# Raspberry Pi Pico as a Disruptive Radio Transmitter

M. Andrecut

August 19, 2025

Unlimited Analytics Inc.
Calgary, Alberta, Canada
mircea.andrecut@gmail.com

**Abstract**

In this paper we explore several surprisingly simple methods for hacking the Raspberry Pi Pico (RP2) microcontroller into a disruptive radio transmitter. We show how these techniques can be easily used to transform the RP2 into a (potentially clandestine) radio station, or even into a microphone radio (spying bug), by simply using cheap off the shelf electronic components and open source software.

## 1 Introduction

For many years, spying movies have always excited our imagination with interesting action stories, but also with intriguing and cool spying gadgets and techniques. Watching these highly specialized devices "at work" in the movies is always fun, however today's technology has evolved so much that one can easily turn some of the commercially existing devices (computers, TVs, radios, fridges, stoves etc.) into spying bugs. This is also the case of the Raspberry Pi Pico (RP2) [1], which is a very popular and cheap microcontroller frequently used in IOT (Internet of Things) applications. While the RP2 comes in several flavors, such as the more expensive version W (containing the additional Wi-Fi and Bluetooth connectivity modules) [2], here we limit ourselves to the cheapest and most stripped down version, which currently one can buy on Amazon for a mere $4. This cheap RP2 version supposedly lacks any radio communication capabilities, however here we will show how it can be easily transformed into a (potentially clandestine) radio station, or even into a microphone radio (spying bug), just by using very cheap, off the shelf electronic components, also available on Amazon, and existing free open source software.

While the project sounds exciting, and one can be very eager to try it out immediately, we should warn the reader that any strong radio transmissions are illegal, unless you have a permit. Also, the legislation varies from country to country, and to avoid any legal problems it is strongly recommended to minimize the transmission time and range to several seconds and meters, and to use the described techniques only for experimental and educational purposes, and to avoid as much as possible causing any significant disruption. The scope of this work is to raise awareness, and the author is not responsible for the illegal use of the techniques described here.

## 2 The RP2

The RP2 is a low-cost, high-performance microcontroller board using the ARM Cortex-M0+ RP2040 chip produced by the Raspberry Pi Foundation [1], [2]. The RP2040 chip is a very efficient 133MHz dual-core CPU, and incorporates 264KB of SRAM, 2MB of Flash memory, accurate timing modules, 26 GPIO (General-Input-Output) pins, 3 ADC pins (Analog-to-Digital Converters), 8xPIO (Programmable I/O) state machines for custom peripherals, and commonly used peripheral interface modules such as: 2xUART (Universal Asynchronous Receiver-Transmitter), 16xPWM (Pulse Width Modulation) channels, 2xSPI (Serial Peripheral Interface), 2xI2C (Inter-Integrated Circuit). However, in its most basic version which we will use here, the board does not offer any radio communication capabilities, such as Wi-Fi or Bluetooth.

The RP2 can be programmed in C/C++ or MicroPython. In this paper we use MicroPython [3], since it is more accessible. MicroPython's module provides low level functions related to the hardware on the microcontroller board [4]. Some of these functions have direct and unrestricted hardware access to the CPU, timers, buses etc. An incorrect use may lead to malfunction, lockups, crashes, and hardware damage in some extreme cases. From the MicroPython point of view the RP2 board can be controlled using eleven different software classes. Here we will only use the following classes:

- Pin - control of the I/O pins.

  A Pin object is used to control GPIO, and it has methods to set the mode of the pin as IN or OUT, and to get and set the digital logic level. The RP2W has 26 multifunction GPIO pins connectable to external devices.

- PWM - pulse width modulation.

  The signal sent to an output pin is either HIGH or LOW voltage. PWM it is used to control the amount of time a signal is HIGH. There are 8 independent PWM generators called slices, each having two channels. The generators can be clocked from 8Hz to 62.5Mhz, at a machine frequency of 125Mhz. While the two channels of a slice will run at the same frequency, they can have a different duty rate. The two channels are usually assigned to adjacent GPIO pin pairs with even/odd numbers: slice 0 (GPIO0, GPIO1), slice 1 (GPIO2, GPIO3), and so on.

