# Python Radio 52: Undercover Adventures

## **An AM Transmitter All in Software**

[Simon Quellen Field](https://medium.com/@simon.field_37276?source=post_page---byline--7c8a8683c5f6---------------------------------------)

[Simon Quellen Field](https://medium.com/@simon.field_37276?source=post_page---byline--7c8a8683c5f6---------------------------------------)

Follow

7 min read

·

2 days ago

Listen

Share

More

Press enter or click to view image in full size

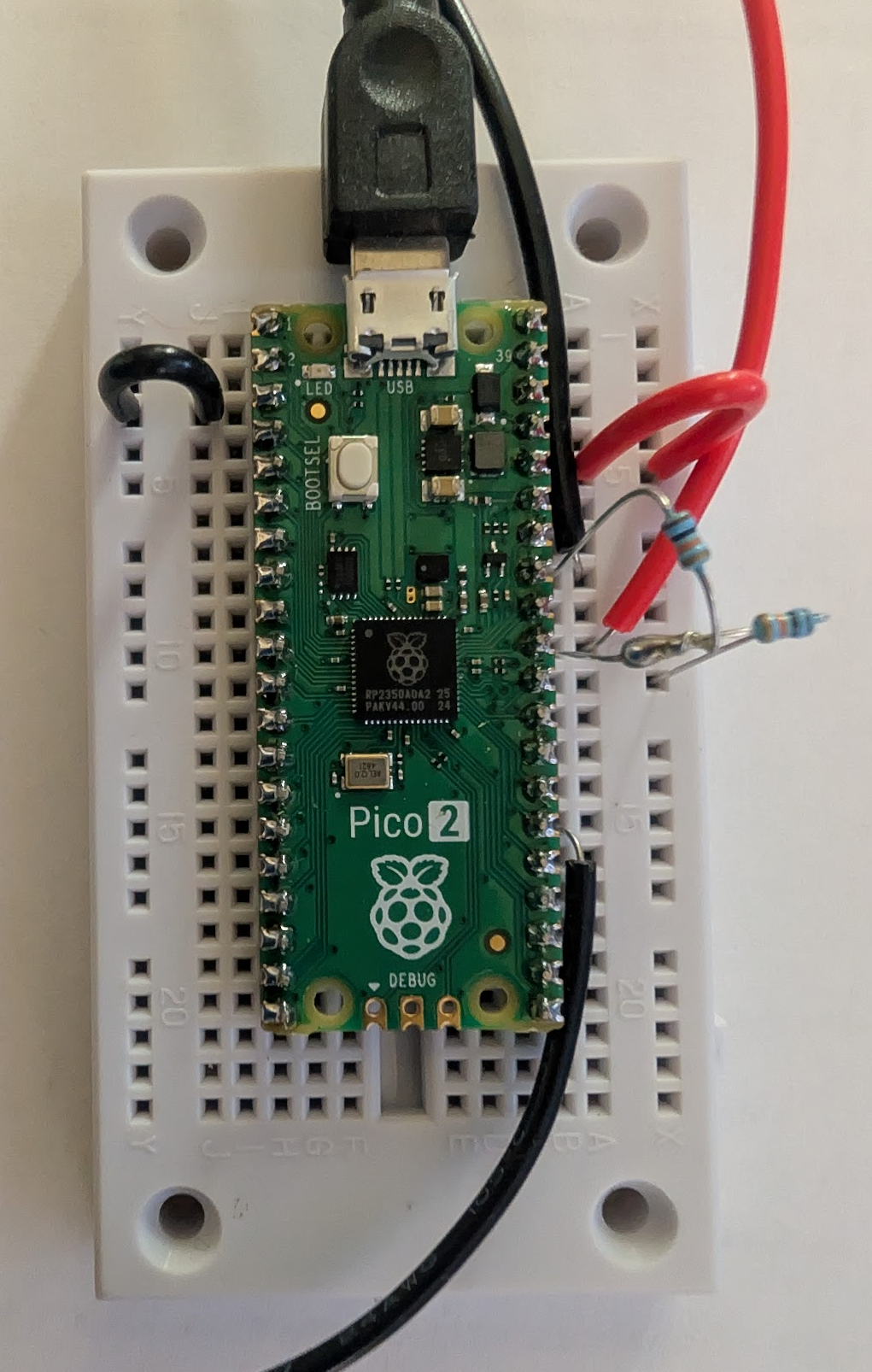


MidJourney

In [Python Radio 51](https://medium.com/radio-hackers/python-radio-51-a-peek-under-the-covers-179f4d3e5bcf), we built fast analog-to-digital and digital-to-analog routines that used DMA, freeing up the CPU to do digital signal processing.

