# Python Radio 10: Another 40-meter transmitter

[Simon Quellen Field](https://medium.com/@simon.field_37276?source=post_page---byline--58f94c8a0e94---------------------------------------)

[Simon Quellen Field](https://medium.com/@simon.field_37276?source=post_page---byline--58f94c8a0e94---------------------------------------)

Follow

5 min read

·

Aug 27, 2024

Listen

Share

More

This time using the Raspberry Pi Pico 2040

Press enter or click to view image in full size

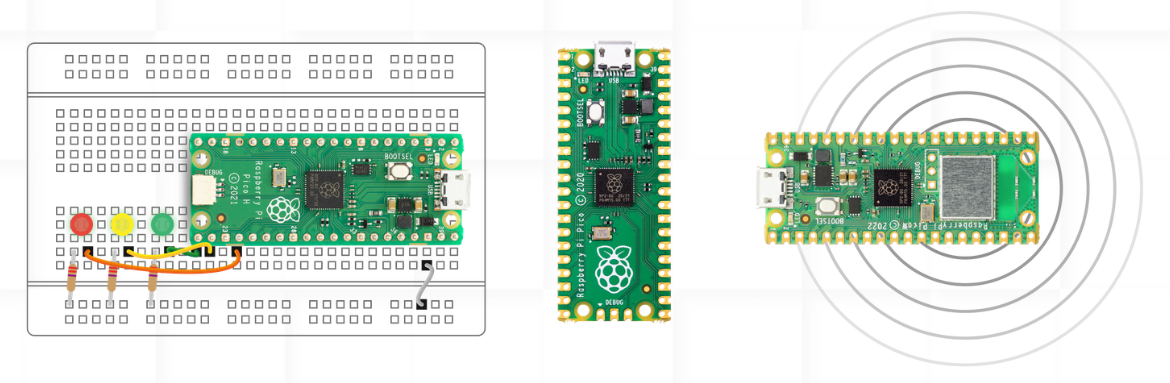


Image by Raspberry Pi

When we built the 40-meter CW transmitter using the ESP32 we noted that the PWM feature could not reach all frequencies in the amateur radio bands because it had to divide its 80 MHz timer clock by integers.

The $4 Raspberry Pi Pico 2040 has a PWM timer that runs at the system clock frequency (nominally 133 MHz but you can change that). This gives us these frequencies in the amateur radio bands:

**160-meter band:** 1811594, 1824818, 1838235, 1851852, 1865672, 1879699, 1893939, 1908397, 1923077, 1937984, 1953125, 1968504, 1984127, 2000000,  
**80-meter band:**3521127,3571429, 3623188, 3676471, 3731343, 3787879, 3846154, 3906250, 3968254,  
**40-meter band:** 7142857, 7352941,  
**20-meter band:** 14705882,  
**10-meter band:** **27777778**

Things look a little sparse. Only one frequency in the areas where technicians can operate.

But, maybe all is not lost.

Since we can change the system clock frequency, we can hunt for values of the clock frequency that give us legal radio frequencies. A program to do this looks like this:

from machine import Pin, PWM, freq  
from time import ticks\_ms, ticks\_diff  
  
def main():  
 pin = Pin(15, Pin.OUT)  
 pwm = PWM(pin, freq=1800000)  
 pwm.duty\_u16(32025)  
  
 bands = (  
 ( 1800000, 2000000), # 160 meters  
 ( 3500000, 4000000), # 80 meters  
 ( 7000000, 7300000), # 40 meters  
 ( 10100000, 10150000), # 30 meters  
 ( 14000000, 14350000), # 20 meters  
 ( 18068888, 18168000), # 17 meters  
 ( 21000000, 21450000), # 15 meters  
 ( 24890000, 24990000), # 12 meters  
 ( 28000000, 29700000), # 10 meters  
 ( 50000000, 54000000), # 6 meters  
 (144000000, 148000000) # 2 meters  
 )  
  