- ADC - analog to digital conversion.

  We can use this class to sample continuous voltages and convert them to discrete values. The standard ADC input value range is 0-3.3V. There are four ADC channels, where the input signal for ADC0, ADC1, ADC2 and ADC3 can be connected with: GPIO26, GPIO27, GPIO28, and GPIO29.

## 3 Radio configuration test

The *metamorphosis* of the stripped down RP2 into a radio transmitter is surprisingly simple, and it only requires the use of PWM to generate both the modulator and the carrier signals, a capacitor to connect the signals, and a wire as an antenna.

For experimental purposes we would like to limit our transmission range to a small local radius (around 10-20m), and therefore here we will use a simple antenna consisting of a short wire of about 10-20cm. The basic connectivity schema is shown in Figure 1. The RP2 is powered by using the USB connector, and this ensures that VBUS is at Vcc=5V, and all GND pins are connected to the ground. Here we simply connect the GPIO0 and GPIO2 using a capacitor C $\simeq 1 - 100\mu$F, and we attach the short wire antenna to the GPIO0. Also, we can use the Thonny IDE [5] to write a simple Python test program, and run it on RP2 (Listing 1).

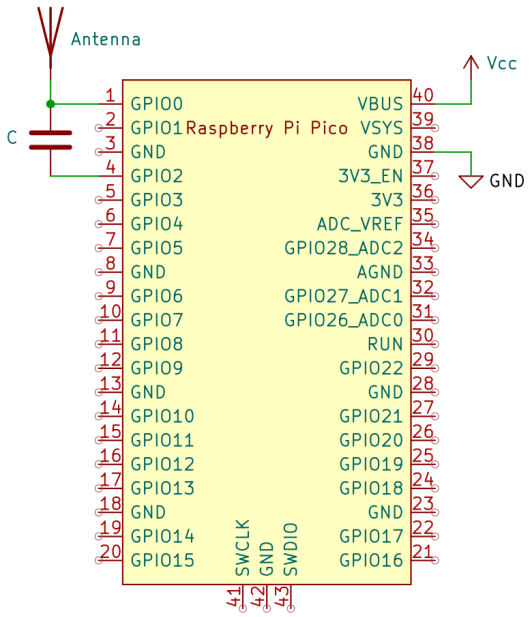


Figure 1: RP2 configuration.

```
import time
from machine import Pin, PWM

pwm0 = PWM(Pin(0))
pwm0.duty_u16(16384)
pwm0.freq(31250000)
pwm1 = PWM(Pin(2))
pwm1.duty_u16(32768)
while True:
    for n in range(100):
        pwm1.freq(10*n + 200)
        time.sleep(0.01)
```

Listing 1: Initial RP2 test

The program generates a carrier signal with the frequency $\nu^* = 31.25$MHz on GPIO0, and a linear frequency sweep signal with a frequency $\nu = (10n + 200)$Hz, $n = 0, 1, ..., 99$, and a duration of 1s on GPIO2. The carrier signal is then modulated by the sweep signal on GPIO2 via the capacitor connection C. The sweep signal is repeated until the program is stopped from the Thonny IDE.

Now that we have the sweep signal generated and radio transmitted, we should be able to tune the radio on 31.25MHz and hear it. Unfortunately, the FM radio band range is between 88MHz to 108MHz, so unless you have a radio that can be tuned outside of the FM band you won't be able to hear anything, unless you tune the radio to some multiple of the carrier frequency which is in the FM band, such as 93.75MHz which may be already occupied by a stronger radio channel.

