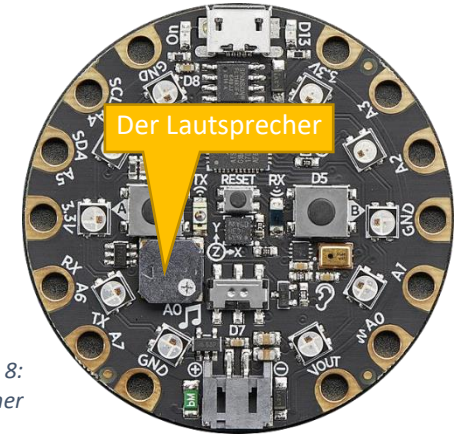


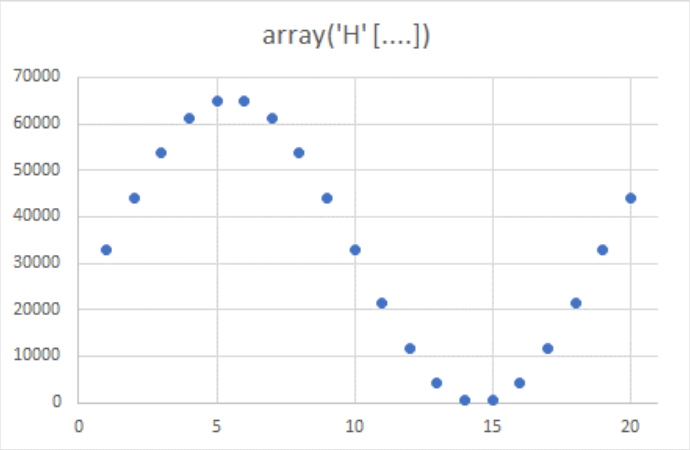
Abbildung 8:
Der Lautsprecher

```
import board
import time
import digitalio
import audioio
import array
import math

# ---------------------------------------------------------------  Samplerate und Frequenz eines Tons festlegen
PunkteProSekunde = 8000                                              # Anzahl Datenpunkte pro Sekunde, Samplerate
Frequenz = 400                                                       # Frequenz der Schwingung


# ---------------------------------------------------------------  Berechnung einer Sinusschwingung.
Punktanzahl = SAMPLERATE // FREQUENZ                                 # // "Floordivision", Teilen ohne Rest
Sinuspunkte = array.array("H", [0] * Punktanzahl)                    # "H": Array wird mit 2-Byte-Integer dimensioniert
for i in range(Punktanzahl):                                         # Schleife, wird 'Punktanzahl'-mal wiederholt
    Punkte[i] = int(32768 + 32768 * math.sin(math.pi * 2 * i / 18))  #Punkteberechnung. Mathe!

# ---------------------------------------------------------------  berechnete Punkte im Seriellen Fenster ausgeben
print(Punkte)

# ---------------------------------------------------------------  Lautsprecher anschalten
speaker_enable = digitalio.DigitalInOut(board.SPEAKER_ENABLE)        # Lautsprecher ansprechen
speaker_enable.direction = digitalio.Direction.OUTPUT                # auf 'Ausgabe' einstellen
speaker_enable.value = True                                          # ... und zum Schluss tatsächlich anschalten

# ---------------------------------------------------------------  Ausgabe-Objekt und Tondatei erzeugen
Ausgabe = audioio.AudioOut(board.A0)                                 # Audio-Objekt erzeugen
Sinuspunkte_sample = audioio.RawSample(Punkte)                       # eine abspielbare Datei erzeugen

# ---------------------------------------------------------------  Ton abspielen
Ausgabe.play(Sinuspunkte_sample, loop=True)                          # Sample wird unendlich oft abgespielt
time.sleep(1)                                                        # eine Sekunde warten,
Ausgabe.stop()                                                       # dann das Abspielen des Tons beenden
```

Der , `print(Punkte)`‘-Befehl gibt hier die folgenden Zahlenwerte aus, welche im Array gespeichert sind:

```
array('H',[32768,43975,53830,61145,65038,65038,61145,53830,43975,32768,21560,11705,4390,497,497,4390,11705,21560,32767,43975])
```

‚H' bedeutet hier ‚Integer-Zahlenwert, bestehend aus 2 Byte, also von 0 bis 65536. Siehe https://docs.python.org/3/library/array.html#module-array



In ein Diagramm übertragen sehen die Zahlenwerte des Arrays so aus.
Diese Punkte werden vom DAC (Digital-Analog-Wander) so abgespielt.