 t = ticks\_ms()  
 results = {}  
 guess = 0  
 old\_len = 0  
 for x in bands:  
 f\_lo, f\_hi = x  
 print(f\_lo)  
 for f in range(f\_lo, f\_hi):  
 for clock\_mul in range(2, 21):  
 freq(12000000 \* clock\_mul)  
 pwm.freq(f)  
 actual = pwm.freq()  
 if actual != guess and actual > f\_lo and actual < f\_hi:  
 results[str(actual)] = clock\_mul  
 if len(results) != old\_len:  
 print(str(round(ticks\_diff(ticks\_ms(), t) / 1000.0)) + “:”, results)  
 old\_len = len(results)  
 guess = actual  
 print(results)  
  
main()

After running for over five minutes, it gives us this line:

325: {’1815126’: 18, ‘1806452’: 14, ‘1821429’: 17, ‘1824000’: 19, ‘1833333’: 11, ‘1837838’: 17, ‘1838710’: 19, ‘1809524’: 19, ‘1811321’: 8, ‘1830508’: 18, ‘1846154’: 8, ‘1808219’: 11, ‘1805310’: 17, ‘1835294’: 13, ‘1804511’: 20, ‘1828571’: 16, ‘1813953’: 13, ‘1822785’: 12, ‘1826087’: 14, ‘1818182’: 10, ‘1832061’: 20, ‘1836735’: 15}

This requires a bit of explanation. We can’t just set the system clock to any frequency willy-nilly. The board has a 12 MHz crystal as a timebase and the chip multiplies that by some integer factor between 2 and 21 to create the system clock. That output line tells us that if we set the system clock to 18 times 12 MHz, we can set the PWM to 1815126 and it will stay there. The same goes for the other 21 values.

It might take all night to finish running the program. Or it might still be running next week.

Maybe there’s a better way.

The following program will send beeps to a radio in CW mode on the frequency 7030000 Hertz:

class RP\_CW:  
 def \_\_init\_\_(self, pin):  
 from machine import Pin  
 from rp2 import PIO, StateMachine, asm\_pio  
  
 @asm\_pio(set\_init=PIO.OUT\_LOW)  
 def square():  
 wrap\_target()  
 set(pins, 1)  
 set(pins, 0)  
 wrap()  
  
 self.pin = Pin(pin, Pin.OUT)  
 self.f = 7030000  
 self.sm = rp2.StateMachine(0, square, freq=2\*self.f, set\_base=self.pin)  
  
 def on(self):  
 self.sm.active(1)  
  
 def off(self):  
 self.sm.active(0)  
   
 def frequency(self, frq):  
 self.f = frq  
   
def main():  
 from time import sleep\_ms  
   
 cw = RP\_CW(15)  
   
 while True:  
 cw.on()  
 sleep\_ms(100)  
 cw.off()  
 sleep\_ms(100)  
   
main()

The program uses a feature of the RP2040 called the PIO. That is a set of programmable state machines (rudimentary computers) that have their own instruction set and can be controlled by either of the two main processors in the chip.

There are two main parts to explain. The @asm\_pio and the square method that follows it define two instructions for the state machine. The first sets the pin to high. The second sets it too low. That is all that happens there.

the rp2.StateMachine line says to run the square program on state machine 0, at a rate of 2 \* self.f (since there are two instructions in the program, and we want a square wave at the frequency self.f on the pin self.pin.

Once we have set up our state machine, we can activate it and deactivate it to control whether or not we are transmitting.

With this little bit of code, we now have access to every amateur radio frequency in the HF bands, with single Hertz resolution.

