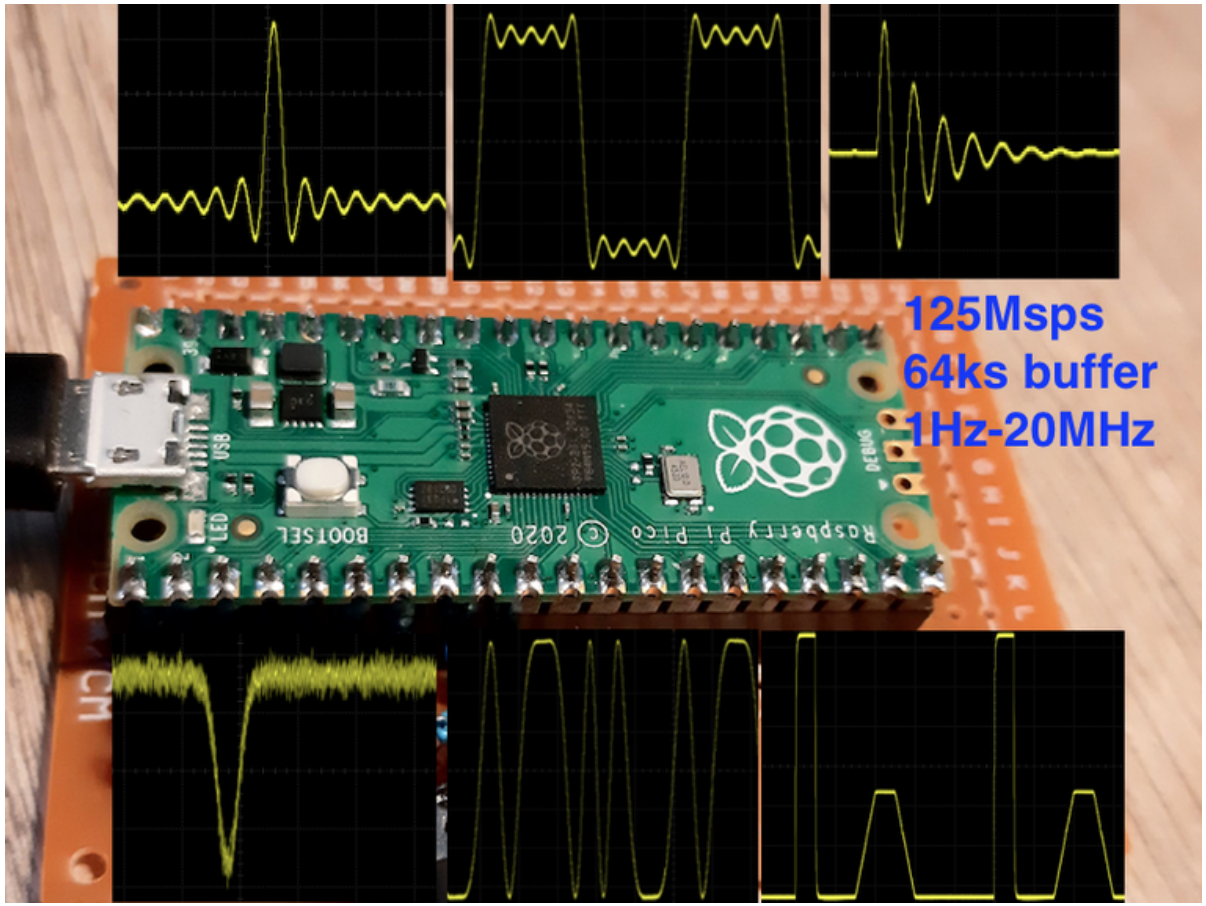
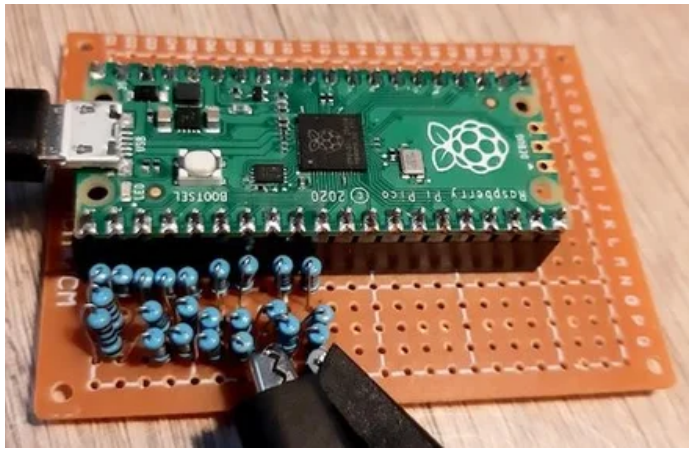
AUTODESK
Instructables