Now we will use those routines to transmit AM radio signals, using nothing more than the $5 Raspberry Pi Pico 2 (the RP2350 and two resistors).

Press enter or click to view image in full size



The Completed AM Radio Transmitter (all photos by the author)

Our Python code uses lessons we learned when building the receiver:

from machine import PWM, ADC, Pin  
import array  
from sdr\_radio import SDR\_Radio, configure\_transmitter\_pwm, audio\_play\_chunk  
from sdr\_radio import audio\_wait\_done, audio\_deinit, deinit\_capture, audio\_configure  
  
machine.freq(250\_000\_000)  
  
# --- Configuration ---  
TRANSMIT\_FREQ = 600\_000  
AUDIO\_SAMPLE\_RATE = 22050  
PWM\_UPDATE\_RATE = 5\_000\_000  
PWM\_TOP = 49 # 250MHz / (5MHz \* (49+1)) = 1.0 divider  
  
AUDIO\_BUFFER\_SAMPLES = 128   
# Calculate how many 32-bit words we need for the PWM buffer  
PWM\_BUFFER\_WORDS = int(AUDIO\_BUFFER\_SAMPLES \* (PWM\_UPDATE\_RATE / AUDIO\_SAMPLE\_RATE))  
  
PWM\_PIN = 20  
ADC\_PIN = 26  
  
sdr = SDR\_Radio()  
sdr.set\_tx\_carrier(TRANSMIT\_FREQ, PWM\_UPDATE\_RATE)  
  
# Audio buffer is uint16\_t ('H')  
# This buffer will be filled by the DMA ADC  
audio\_in\_buf = array.array('H', (0 for \_ in range(AUDIO\_BUFFER\_SAMPLES)))  
  
# PWM buffers are uint32\_t ('L') for DMA compatibility  
pwm\_out\_bufs = [  
 array.array('L', (0 for \_ in range(PWM\_BUFFER\_WORDS))),  
 array.array('L', (0 for \_ in range(PWM\_BUFFER\_WORDS)))  
]  
  
pwm = PWM(Pin(PWM\_PIN))  
adc = ADC(Pin(ADC\_PIN))  
configure\_transmitter\_pwm(pwm, PWM\_UPDATE\_RATE, PWM\_TOP)  
audio\_configure(pwm, AUDIO\_SAMPLE\_RATE)  
  
sdr.configure\_and\_init\_capture(AUDIO\_SAMPLE\_RATE, AUDIO\_BUFFER\_SAMPLES)  
sdr.start\_capture()  
  
print("Starting transmitter... Speak into the microphone.")  
buf\_idx = 0  
  
try:  
 # --- Prime the Pump ---  
 # 1. Capture the first block of audio  
 sdr.capture\_chunk(audio\_in\_buf)  
 # 2. Process it into the first PWM buffer  
 sdr.am\_transmit\_pipeline(audio\_in\_buf, pwm\_out\_bufs[0])  
 # 3. Start playing the first PWM buffer  
 audio\_play\_chunk(pwm, pwm\_out\_bufs[0])  
  
 while True:  
 next\_buf\_idx = 1 - buf\_idx  
  
 # 1. Capture the next block of audio. This happens while the  
 # previous PWM buffer is still being transmitted by DMA.  
 sdr.capture\_chunk(audio\_in\_buf)  
  
 # 2. Process the newly captured audio into the \*other\* PWM buffer.  
 sdr.am\_transmit\_pipeline(audio\_in\_buf, pwm\_out\_bufs[next\_buf\_idx])  
  
 # 3. Wait for the PWM transmission (started in the previous loop) to complete.  
 audio\_wait\_done(pwm)  
  
 # 4. Immediately start transmitting the PWM buffer we just prepared.  
 audio\_play\_chunk(pwm, pwm\_out\_bufs[next\_buf\_idx])  
  
 buf\_idx = next\_buf\_idx  
  
except KeyboardInterrupt:  
 print("\nStopping transmitter.")  
finally:  
 audio\_deinit(pwm)  
 deinit\_capture()

Like the receiver, we use ping-pong buffers to allow the DMA to happen in the background while we process the inputs on the CPU.