We should note that while this method of using rectangular PWM signals for both the carrier and the modulation does generate a radio transmission, it also creates an infinite number of harmonics in the radio spectrum. It is well known that a rectangular wave signal is made up of an infinite number of sine waves, starting at the fundamental frequency and then going up with all the odd numbered harmonics [4]. In general, a rectangular PWM signal with the period $T = 2L = 1/\nu$ can be defined as following:

$$\Pi(t) = 2[H(t/L) - H(t/L - 1)] - 1, \tag{1}$$

where $H(x)$ is the Heaviside step function:

$$H(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1/2 & \text{for } x = 0 \\ 1 & \text{for } x > 0 \end{cases} \tag{2}$$

The Fourier series of a periodic signal $f(t)$ is defined as an infinite sum of sines and cosines:

$$f(t) = a_0 + \sum_{n=1}^{\infty} [a_n \cos(nt) + b_n \sin(nt)]. \tag{3}$$

For a rectangular function we have $a_0 = a_n = 0$, since the rectangular function is odd. Therefore only the $b_n$ coefficients need to be calculated, and they are given by:

$$b_n = \frac{1}{L} \int_0^{2L} \Pi(t) \sin\left(\frac{n\pi t}{L}\right) dt = \frac{4}{n\pi} \sin^2\left(\frac{n\pi}{2}\right) = \frac{2}{n\pi}[1 - (-1)^n] = \begin{cases} 0 & \text{for } n = 2k \\ \frac{4}{n\pi} & \text{for } n = 2k + 1 \end{cases} \tag{4}$$

such that we have:

$$\Pi(t) = \frac{4}{\pi} \sum_{k=0}^{\infty} \frac{1}{2k+1} \sin\left(\frac{(2k+1)\pi t}{L}\right). \tag{5}$$

In principle, the unwanted harmonics could be attenuated using a band filter [4]. However, to keep things simple, and as disruptive as possible, here we continue without the band filter.

So, how can we hear the radio transmission if the frequency falls outside of the standard radio bands? The simplest way to approach this is by using an RTL-SDR receiver, which can be connected to a computer or to a cellular phone using an USB connector.

The RTL-SDR receiver is built around the RTL2832U and R820T chips, which are a high-performance demodulator and tuner, initialy developed for the digital TV industry. The RTL-SDR dongles can receive radio frequencies from 500kHz up to 1.75GHz, depending on the particular model, and here we will use the cheapest version, which also can be found on Amazon for about $30.