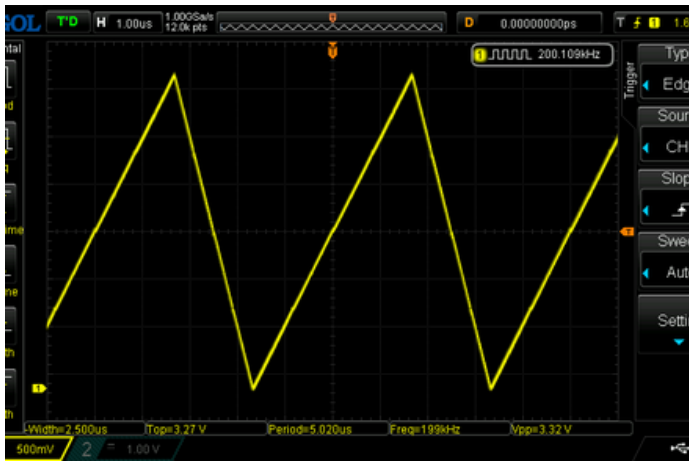
Arbitrary Wave Generator With the Raspberry Pi Pico

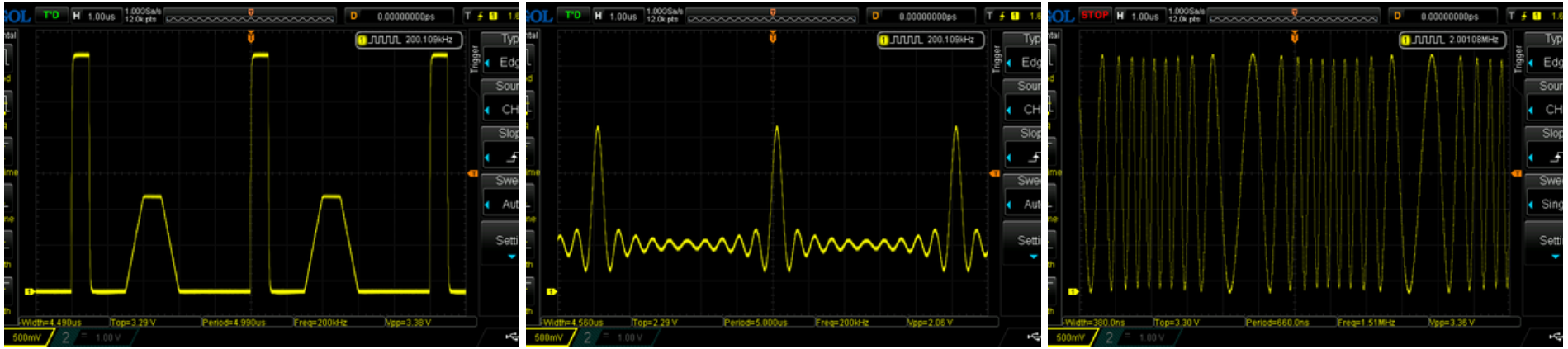
By rgco in CircuitsMicrocontrollers

Introduction: Arbitrary Wave Generator With the Raspberry Pi Pico









Just two weeks ago, the pico, a new microcontroller, the pico, was released by the Raspberry Pi Foundation, well known for the incredibly successful series of Raspberry Pi single-board computers. The new microcontroller uses a brand new chip, designed in-house, the RP2040. It has two 32-bit cores running by default at 125MHz. It has been criticised for not having Wifi or Bluetooth, and no hardware floating point math. But it has a very fast internal bus and powerful peripherals. It has been designed for makers and has very strong support: it was released with 6 detailed datasheets and a beginner's guide book, which is available free of charge as pdf. Best of all, it is cheap at $4. I got 5 for under 30EUR including shipping.

As a test I wanted to see if any of my previous projects based on the Arduino Uno/Nano could benefit from a remake with this much more powerful board: After all it has 4x the bus width, 8x the clock frequency, 130x the RAM, and is more than a decade more modern. My choice fell on the Arbitrary waveform generator (AWG). With the Arduino, I managed to squeeze out 381ksps, since every sample update took 42 instruction cycles, mostly because updating a 32-bit phase counter takes a quadruple loop with an 8-bit CPU. My expectation was that it should be possible to improve this by a factor 8 just from clock speed and maybe another factor 2 because the new board is 32-bit. However, after reading selected parts of the 637-page datasheet of the new RP2040 chip, I realised it might be possible to update every single clock cycle! Just by initialising 2 peripherals, the DMA (Direct Memory Access) and the PIO (programmable Input/Output), an array can be cyclically streamed to the output pins.