Ein einfacherer Weg, um Töne abzuspielen:

```
from adafruit_circuitplayground.express import cpx

while True:
    if cpx.touch_A4:
        cpx.start_tone(262)
    elif cpx.touch_A5:
        cpx.start_tone(294)
    elif cpx.touch_A6:
        cpx.start_tone(330)
    elif cpx.touch_A7:
        cpx.start_tone(349)
    elif cpx.touch_A1:
        cpx.start_tone(392)
    elif cpx.touch_A2:
        cpx.start_tone(440)
    elif cpx.touch_A3:
        cpx.start_tone(494)
    else:
        cpx.stop_tone()
```

Abbildung 9: I2C-Beispiel LIS3DH

Der I²C-Bus (sprich: „I-Quadrat-C“)

Problem: Viele Sensoren sind als Analogsensoren zu ungenau. Außerdem benötigen viele Sensoren eine aufwendige Datenaufbereitung, die mathematisch aufwendig ist. Ein gutes Beispiel ist der Beschleunigungssensor des CPX, ein, der für die Raum-Richtungen x-, y- und z Beschleunigungs-Rohdaten liefert. Diese 3 unterschiedlichen Rohdaten müssen mathematisch aufwendig verarbeitet werden und würde deshalb dem CPX viel Rechenleistung abverlangen.

Deshalb verbaut man solche Sensoren mit eigenen kleinen Mikrocontrollern zusammen. Der Sensor-eigene Mikrocontroller ist dann einerseits für die Datenverarbeitung zuständig. Weiterhin stellt dieser Sensor-Controller dem CPX dann die fertig aufbereiteten Daten zur Verfügung.

Die Daten fließen vom Sensor-Controller zum Arduino in einer eigenen standardisierten Sprache, die über eigene standardisierte Leitungen erfolgt. Man spricht bei der Sprache von einem sogenannten „Protokoll“, die Leitungen nennt man „BUS“ (BUS für „binary unit system“).