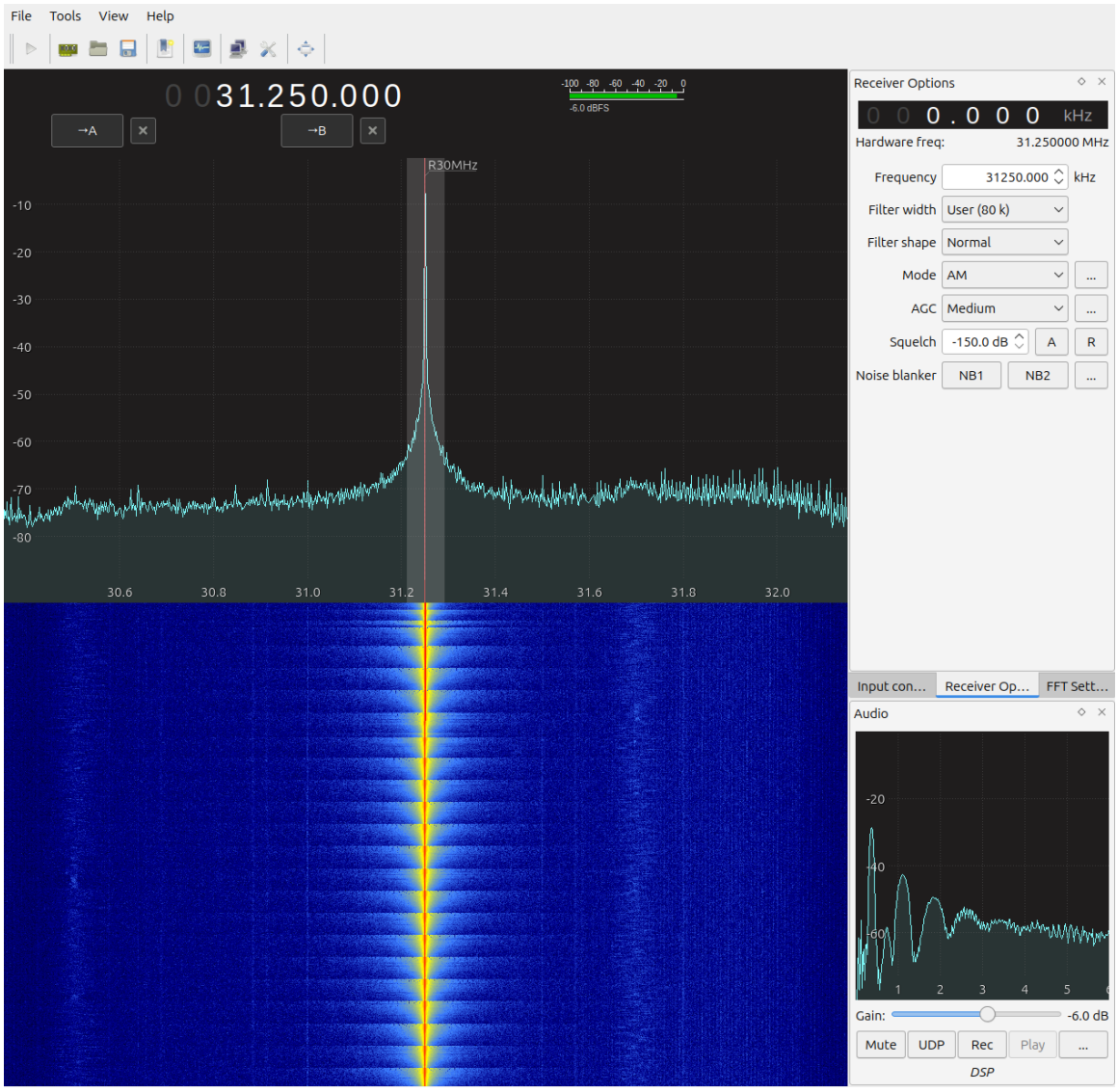


Figure 2: RTL-SDR receiver test.

The RTL-SDR receiver can be used as a computer based radio receiver, with a software defined radio (SDR) controller. The software for the RTL-SDR is community developed and open source, and it is freely available on all the common computer platforms [6]. Also, one can install the RTL-SDR software on a cell phone for creating a mobile receiver (which can be useful in surveillance applications).

Here we will use the GQRX receiver [7] which is powered by the GNU radio [8], but one can use any other receiver available, and obtain similar results. Once the software is properly installed, and the RTL-SDR receiver recognized, one can tune on the carier frequency, or on any of the harmonic frequencies that fall in the bandwidth of GQRX. In my area the signal is relatively clean and strong at the carrier frequency 31.25MHz.

In the Receiver Options of the GQRX, we should select the following options: Frequency: 31250.000KHz, Filter width: User (80k) (drag the window to make it as large as possible), Filter shape: Normal, Mode: AM, AGC: Medium. Also, in the Input Controls tab set the LNA around 24.8 db, No limits, and DC remove. The rest of the controls can remain as they are. You will see a quite strong signal in the FFT (Fast Fourier Transform) window (Figure 2). Even more surprisingly, if you turn on the volume you will hear the sound of the police car siren coming from the speakers of your device (PC or cell phone). Do not panic, this is just the sound of the sweep signal we have generated with the MicroPython program, which ironically sounds like a police car siren. Next, try to tune the frequency to multiples of 31.25MHz, and you will notice that you can hear the same "police car" signal. Now that we have a successful proof of concept we can experiment with some more complex and interesting transmissions.

# 4 Morse code radio transmitter

Using the previously described PWM modulation method we can also transmit Morse code [9], which could be useful for stealth communications. Let us first discuss some details about the Morse code, for the readers who are not familiar with it.