Indeed, it works, and the increase in speed with the Arduino is more than a factor 300, from 381ksps to 125Msps. That is similar to serious lab-AWGs, which cost ~100EUR for budget models. There is no attempt here to provide a buffer or amplifier for the produced signal, it is beyond my skills and my equipment to come up with a buffer/amplifier beyond the 10MHz range. The produced signal is thus rather weak, with an output impedance of ~1kOhm, and a maximum current draw of ~1mA. Suggestions for a buffer/amplifier are welcome in the comments! There is no attempt either to provide a dedicated user interface in terms of a screen, buttons, rotary encoders etc. That adds cost and complexity. I found it is much more convenient and much more powerful to set the requested waveform in the micropython code itself!

For comparison, several instructables (e.g. here, here and here) describe how to make a function generator based on the dedicated AD9833 chip. This chip runs at 25Msps and can generate only 3 predefined waveforms: sine, triangle and square. The pico is 5x fast

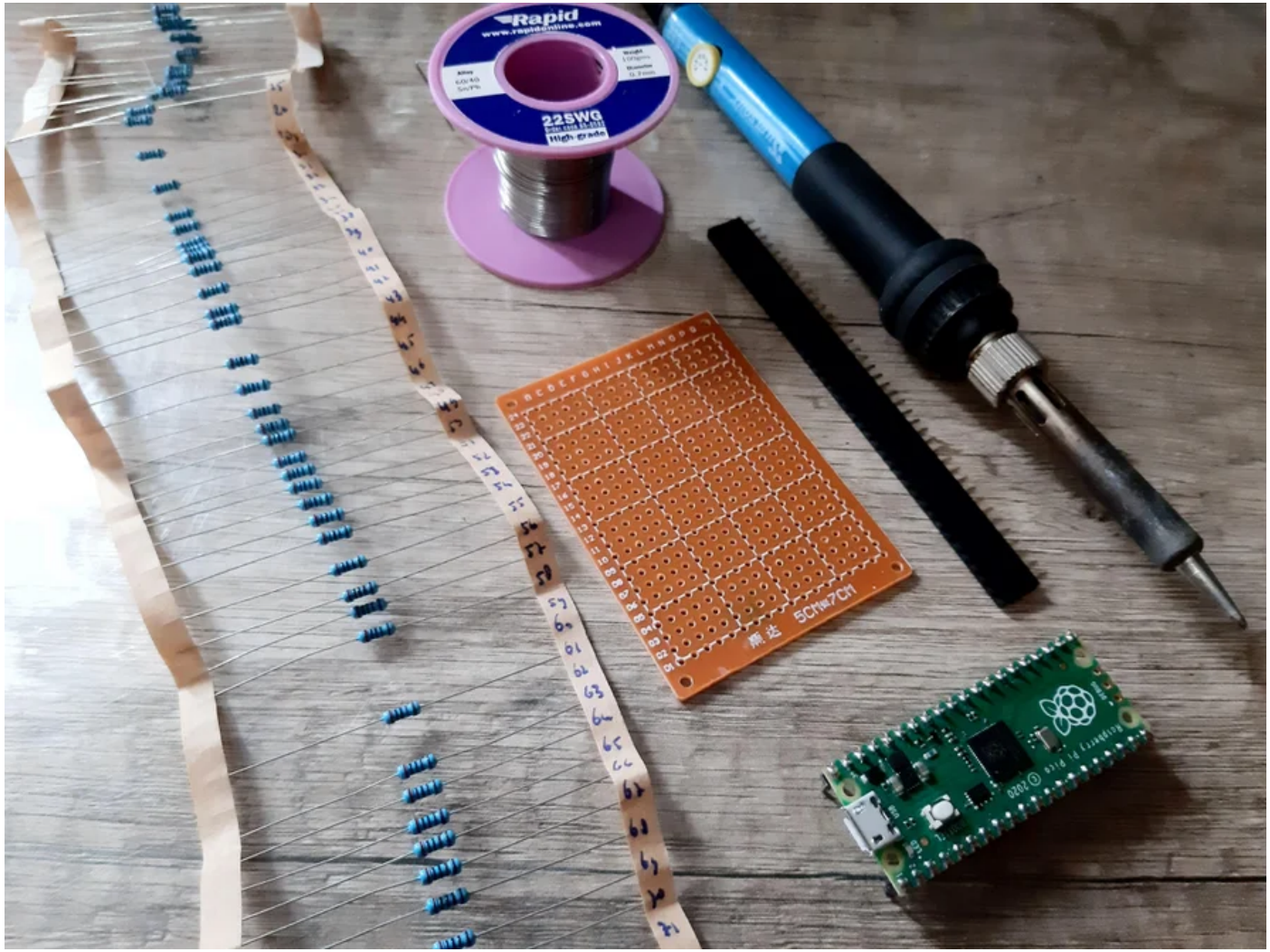
er and can generate any possible wave that fits in an array, up to many thousands of points.