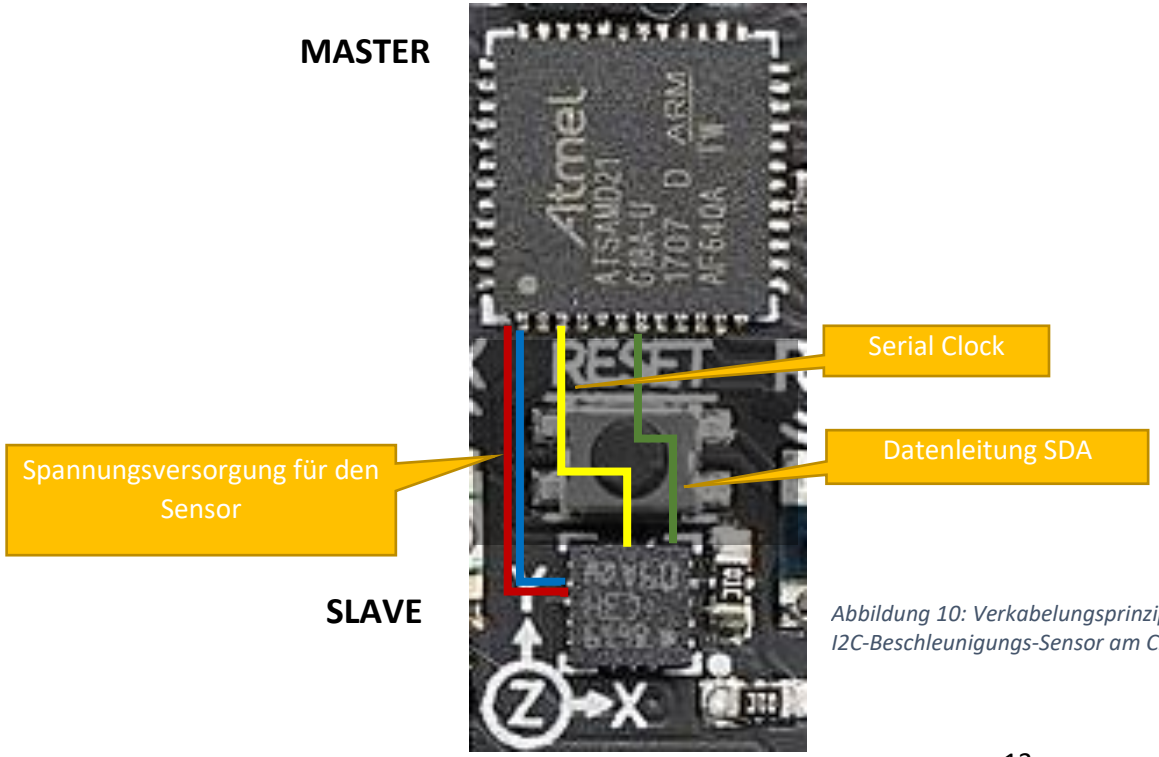
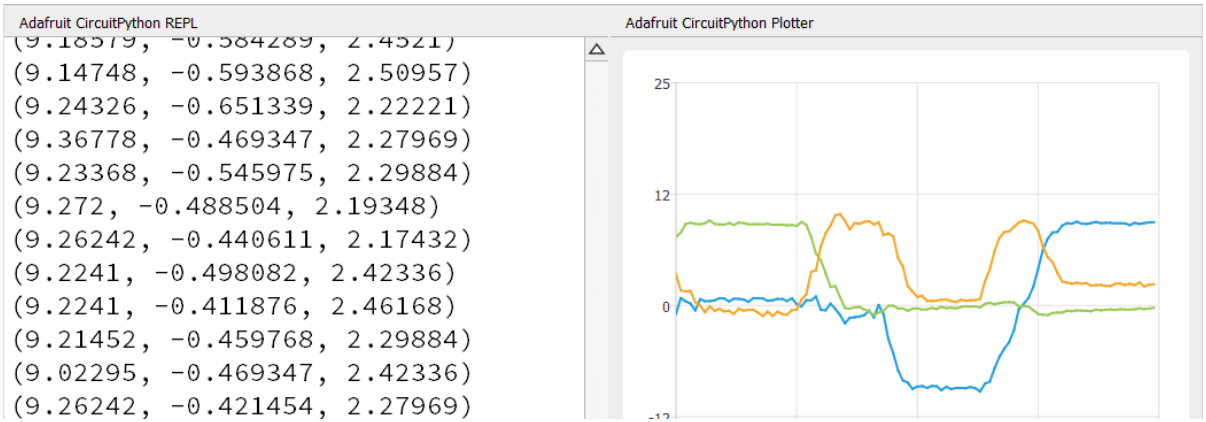


Abbildung 10: Verkabelungsprinzip I2C-Beschleunigungs-Sensor am CPX

CPX-eigener Beschleunigungssensor: LIS3DH

```
import board
import time
import busio                                    # i2c-Bus-Bibliothek.
import adafruit_lis3dh                          # Beschleunigungs-Sensor-Bibliothek

# ------------------------------------ Setup für i2c-Bus und Beschleunigungs-Sensor
i2c_bus = busio.I2C(board.ACCELEROMETER_SCL, board.ACCELEROMETER_SDA)
Beschl_Sensor = adafruit_lis3dh.LIS3DH_I2C(i2c_bus, address=0x19)

while True:
    x, y, z = Beschl_Sensor.acceleration        # Sensorwerte auslesen
    print((x, y, z))                            # als Python-Tupel ausgeben für Mu-Plotter
    time.sleep(0.1)
```



Die Ausgabe des obigen Programms im Mu-Editor, sowohl im Seriellen Fenster als auch im Plotter.

OLED128x32 I²C mit SSD1306-Controller

Der Controller besitzt die I²CAdresse 0x3C. Angeschlossen wird das Display wie folgt:

OLED	*Arduino UNO*
VCC	+3,3V
GND	GND
SCL	A4 (I²C Anforderung)
SDA	A5 (I²C Anforderung)