Morse Code is one of the oldest radio communication methods. Basically, the Morse code is a text encoding and decoding method, which can be used in radio transmissions. Using the Morse code, the letters, the numbers and the punctuation characters, can be encoded in units of transmission time dt as following: 1 (dit='.'); 3 (dah='-'); 1 (spacing between dit and dah); 3 (spacing between the characters of a word); 7 (spacing between words). For example the word PARIS consists of 50 units of time, encoded as following:

P='.–.', 1131311, 3 (14 time units)
A='.-', 113, 3 (8 time units)
R='.-.', 11311, 3 (10 time units)
I='..', 111, 3 (6 time units)
S='...', 11111, 7 (12 time units)

One can see that at the end of a word character we insert an inter-character time (3), and at the end of a word we insert an inter-word time (7).

The Morse encoding dictionary (MorseCode) is included in the example Listing 2, where we use MicroPython to transmit "Hello World!" (or potentially any other "disruptive" message) with the carrier frequency of 31.25MHz (and its multiples). As mentioned before, an important parameter is the unit of transmission time, which in this example was set to dt=0.05s, but it can be changed. The program transmits the string 'Hello World!'. First the string is converted to upper case (Morse code only uses upper case letters), then it is encoded into the Morse code using the Morse dictionary and the corresponding time units using the function "morse_encode()". The resulted Morse encoded text is then radio transmitted using the "morse_broadcast()" function, with the same PWM carrier signal of 31.25MHz on GPIO0, modulated with a 600Hz PWM signal on GPIO2 (duty=50%), using the same circuit from Figure 1.

```
from machine import Pin,PWM
import time

MorseCode = {
    'A': '.-', 'B': '-...', 'C': '-.-.', 'D': '-..',
    'E': '.', 'F': '..-.', 'G': '--.', 'H': '....',
    'I': '..', 'J': '.---', 'K': '-.-', 'L': '.-..',
    'M': '--', 'N': '-.', 'O': '---', 'P': '.--.',
    'Q': '--.-', 'R': '.-.', 'S': '...', 'T': '-',
    'U': '..-', 'V': '...-', 'W': '.--', 'X': '-..-',
    'Y': '-.--', 'Z': '--..',
    '0': '-----', '1': '.----', '2': '..---',
    '3': '...--', '4': '....-', '5': '.....',
    '6': '-....', '7': '--...', '8': '---..',
    '9': '----.',
    '.': '.-.-.-', ',': '--..--', '?': '..--..',
    "'": '.----.', '!': '-.-.--', '/': '-..-.',
    '(': '-.--.', ')': '-.--.-', '&': '.-...',
    ':': '---...', ';': '-.-.-.', '=': '-...-',
    '+': '.-.-.', '-': '-....-', '_': '..--.-',
    '"': '.-..-.', '$': '...-..-', '@': '.--.-.',
    ' ': '/',
}

def morse_encode(text):
    x,dt = [],0.05
    for ch in text:
        for c in MorseCode[ch]:
            if c == '-':
                x.append((1,3*dt))
                x.append((0,dt))
            elif c == '.':
                x.append((1,dt))
                x.append((0,dt))
            elif c == '/':
                x.append((0,6*dt))
        x.append((0,2*dt))
    return x

def morse_broadcast(encoded_text):
    led = Pin(25, Pin.OUT) # Pico
#    led = Pin("LED", Pin.OUT) # Pico W
    for x in encoded_text:
        if x[0] == 1:
            led.on()
            pwm1.duty_u16(32767)
        else:
            led.off()
            pwm1.duty_u16(0)
        time.sleep(x[1])
```

```
if __name__ == "__main__":

    text = 'Hello World!'
    text = text.upper()
    print(text)

    encoded_text = morse_encode(text)
    print(encoded_text)

    pwm0 = PWM(Pin(0))
    pwm0.freq(31250000)
    pwm0.duty_u16(32768)

    pwm1 = PWM(Pin(2))
    pwm1.duty_u16(32768)
    pwm1.freq(600)

    while True:
        morse_broadcast(encoded_text)
```

Listing 2: Morse code transmitter.