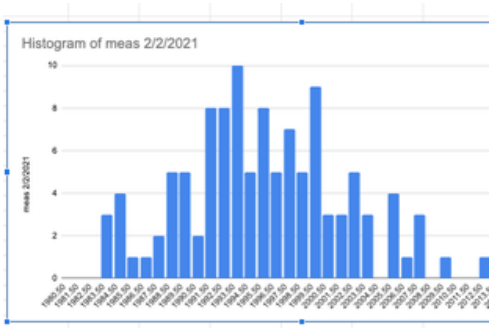
Supplies

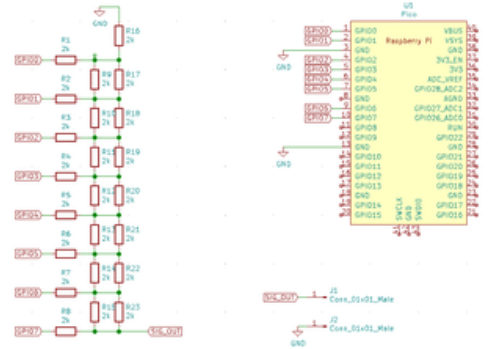
Required materials:

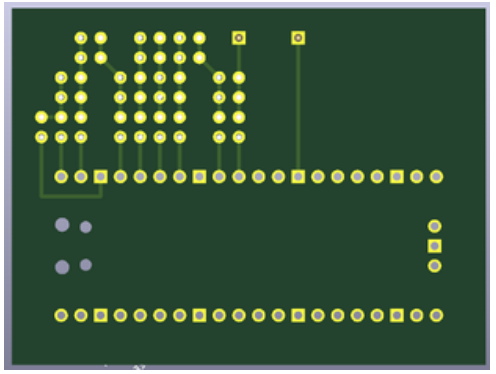
- Raspberry pi pico microcontroller with male pin headers
- 1 5x7cm prototype board
- 2 20-pin female pin headers
- 23 resistors of identical value, near 2kOhm