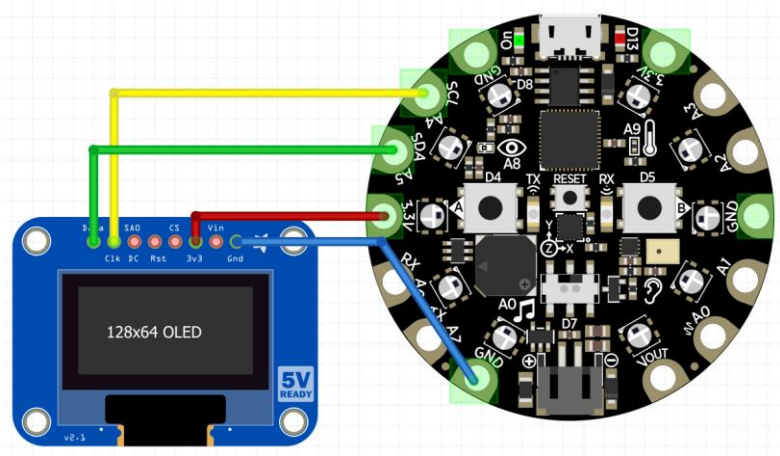
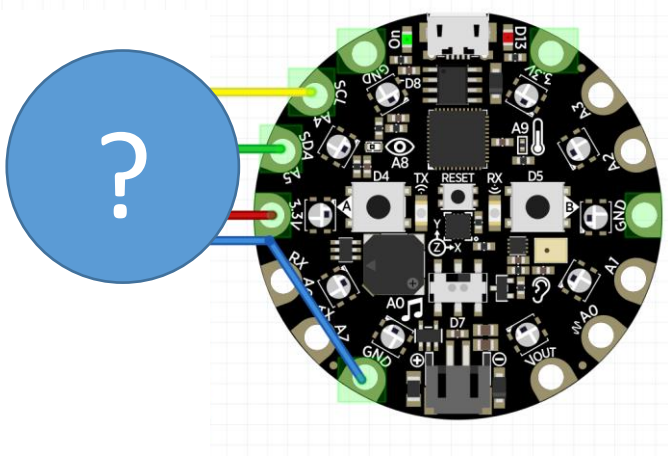


Abbildung 11: OLED Anschlüsse

```
import board
import busio
import adafruit_ssd1306

i2c = busio.I2C(board.SCL, board.SDA)
oled = adafruit_ssd1306.SSD1306_I2C(128, 64, i2c, addr=0x3c)

oled.fill(0)
oled.text('Hello', 0, 0)
oled.text('World', 0, 10)
oled.show()
```

Wo ist mein I2C-Gerät??




```
import time
import board
import busio

i2c = busio.I2C(board.SCL, board.SDA)

while not i2c.try_lock():
    pass

while True:
    print("I2C Adressen:", [hex(addr) for addr in i2c.scan()])
    time.sleep(2)
```

Blink erneut betrachtet: PWM (Pulsweitenmodulation)