The next question is: how can we receive and decode the Morse code message? This is a bit more complex problem than it looks, and besides the previously mentioned GQRX, it requires several other software packages, also open source and freely available on any software platform, such as:

- nc [10] - also known as Netcat, is a command-line utility used for reading from and writing to network connections using TCP or UDP (here we will use UDP).
- sox [11] - is a command-line audio processing tool, particularly suited for batch processing.
- multimon-ng [12] - can decode a variety of digital radio transmission modes.

First we need to start GQRX using the same settings (31.25MHz) as before, with the exception of the Filter width, which now should be set to Normal or Narrow. In the Receiver Options select UDP for the communication protocol. Then GQRX will use the port 7355 on the computer to pipe the received audio signal. This signal is then processed using the following command (Listing 3).

```
nc -l -u 7355 | sox -r 48000 -t raw -b 16 -c 1 -e signed-integer /dev/stdin -r 22000
    -t raw -b 16 -c 1 -e signed-integer - | multimon-ng -t raw -a MORSE_CW /dev/stdin
```

Listing 3: Morse code decoding.

and "voila", we can decode the Morse code message radio transmitted by the RP2 (Listing 4).

```
multimon-ng 1.3.0
...
Enabled demodulators: MORSE_CW
HELLO WORLD!HELLO WORLD!HELLO WORLD!HELLO WORLD!HELLO WORLD!HELLO WORLD!HELL
```

Listing 4: Morse code decoded message.

Let us see what the command actually does: (1) nc -l -u 7355, listens for UDP traffic on port 7355 configured in GQRX; (2) sox -r 48000 -t raw -b 16 -c 1 -e signed-integer /dev/stdin -r 22000 -t raw -b 16 -c 1 -e signed-integer -, resamples the audio received from netcat (via the pipe operator) to a sample rate suitable for multimon-ng (e.g., 22000) and ensures the correct audio format; (3) multimon-ng -t raw -a MORSE_CW /dev/stdin, reads raw audio from standard input (/dev/stdin) and activates the MORSE_CW decoder.

Because of the rectangle PWM signals used, the Morse message is disruptive, since it is also transmitted at every multiple of the 31.25MHz frequency. If necessary, the Morse code can be also easily obfuscated (encrypted), for example by randomly shuffling the dictionary. This way, only the receivers that also have the obfuscated dictionary will be able to decode the message.

# 5 Sequencer music broadcasting

We can use the same basic configuration of the RP2 to broadcast sequencer music. To illustrate this approach, here we will use the "buzer_music" package [13], which is a MicroPython library to play music through a buzzer, and replaces chords with fast arpegios to simulate polyphony. Contrary to its initial purpose, we will not use the library to play music through a buzzer, but we will use it to modulate the PWM carrier signal, such that we can broadcast the music on the 31.25MHz frequency (and its multiples). The result could be a radio station playing your favorite sequencer music files. The MicroPython code is given in the Listing 3.

```
from buzzer_music import music
from time import sleep
from machine import Pin, PWM

pwm0 = PWM(Pin(0))
pwm0.freq(31250000)
pwm0.duty_u16(16384)

song = open("seq_music.txt", "r").read()
mySong = music(song[25:-2], pins=[Pin(2)])

while True:
    print(mySong.tick())
    sleep(0.04)
```

Listing 5: Sequencer music broadcasting.

The sequencer music can be download from `onlinesequencer.net`. For example here we will use the song: `(https://onlinesequencer.net/1696155` (Undertale - Heartache). On this webpage we have to click edit, select all notes with CTRL+A and then copy them with CTRL+C, paste the string in a simple text editor (notepad, gedit,...), and save it as a text file "seq_music.txt", then copy the file to the RP2. We should notice that here we set the PWM carrier signal duty to 16384 (25%). Also, we have removed the header (the first 25 characters) and the last 2 characters of the music string before the broadcasting.