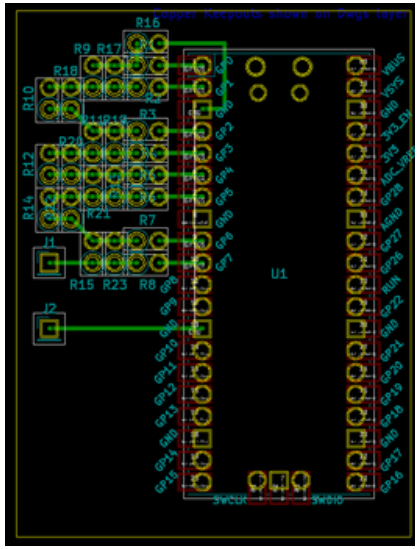
Step 1: Construction of the R2R DAC

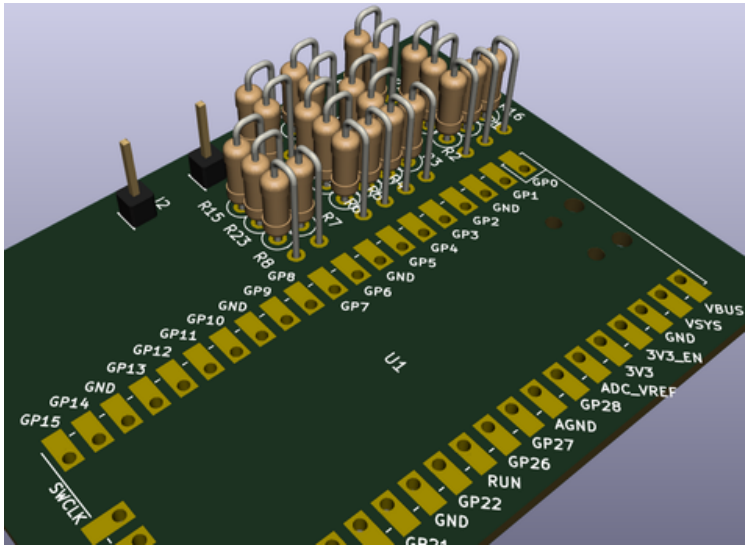


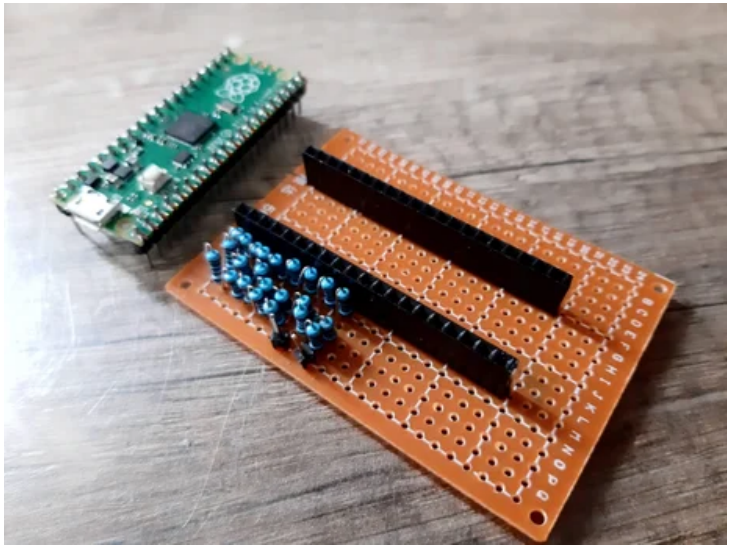












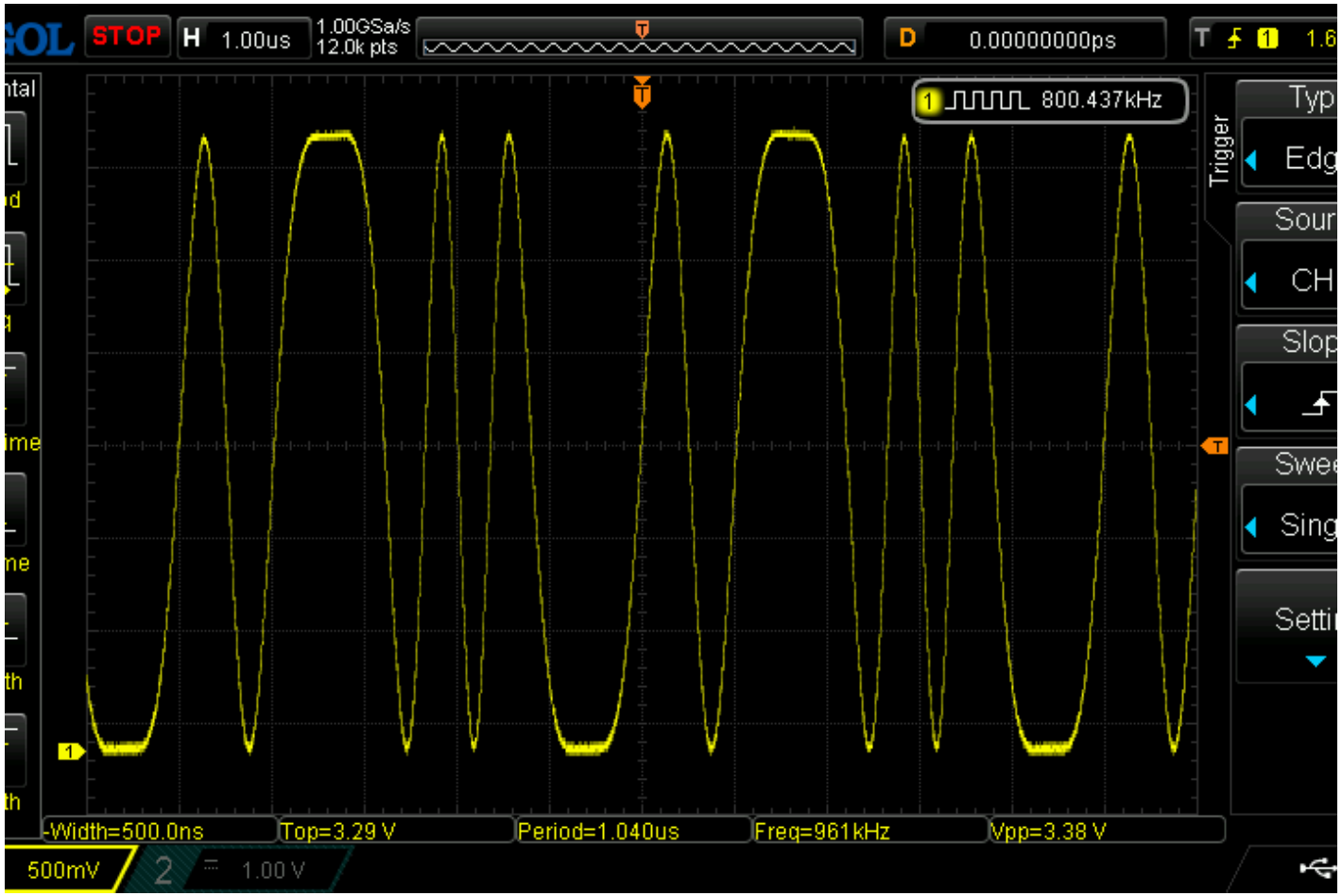
The pico itself cannot produce analog signals, it does not have a digital-to-analog converter (DAC). But they are simple to build from r esistors. We use here the classic R2R DAC. An 8-bit DAC requires 7 resistors of value R and 9 resistors of value 2R. The actual valu e of R is not critical, but it is essential that all ‘R’ resistors have the same value and that all 2R resistors have double that value. In pra ctice, this is best achieved by using 23 resistors of 2R, and putting 7 pairs of them in parallel to create the 7 ‘R’ resistors. Resistors wi th a power rating of 0.25W and a tolerance of 1% are cheap in packs of 100, and that is what I recommend to use. I had an unused p ack of 2kOhm resistors. I think any value of R in the range 1kOhm-10kOmh will be fine. Smaller values will draw more current than th e Pico can provide and larger values result in poor performance because there is too little current to counteract parasitic capacitance and/or inductance at high frequencies.