Our CW transmitter program now looks like this:

from machine import Pin, PWM  
from rp2 import PIO, StateMachine, asm\_pio  
  
class RP\_CW:  
 def \_\_init\_\_(self, pin):  
 from machine import Pin  
 from rp2 import PIO, StateMachine, asm\_pio  
  
 @asm\_pio(set\_init=PIO.OUT\_LOW)  
 def square():  
 wrap\_target()  
 set(pins, 1)  
 set(pins, 0)  
 wrap()  
  
 self.pin = Pin(pin, Pin.OUT)  
 self.f = 7030000  
 self.sm = rp2.StateMachine(0, square, freq=2\*self.f, set\_base=self.pin)  
  
 def on(self):  
 self.sm.active(1)  
  
 def off(self):  
 self.sm.active(0)  
   
 def frequency(self, frq):  
 self.f = frq  
  
class CWMorse:  
 character\_speed = 18  
  
 def \_\_init\_\_(self, pin, freq):  
 self.cw = RP\_CW(pin)  
 self.cw.frequency(freq)  
   
 def speed(self, overall\_speed):  
 if overall\_speed >= 18:  
 self.character\_speed = overall\_speed  
 units\_per\_minute = int(self.character\_speed \* 50) # The word PARIS is 50 units of time  
 OVERHEAD = 2  
 self.DOT = int(60000 / units\_per\_minute) - OVERHEAD  
 self.DASH = 3 \* self.DOT  
 self.CYPHER\_SPACE = self.DOT  
  
 if overall\_speed >= 18:  
 self.LETTER\_SPACE = int(3 \* self.DOT) - self.CYPHER\_SPACE  
 self.WORD\_SPACE = int(7 \* self.DOT) - self.CYPHER\_SPACE  
 else:  
 # Farnsworth timing from “https://www.arrl.org/files/file/Technology/x9004008.pdf”  
 farnsworth\_spacing = (60000 \* self.character\_speed - 37200 \* overall\_speed) / (overall\_speed \* self.character\_speed)  
 farnsworth\_spacing \*= 60000/68500 # A fudge factor to get the ESP8266 timing closer to correct  
 self.LETTER\_SPACE = int((3 \* farnsworth\_spacing) / 19) - self.CYPHER\_SPACE  
 self.WORD\_SPACE = int((7 \* farnsworth\_spacing) / 19) - self.CYPHER\_SPACE  
  
 def send(self, str):  
 from the\_code import code  
 from time import sleep\_ms  
 for c in str:  
 if c == ‘ ‘:  
 self.cw.off()  
 sleep\_ms(self.WORD\_SPACE)  
 else:  
 cyphers = code[c.upper()]  
 for x in cyphers:  
 if x == ‘.’:  
 self.cw.on()  
 sleep\_ms(self.DOT)  
 else:  
 self.cw.on()  
 sleep\_ms(self.DASH)  
 self.cw.off()  
 sleep\_ms(self.CYPHER\_SPACE)  
 self.cw.off()  
 sleep\_ms(self.LETTER\_SPACE)  
  
from time import sleep  
  
frequency = 7030000  
  
def main():  
 cw = CWMorse(15, frequency)  
 cw.speed(5)  
 print(”CW transmitter”)  
 msg = “AB6NY testing RP2040 as a 40 meter transmitter sending on “ + str(frequency) + “ Hertz.”  
 while True:  
 print(msg)  
 cw.send(msg)  
 sleep(5)  
  
main()

My RP2040 without any amplification was putting out 3.8 milliwatts (5.8 dBm) after the low pass filter. That tiny signal could be heard several miles away, even though the antenna was just a 6-foot whip tuned for 40 meters.

While the RP2040’s PIO processor allows a much better range of frequencies than its or the ESP32’s PWM feature, it is still based on a clock that is divided by integers. We can’t get single-hertz resolution with it.

[Raspberry Pi](https://medium.com/tag/raspberry-pi?source=post_page-----58f94c8a0e94---------------------------------------)

[Raspberry Pi Pico](https://medium.com/tag/raspberry-pi-pico?source=post_page-----58f94c8a0e94---------------------------------------)

[Electronics](https://medium.com/tag/electronics?source=post_page-----58f94c8a0e94---------------------------------------)

[Radio](https://medium.com/tag/radio?source=post_page-----58f94c8a0e94---------------------------------------)

[Python](https://medium.com/tag/python?source=post_page-----58f94c8a0e94---------------------------------------)