# 6 Recorded audio broadcasting

In the previous section we have shown how to broadcast sequencer music, now we will extend this approach to broadcasting recorded audio, for example "wav" files. To illustrate this approach, here we use the "mgm_music.wav" file containing the well known piece of music running at the beginning of the 20th Century Fox and MGM movies [14]. Unfortunately we cannot play directly the file on the RP2. However, here we will use a quite simple workaround, which only requires the conversion of the file into a "raw" file using FFmpeg (Listing 6), which is a well known open source cross-platform solution to record, convert and stream audio and video [15].

```
ffmpeg -i mgm_music.wav -ar 16000 -acodec pcm_u8 -f u8 mgm_music.raw
```

Listing 6: FFmpeg conversion from "wav" to "raw".

Here, "-i" specifies the input file "mgm_music.wav", "-ar 16000" is the sample rate, "-acodec pcm_u8" specifies the audio codec, "-f u8" specifies the 8bit conversion output to "mgm_music.raw". After the "raw" conversion the music has an 8bit representation. We copy the "mgm_music.raw" on the RP2, and we run the program from Listing 4.

```
import time
from machine import Pin, PWM

pwm0 = PWM(Pin(0))
pwm0.freq(31250000)
pwm0.duty_u16(16384)

pwm1 = PWM(Pin(2))
pwm1.freq(1000000)

buf = bytearray(4096)
while True:
    f = open("mgm_music.raw","rb")
    while f.readinto(buf) > 0:
        for sample in buf:
            pwm1.duty_u16(sample<<8)
            time.sleep_us(85)
    f.close()
```

Listing 7: Recorded audio broadcasting.

We continuously read 4096 samples from the file, then each sample is multiplied with $2^8 = 256$, and we use the result to change the duty of the PWM modulator (pwm1). Here, "«" represents the bitwise left shift operator, and in this case we perform an 8 bit left shift, which is equivalent to multiplying the original sample by $2^8 = 256$, but the shift operator is faster than the multiplication. After each sample the program sleeps for $85\mu$s. This duration can be changed if you want to play the file slower or faster. In general, the optimal duration is a function of the sampling rate, so if we change the sampling rate we may have to change the sleep duration. It is surprising to hear how well the music sounds on the radio at 31.25MHz. Instead of music, one can also record, convert, and broadcast any kind of disruptive radio show.

# 7 Microphone radio spying bug

Let us now see how we can transform RP2 into a microphone radio spying bug, or into a live radio transmitter. To accomplish this we will need a microphone, and here we will use a cheap MAX4466 electret microphone module [16] which one can also buy from Amazon for about $1. Obviously the results will improve with a more expensive and better amplified product, but as a proof of concept we prefer to keep it cheap and simple. In Figure 3 we give the circuit schematic. The electret microphone MAX4466 is powered through the 3V3 pin, and grounded on AGND pin, which is the recommended ground for ADC. The output of the microphone is connected to the RP2 on the GPIO27_ADC1. The MicroPython code is given in Listing 8. The rest of the radio transmitter is the same as before.