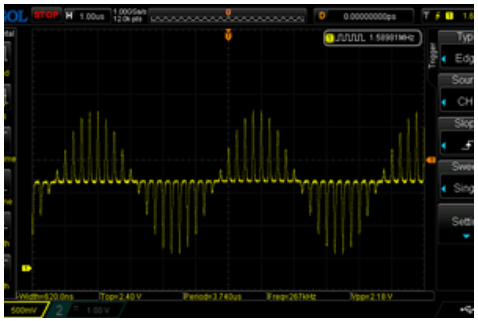
I numbered all the resistors, measured their values and put them in a spreadsheet. None differed by more than 1% from the nominal value, but by selecting I could reduce the effective spread from 1 percent to 1 per mille. For the ‘2R’ resistors I picked a value near th e mean of which there were at least 9, which happened to be the value 2000. Then I picked 7 pairs that were equally distant from the value 2000, for example 3 pairs of 1998+2002 Ohm and 4 pairs of 1997+2003 Ohm. In parallel, equal but opposite deviations cancel!

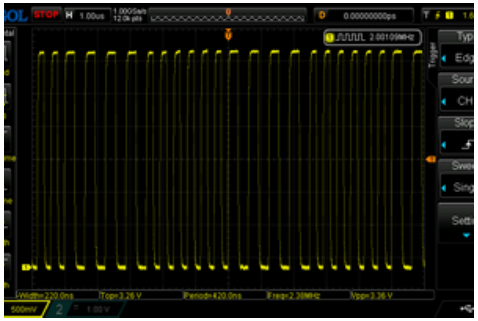
Solder the female headers to the prototype board such that the pico fits comfortably. Now solder the resistors according to the provid ed schematics and pictures. Note that the resistors are mounted vertically only to save space. Feel free to mount them horizontally if space allows. The board layout is done with KiCad. I did not make a PCB, but a PCB layout helps to solder effectively on the 5x7cm prototype board. Note that my build differs slightly from the KiCad design, I made some improvements to the design after soldering it up.

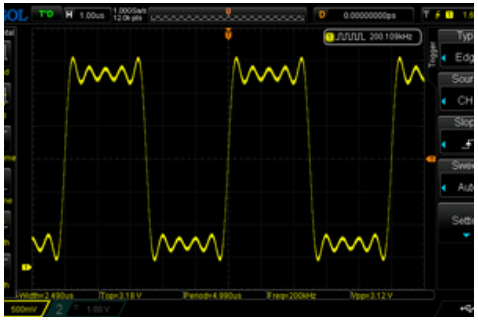
At the end I added to two male header pins: one for the signal and one for ground. Probe clips and crocodile clips attach well to them.

Step 2: Uploading the Software and Run









The pico can be programmed in C or micropython. Using micropython is much easier since you don't have to install the 'C-SDK'. Micr opython may be 100x slower than C, but here it doesn't matter. All the code does is making an array with a waveform and then instru cting the peripherals to stream that array to the output pins. The CPU is idle and free to perform other tasks.

Make sure your pico has the micropython UF2 file uploaded. I used version rp2-pico-20210205-unstable-v1.14-8-g1f800cac3.uf2. Ver sion 1.13, which was originally provided on release day, misses the 'uctypes' module. I used the Thonny IDE to upload the python scr ipt, I had never heard of that before but it works well, at least for small scripts like this.

The code as is will run though a few hundred example waves (see video at step 1). To modify this, scroll to line 155 and set up your o wn wave.

There are 6 basic pulse shapes: sine, pulse, gaussian, sinc, exponential and noise. The sine has no extra parameters, but the pulse has 3: risetime, uptime and falltime. Gaussian, sinc and exponential have one parameter, which determines their width, and noise ha s one parameter, which determines the pdf of the output values, from 1 (flat) to 8 or larger (nearly gaussian).

A wave will have a shape, but also amplitude and offset. The shape can be replicated within a period, which increases its frequency. That way, operations like summing or multiplying can be done with waves of different frequencies. A non-zero phase shift can also be set, but its effect is only noticeable when combining waves.