The ADC ping-pong happens in the C code. We manage the PWM ping-pong buffers in Python.

The RP2350 can easily run at 250 MHz without needing any extra cooling or heat syncs, so we set that up right away. For our transmitting frequency, we chose 600 kHz somewhat arbitrarily (there were no local stations on that frequency in my area).

We will run our PWM at 5 MHz. This will give us 8 and a third cycles of PWM for every cycle of our target frequency (600 kHz), allowing us to set our PWM duty cycle to any number between 0 and 49. With 50 levels between 0 volts and 3.3 volts, we can generate a sine wave that has only a small bit of high harmonics that are easy to filter out.

As with the receiver, the parts that need to be very fast are done in C, and linked into the Micropython runtime. Besides the DMA for the input and output, those parts are configure\_transmitter\_pwm(), set\_tx\_carrier(), and am\_transmit\_pipeline():

static mp\_obj\_t configure\_transmitter\_pwm(mp\_obj\_t pwm\_obj, mp\_obj\_t update\_rate\_obj, mp\_obj\_t top\_obj) {  
 machine\_pwm\_obj\_t \*pwm = MP\_OBJ\_TO\_PTR(pwm\_obj);  
 uint32\_t update\_rate = mp\_obj\_get\_int(update\_rate\_obj);  
 uint32\_t top = mp\_obj\_get\_int(top\_obj);  
  
 pwm\_set\_enabled(pwm->slice, false);  
  
 uint32\_t source\_hz = clock\_get\_hz(clk\_sys);  
 float div = (float)source\_hz / ((float)(top + 1) \* (float)update\_rate);  
 if (div < 1.0f) div = 1.0f;  
   
 pwm\_set\_clkdiv(pwm->slice, div);  
 pwm\_set\_wrap(pwm->slice, top);  
  
 pwm\_set\_chan\_level(pwm->slice, pwm->channel, 0);  
 pwm\_set\_enabled(pwm->slice, true);  
  
 audio\_is\_configured = true;   
 return mp\_const\_none;  
}  
static MP\_DEFINE\_CONST\_FUN\_OBJ\_3(configure\_transmitter\_pwm\_obj, configure\_transmitter\_pwm);  
  
  
static mp\_obj\_t sdr\_radio\_set\_tx\_carrier(mp\_obj\_t self\_in, mp\_obj\_t freq\_obj, mp\_obj\_t pwm\_rate\_obj) {  
 sdr\_radio\_obj\_t \*self = MP\_OBJ\_TO\_PTR(self\_in);  
 self->tx\_carrier\_freq\_hz = mp\_obj\_get\_int(freq\_obj);  
 uint32\_t pwm\_update\_rate = mp\_obj\_get\_int(pwm\_rate\_obj);  
  
 self->tx\_nco\_phase\_increment = (q31\_t)(((uint64\_t)self->tx\_carrier\_freq\_hz << 31) / pwm\_update\_rate);  
  
 return mp\_const\_none;  
}  
  
static MP\_DEFINE\_CONST\_FUN\_OBJ\_3(sdr\_radio\_set\_tx\_carrier\_obj, sdr\_radio\_set\_tx\_carrier);  
  
  
  
# define AUDIO\_GAIN 3.0f;  
  
static mp\_obj\_t sdr\_radio\_am\_transmit\_pipeline(mp\_obj\_t self\_in, mp\_obj\_t audio\_buf\_obj, mp\_obj\_t pwm\_buf\_obj) {  
 sdr\_radio\_obj\_t \*self = MP\_OBJ\_TO\_PTR(self\_in);  
  
 mp\_buffer\_info\_t audio\_info; mp\_get\_buffer\_raise(audio\_buf\_obj, &audio\_info, MP\_BUFFER\_READ);  
 mp\_buffer\_info\_t pwm\_info; mp\_get\_buffer\_raise(pwm\_buf\_obj, &pwm\_info, MP\_BUFFER\_WRITE);  
  
 uint16\_t \*audio\_in\_ptr = (uint16\_t \*)audio\_info.buf;  
 uint32\_t \*pwm\_out\_ptr = (uint32\_t \*)pwm\_info.buf;   
  
 const int num\_audio\_samples = audio\_info.len / sizeof(uint16\_t);  
 const int num\_pwm\_words = pwm\_info.len / sizeof(uint32\_t);   
   