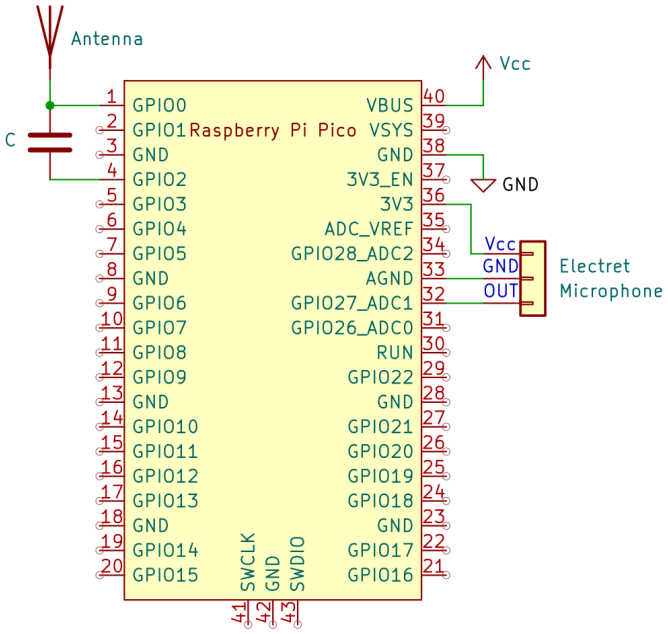


Figure 3: Microphone radio spying bug.

```
from machine import Pin, ADC, PWM

pwm0 = PWM(Pin(0))
pwm0.freq(31250000)
pwm0.duty_u16(32768)

pwm = PWM(Pin(2))
pwm.freq(1000000)

adc = ADC(27)
while True:
    pwm.duty_u16(adc.read_u16())
```

Listing 8: Microphone radio spying bug.

The output of the MAX4466 electret microphone is digitally converted by the ADC using the GPIO27_ADC1 connection, such that the resulted amplitude digital values are used to specify the duty value of the PWM modulator running on GPIO2. This signal is then used to modulate the PWM high frequency signal on GPIO0. The microphone will pick up any local audio (such as conversations) and radio broadcast it, like the radio bugs seen in the spying movies. The result is a surprisingly clear and strong audio reception on 31.25MHz (and its multiples).

# 8 Conclusion

In this paper we have shown how surprisingly simple is to hack the Raspberry Pi Pico (RP2) microcontroller into a disruptive radio transmitter. The techniques described here can be easily used to transform the RP2 into a (potentially clandestine) radio station, or into a microphone radio spying bug, by simply using cheap off the shelf electronic components and free open source software. While in our approach we have limited the range of the transmission to around 10-20m, by using a short 10-20cm antenna wire, the range can be increased by simply increasing the length of the antenna. The optimal length of the antenna can be estimated with the formula $L = 142.6/\nu^*$, where $\nu^*$ is the carrier frequency in MHz. For example, an optimal transmission on $\nu^* = 31.25$MHz, requires a half-wave dipole antenna with a length of approximately 4.56m. Obviously, for some "sensitive" applications this length may be too much, so lengths of 1/4, 1/8, 1/16 the wavelength may be a good starting point. Proper antenna shielding can also be used to enhance the transmission performance, and reduce the interference. If a longer transmission range is required one can use an additional RF amplifier and a better antenna, which are also available on Amazon at low cost.

We should note that while these relatively simple hacking techniques have been illustrated using the RP2, they can be applied to any other microcontroller, or computer board that offers PWM output signals with frequencies in the RF range. In closing, we should stress once again that it is always essential to comply with local regulations regarding radio frequency transmission in order to avoid legal issues.

# 9 Appendix

The Github repository with the code and the data used in the paper is available at:

`https://github.com/mandrecut/rp2_radio_transmitter`

# References


[1] RP2040 Datasheet, Raspberry Pi Ltd. (2024).

[2] Raspberry Pi Pico W Datasheet, Raspberry Pi Ltd. (2024).

[3] Raspberry Pi Pico-series Python SDK, Raspberry Pi Ltd. (2024).

[4] M. Andrecut, *Connecting with the Raspberry Pi Pico*, KDP (2024).

[5] Thonny IDE, https://thonny.org

[6] RTL-SDR, https://www.rtl-sdr.com.

[7] GQRX, https://www.gqrx.dk

[8] GNU radio, https://www.gnuradio.org/

[9] Morse code, https://en.wikipedia.org/wiki/Morse_code

[10] Netcat, https://en.wikipedia.org/wiki/Netcat

[11] SoX, https://en.wikipedia.org/wiki/SoX

[12] multimon-ng, https://github.com/EliasOenal/multimon-ng

[13] buzzer_music, https://github.com/james1236/buzzer_music.

[14] 20th Century Fox theme, https://www.moviesoundclips.net/sound.php?id=147.

[15] FFmpeg, https://ffmpeg.org.

[16] MAX4466, https://www.analog.com/media/en/technical-documentation/data-sheets/max4465-max4469.pdf.