Waves can be combined by either summing, multiplying or applying a phase modulation with another wave. The resulting combined wave can then be added, multiplied or modulated with another basic or complex wave. The screenshots and video show just a few ex amples of how complex shapes can be made.

There is only one value for the frequency, even when combining waves. Operations are not performed on-the-fly by the CPU, but stor ed in a buffer and played by the DMA/PIO. The 'duplication' keyword however does allow for a shape to fill at double, tripe, quadruple etc values of the (base) frequency.

The buffer size (value of maxnsamp) determines how complex the shape can be made, and how accurate the frequency can be set. For simple waves, like single sine or wide pulses, 1024 samples may be sufficient. A buffer size of 65536 (64kB) is the practical maxi mum. Filling the buffer may take 20-60s at that size!

Step 3: Comments About the Code

```
# product: sine * exponential
wave1=wave()
wave1.amplitude=0.4
wave1.offset=0.3
wave1.phase=0.0
wave1.replicate=10
wave1.func=sine
wave1.pars=[]

wave2=wave()
wave2.amplitude=1.0
wave2.offset=0.0
wave2.phase=0.0
wave2.replicate=1
wave2.func=exponential
wave2.pars=[0.2]
wave1.mult=wave2
for width in (0.05,0.10,0.15,0.20,0.25):
    wave2.pars=[width]
    setupwave(wavbuf[ibuf],freq,wave1); ibuf=(ibuf+1)%2
```

I admit the code looks obscure: most of it is based on direct register access, and require studying the 637-page datasheet of the RP2040 chip to understand. But I'll try to explain the thought behind it.

The crucial peripheral is the so-called Direct Memory Access (DMA) module of the chip. The DMA can be instructed to perform block copies between memory and peripherals without requiring the attention of the CPU. Really, I had no clue this existed just 2 weeks go either! It is like having an assistant whom you tell to do shopping for you!

Anyway, this DMA is being told to transfer the contents of the array with the waveform to another peripheral (the PIO) which will put the values on the output pins. One complication is that this DMA needs to do this cyclically, and without interruption. For that, a second DMA channel is instructed to reconfigure and restart the first DMA channel as soon as it is done. This is called 'chaining'. So channel 0 does the transfer, and passes the stick on to channel 1. But channel 1 immediately tells channel 0 to restart. You might expect a delay in this swapping between the channels, but that delay is absorbed in buffers: the DMA transfers 32-bit words, while the PIO only 'eats' 8-bit bytes. So the PIO still has some snacks in its buffer while the DMA is losing 1 or 2 cycles to start over again. I am amazed by the engineers and scientists who came up with such intricate hardware!

The DMA cannot control the pins directly, but there is something better: the pico has 2 Programmable Input/Output (PIO) units, which have 4 processing units themselves, (called 'state machines'). They are really 8 tiny microcontrollers inside the microcontroller itself. Here, only 1 state machine is used, and it is programmed with a single command (would this qualify for a Guinness World record of smallest computer program?) The command is 'out(pins,8)' which instructs the state machine to pass 8 bits from its buffer to the output pins. Wrapping is implied, so the state machine just keeps doing this single command, every clock cycle. As the 8 bits are shifted to the pins, the buffer will request to be refilled by the DMA when 32 bits have been consumed.

And that's all. So the code consists of

- The configuration of the DMA
- The configuration and programming of the PIO
- The filling of the array with the waveform
- Starting up the DMA.

At that moment the waveform is produced and the CPU's of the pico are free for other tasks. There will be data traffic on the bus, but the pico has a highly parallel bus structure, and I expect no noticeable slow-down.