 // --- Constants ---  
 const int PWM\_TOP = 49; // Must match Python  
 const int PWM\_CENTER = 25; // (PWM\_TOP + 1) / 2  
 const float MODULATION\_DEPTH = 0.99f;  
  
 // These must match your Python script's constants  
 const uint32\_t PWM\_UPDATE\_RATE = 5000000;  
 const float RATIO = (float)PWM\_UPDATE\_RATE / (float)AUDIO\_SAMPLE\_RATE;  
  
 int pwm\_idx = 0;  
  
 // --- Main Processing Loop ---  
 for (int i = 0; i < num\_audio\_samples; i++) {  
 // 1. Get current and next audio sample for linear interpolation  
 float audio\_start = ((float)audio\_in\_ptr[i] - 2048.0f) / 2048.0f;  
 float audio\_end = (i + 1 < num\_audio\_samples) ?  
 ((float)audio\_in\_ptr[i + 1] - 2048.0f) / 2048.0f :  
 audio\_start;  
  
 audio\_start \*= AUDIO\_GAIN;  
 audio\_end \*= AUDIO\_GAIN;  
  
 // 2. Linear Interpolation loop  
 // Calculate how many PWM samples this one audio sample covers  
 int start\_j\_idx = (int)(i \* RATIO);  
 int end\_j\_idx = (int)((i + 1) \* RATIO);  
  
 for (int pwm\_sample\_idx = start\_j\_idx; pwm\_sample\_idx < end\_j\_idx; pwm\_sample\_idx++) {  
 if (pwm\_idx >= num\_pwm\_words) break; // Safety break  
  
 float interp\_point = (float)(pwm\_sample\_idx - start\_j\_idx) / (float)(end\_j\_idx - start\_j\_idx);  
 float audio\_interp = audio\_start \* (1.0f - interp\_point) + audio\_end \* interp\_point;  
  
 // 3. Generate carrier sample  
 q31\_t carrier\_q31 = arm\_cos\_q31(self->tx\_nco\_phase);  
 self->tx\_nco\_phase += self->tx\_nco\_phase\_increment;  
 float carrier\_float = (float)carrier\_q31 / 2147483648.0f;  
  
 // 4. Modulate  
 float modulator = 1.0f + (audio\_interp \* MODULATION\_DEPTH);  
 float am\_signal = carrier\_float \* modulator;  
  
 // 5. Scale to PWM duty cycle  
 int32\_t duty\_cycle = (int32\_t)(PWM\_CENTER \* (1.0f + am\_signal));  
   
 // 6. Clamp  
 if (duty\_cycle > PWM\_TOP) duty\_cycle = PWM\_TOP;  
 if (duty\_cycle < 0) duty\_cycle = 0;  
  
 // 7. Pack two 16-bit duty cycles into one 32-bit word for the DMA  
 // PWM B gets silence (25).  
 // PWM A gets the duty\_cycle  
 pwm\_out\_ptr[pwm\_idx] = ((uint32\_t)25 << 16) | (uint32\_t)duty\_cycle;  
 pwm\_idx++;  
 }  
 }  
 return mp\_const\_none;  
}  
  
static MP\_DEFINE\_CONST\_FUN\_OBJ\_3(sdr\_radio\_am\_transmit\_pipeline\_obj, sdr\_radio\_am\_transmit\_pipeline);

We configure the PWM for 5 MHz and 0–49 levels using configure\_transmitter\_pwm().

We set the target frequency for the carrier wave to 600 kHz using set\_tx\_carrier(). This function calculates the phase increment we will use to generate a 600 kHz sine wave.

The real work is done in am\_transmit\_pipeline().

It takes as input the ADC buffer we collect from the analog-to-digital converter. In this case, I am using input from the sound card on my computer, which ranges from -1 volt to 1 volt.

Because the ADC can’t see voltages less than zero, we use two 10kΩ resistors to make a voltage divider. We connect them in series from the 3.3-volt positive power supply to ground.