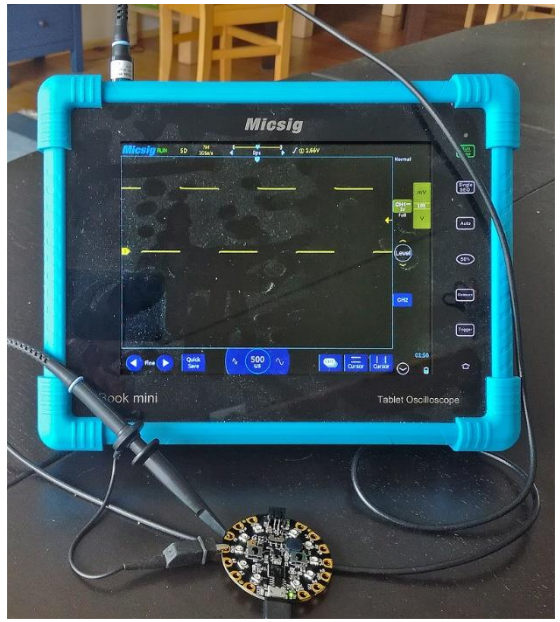


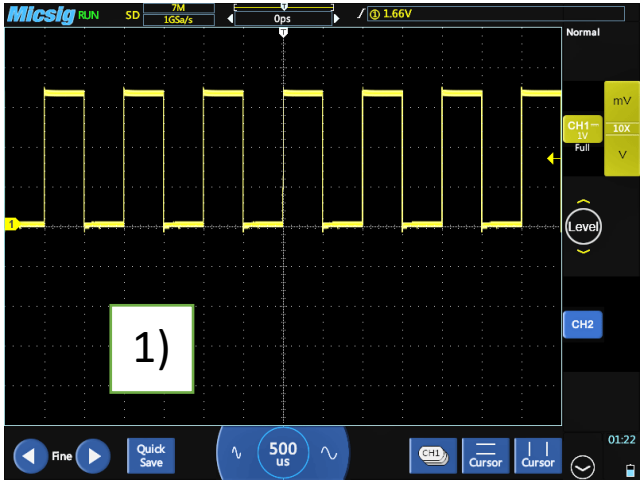
Abbildung 12: Oszilloskop an Port A1

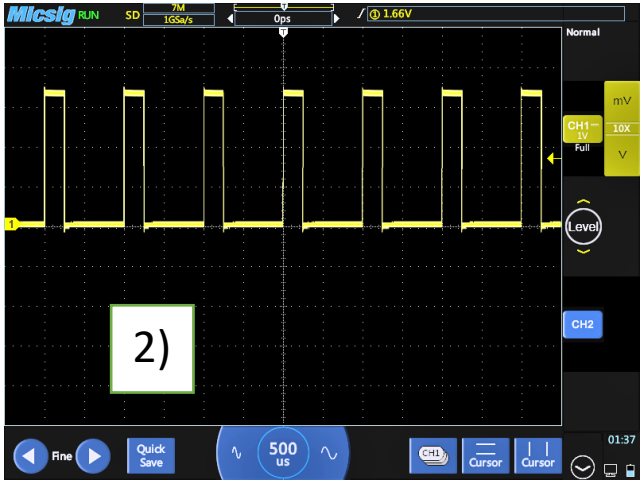
An einem Pin schaltet der Mikrocontroller eine Spannung ein oder aus, entsprechend den Zuständen HIGH oder LOW. Dieses Ein- und Ausschalten kann der Mikrocontroller allerdings sehr schnell und unendlich oft (bzw. zyklisch) durchführen. Zwei Parameter können dabei variiert werden: Die Frequenz, mit der an- und ausgeschaltet wird und der sogenannte Duty-Cycle. Hierbei wird das Verhältnis zwischen der Zeitspanne „Ausgeschaltet“ zur Zeitspanne „Eingeschaltet“ verändert. 100 Prozent Duty-Cycle bedeutet: Spannung liegt immer an. 50% Duty-Cycle heisst, dass zur Hälfte der Zeit die Spannung an, und zur anderen Hälfte ausgeschaltet ist.

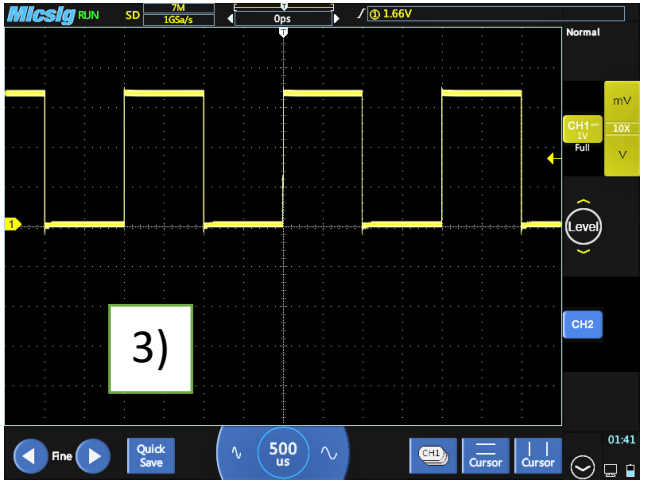
```
import board
import time
import pulseio # Bibliothek für PWM

# D13: Boardeigene LED, duty_cycle: zwischen 0 und 65536, frequency zwischen 1 Hz und 48 MHz
led = pulseio.PWMOut(board.D13, duty_cycle=32768, frequency=1000, variable_frequency=True)

while True:
    time.sleep(1)
```

Ändert man den Output-Pin auf beispielsweise board.A1, so kann man mit einem Oszilloskop die PWM direkt messen:

```
1) led = pulseio.PWMOut(board.A1, duty_cycle=32768, frequency=1000, variable_frequency=True)
2) led = pulseio.PWMOut(board.A1, duty_cycle=16384, frequency=1000, variable_frequency=True)
3) led = pulseio.PWMOut(board.A1, duty_cycle=32768, frequency= 500, variable_frequency=True)
```