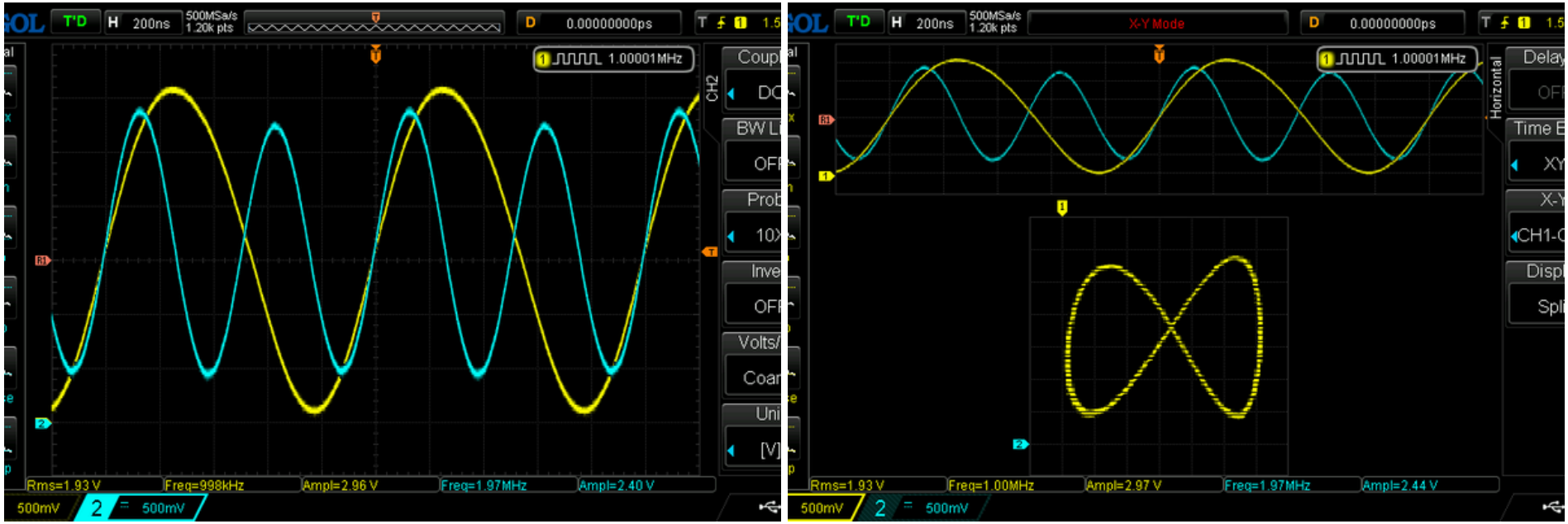
Step 4: Ideas for Improvement

I posted this instructable a bit prematurely to demonstrate the capabilities of this new microcontroller. I hope it will motivate some of y ou to get a board and try out its new features.

For this AWG project I have several ideas on how to improve it:

- It might very well be possible to have higher resolution (10 or 12 bit) and/or a second channel.
- The RP2040 can be overclocked to 250 MHz or more, resulting in 250Msps AWG
- Sweeps of frequency or other parameters can be implemented in the python code after setup.
- A real AWG needs a buffer and/or amplifier to reduce the output impedance and extend the voltage swing.
- A well-designed PCB and SMD resistors might reduce parasitic capacitance or inductance and improve the bandwidth.
- A screen and buttons, or a touch-screen could make it into a self-confined apparatus, like the commercial devices.

Step 5: Appendix on Data Throughput



Several other users have extended the AWG with a larger number of output bits, either for higher precision, a second channel, or bot h. When doing so, it appears the data throughput is not always able to hold up to the sampling speed. I have investigated this a bit an d tell here the current status of my findings.

The key quantity is the number of samples that can be stored in a 32-bit word. In the present project that was 4: every 32-bit word co ntains 4 samples of 8 bits each. Thus, the DMA only has to run at one quarter of the clock speed and it does so without any problem.

For a 10-bit DAC, 3 samples can be stored in a 32-bit word. I found the present setup results in a small hiccup at the restart of the D MA, which, however is resolved by setting fifo_join=PIO.JOIN_TX in the PIO decorator. This enlarges the FIFO that transmits to the P IO from 4 to 8 words, at the cost of the (here unused) FIFO that receives from the PIO.

For an 11-bit DAC, a 12-bit DAC or a 2-channel 8-bit DAC, a 32-bit word can only fit 2 samples. Here I would have expected that the DMA would be able to cope, but it does not. Apparently, in the present setup, a 32-bit word transfer is done at most every other clock

cycle. There may be a way to solve this, but at present I have no idea how. In any case, with the DMA just able to follow up with the PIO, there is a substantial delay when the DMA reconfigures. This is solved by slowing down the PIO by a factor two, but of course this results in the sampling frequency now being only half the clock speed.

For a 2-channel DAC beyond 8 bits, only one sample fits in a 32-bit word. Hiccups can be avoided by slowing down the PIO by a factor three.

Of course a given piece of hardware can be configured either for speed (up to 10 bits, with the remainder unused) or for precision/channels (e.g. 2x11 or 12 bits) running at one third of the sampling speed. Best would be to force the unused pins to zero (not done in the attached script).

An updated script is attached here, which allows to demonstrate and test these scenarios. It is set up for a 2-channel 11-bit DAC, running from pin 0 to pin 21, with the first channel going from pin 0 to pin 10 and the second channel from pin 11 to pin 21. The second channel however has inverted bit order: the MSB is on pin 11 and the LSB on pin 21. This was done to reduce cross-talk, but it has the additional advantage that the most significant bits are in the middle, which makes it possible to run it as a 2-channel 8-bit DAC with consecutive pins. The script now also allows to set the clock speed anywhere between 100 and 250 MHz: many have reported that overclocking by up to a factor two from the nominal 125MHz is OK. In addition, matching the clock frequency with the output frequency can result in a particularly stable output wave.