The center between the two resistors is where we find half the 3.3 volts (1.65 volts). We connect this to the ADC input. Now, when the sound card voltages come in, they get added and subtracted from 1.65 volts to give us a range of 0.65 to 2.65 volts, which is (nearly) perfect for the ADC.

The ADC now gives us numbers in the range of zero to 4095. The middle is 2048, so by subtracting that amount and then dividing by 2048, we get numbers between -1 and 1. We have just undone the work of the two resistors.

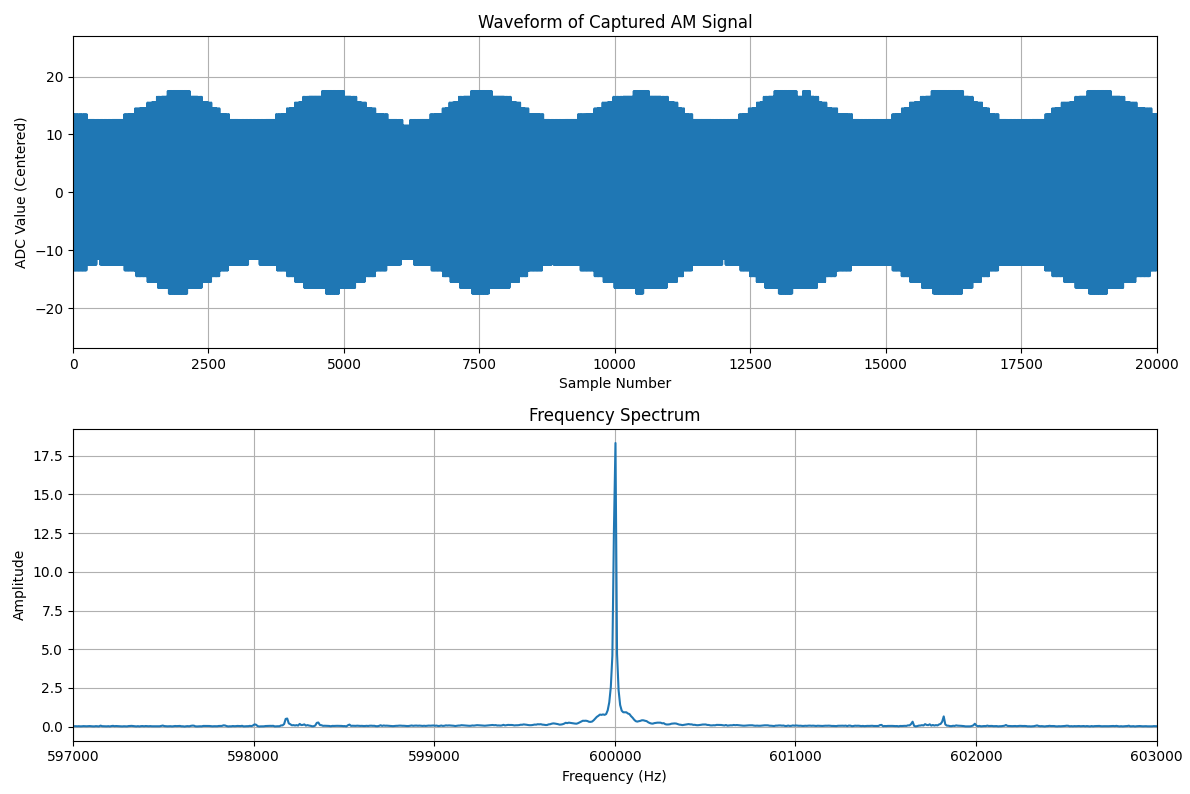
The numbers I got were a little smaller than I liked, so I multiplied them by 3 (AUDIO\_GAIN) to get them closer to 0 to 4095. I could instead have amplified the signal before giving it to the RP2350, but I was aiming for a minimal parts count.

The main loop in am\_transmit\_pipeline() is a linear interpolator, which smooths out the jumps between the samples, making it appear that we were sampling at 5 MHz instead of 22050 Hz.

We use the phase increment we established when tuning to walk through the loop, creating the carrier sine wave. We multiply that by the interpolated modulation signal from the ADC buffer, giving us an AM-modulated carrier wave.