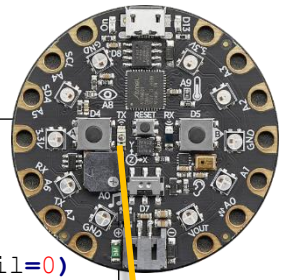


Der CPX als Maus

```
import board
import time
import digitalio
import busio
import adafruit_lis3dh
from adafruit_hid.mouse import Mouse

meineMaus = Mouse() # Maus-Objekt erzeugen

links = digitalio.DigitalInOut(board.BUTTON_A) # Linken und rechten Button initialisieren
links.direction = digitalio.Direction.INPUT
links.pull = digitalio.Pull.DOWN

rechts = digitalio.DigitalInOut(board.BUTTON_B)
rechts.direction = digitalio.Direction.INPUT
rechts.pull = digitalio.Pull.DOWN

i2c_bus = busio.I2C(board.ACCELEROMETER_SCL, board.ACCELEROMETER_SDA) # Beschleunigungssensor anschalten
Beschl_Sensor = adafruit_lis3dh.LIS3DH_I2C(i2c_bus, address=0x19)

while True:
    if links.value:
        meineMaus.press(Mouse.LEFT_BUTTON)
    elif not links.value:
        meineMaus.release(Mouse.LEFT_BUTTON)

    if rechts.value:
        meineMaus.press(Mouse.RIGHT_BUTTON)
    elif not rechts.value:
        meineMaus.release(Mouse.RIGHT_BUTTON)

    x, y, z = Beschl_Sensor.acceleration # Beschleunigungssensor auslesen
    if x<-1:
        meineMaus.move(x=-int(x*2))
    elif x>1:
        meineMaus.move(x=-int(x*2))

    if y<-1:
        meineMaus.move(y=-int(y*2))
    elif y>1:
        meineMaus.move(y=-int(y*2))
    time.sleep(0.01)
```

Infrarot Sender und Empfänger

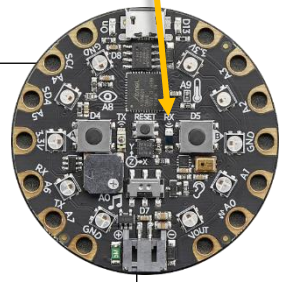
Mit der Bibliothek `adafruit_irremote` kann ein CPX als Sender und der andere als Empfänger programmiert werden. Dabei wird auf eine PWM mit 50% Duty-Cycle bei 38 kHz gearbeitet. Auf dieses 38 kHz-Signal wird nun durch Änderung der Duty-Cycles eine Bitfolge aufmoduliert. Diese Codierung erfolgt nach bestimmten Regeln. Sowohl Sender- als auch Empfänger codieren und decodieren das Sendesignal nach derselben Rechenanweisung.



Der Python-Code für den Sender

```
import board, time, pulseio, adafruit_irremote

traegerPWM = pulseio.PWMOut(board.IR_TX, frequency=38000, duty_cycle=32768)
sendePWM = pulseio.PulseOut(traegerPWM)
encoder = adafruit_irremote.GenericTransmit(header=[9500, 4500], one=[550, 550], zero=[550, 1700], trail=0)

while True:
    encoder.transmit(sendePWM, [128, 2, 128, 0, 77])
    time.sleep(2)
```

Hier wird die Nachricht `[128, 2, 128, 0, 77]` versendet, und zwar alle zwei Sekunden erneut in einer Endlosschleife.



Der Python-Code für den Empfänger

```
import pulseio, board, adafruit_irremote

empfangsPWM = pulseio.PulseIn(board.IR_RX, maxlen=120, idle_state=True)
decoder = adafruit_irremote.GenericDecode()

while True:
    pulses = decoder.read_pulses(empfangsPWM)
    try:
        decodiertesSignal = decoder.decode_bits(pulses, debug=False)
    except adafruit_irremote.IRNECRepeatException:
        print("NEC repeat!")
    except adafruit_irremote.IRDecodeException as e:
        print("Failed to decode: ", e.args)
    print("Anzahl Pulse:", len(pulses), "  Infrarot Code:", decodiertesSignal)
```

Es wird „versucht“, die empfangene Puls-Sequenz zu decodieren („try“). Falls es nicht funktionieren sollte, so werden die Reaktionen des Programms auf zwei verschiedene Fehler definiert. Dies verhindert, dass das Programm während der Ausführung plötzlich abbricht (**except** `IRNECRepeatException` und **except** `IRDecodeException`)

Inhalt