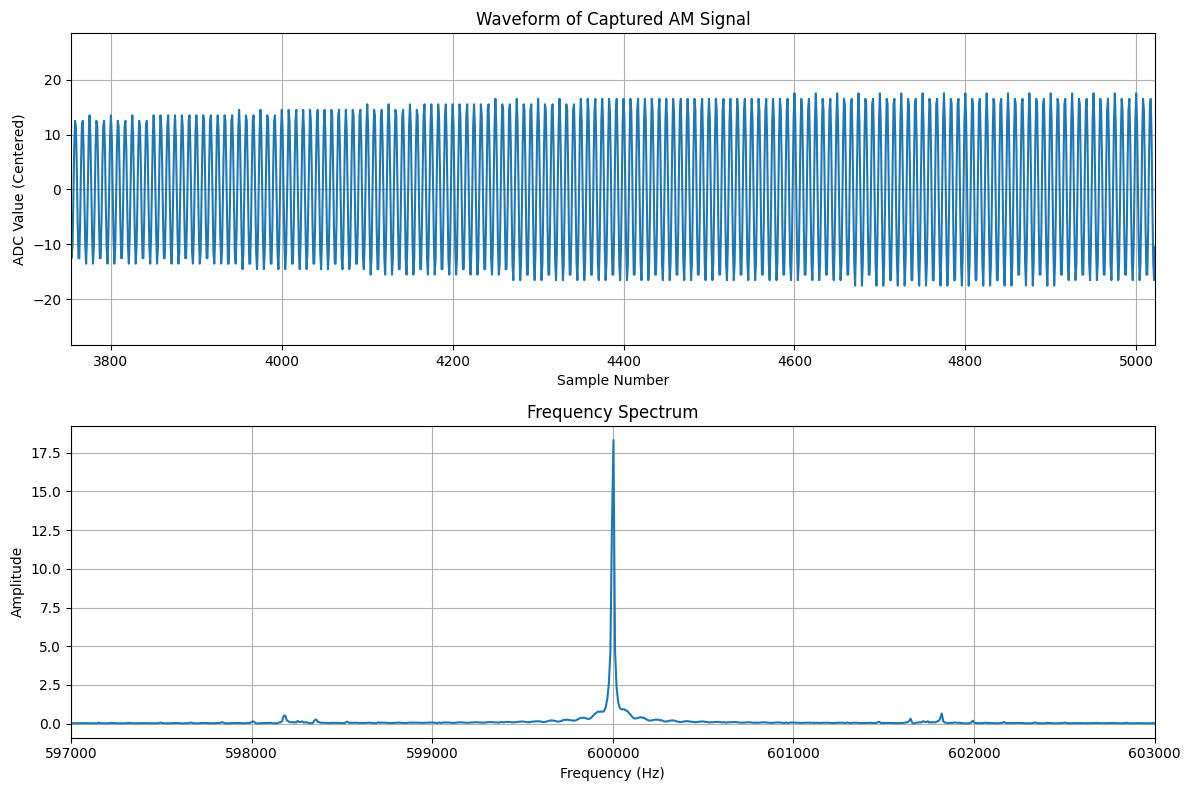
Our final output looks like this:

Press enter or click to view image in full size



In this run, we didn’t max out the volume from the sound card, so the modulation depth does not go from zero to 4095. All that solid blue in the top graph is actually a sine wave, as you can see when we zoom in:

Press enter or click to view image in full size



That low modulation depth shows up in the small amount of energy in the sidebands, as shown in the frequency spectrum. AM modulation has a peak in the center, at the carrier frequency, and two (in this case, tiny) sidebands that carry the information content.

I sent the transmitter a 1600 Hz sine wave as my input. That’s why the two sidebands are 1600 Hz from the carrier. Voice or music would show much broader sidebands.

So there you have it — an AM transmitter done all in software. And you get to see how MicroPython works under the covers using fast routines in C to deal with hardware and to speed up digital signal processing (DSP).

[Programming](https://medium.com/tag/programming?source=post_page-----7c8a8683c5f6---------------------------------------)

[Software Development](https://medium.com/tag/software-development?source=post_page-----7c8a8683c5f6---------------------------------------)

[Python](https://medium.com/tag/python?source=post_page-----7c8a8683c5f6---------------------------------------)

[Radio](https://medium.com/tag/radio?source=post_page-----7c8a8683c5f6---------------------------------------)

[Digital Signal Processing](https://medium.com/tag/digital-signal-processing?source=post_page-----7c8a8683c5f6---------------------------------------)