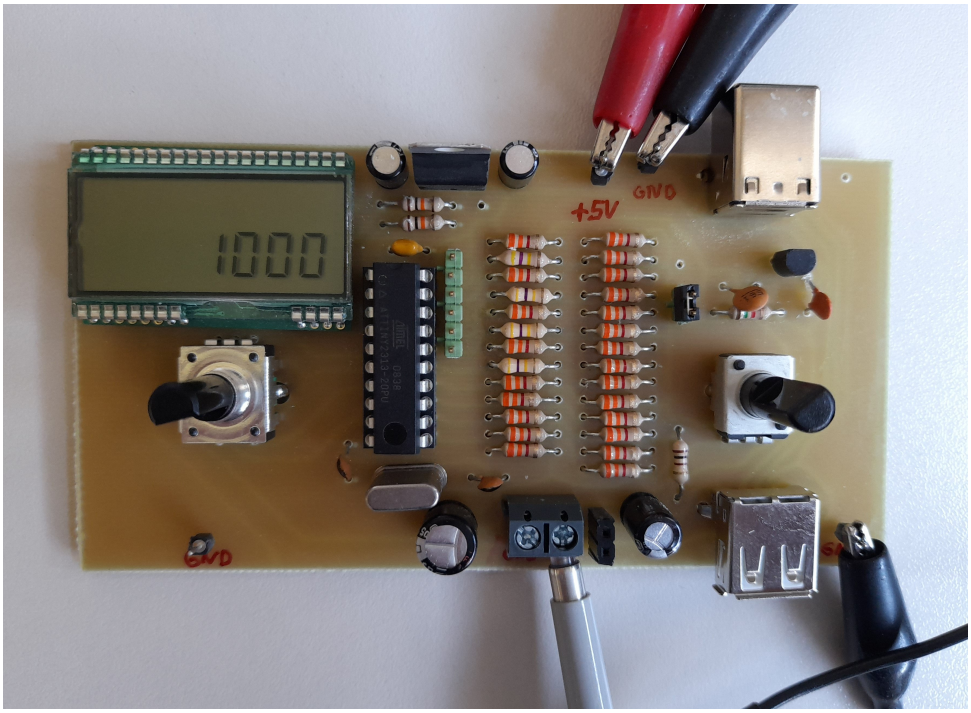
AVR function generator

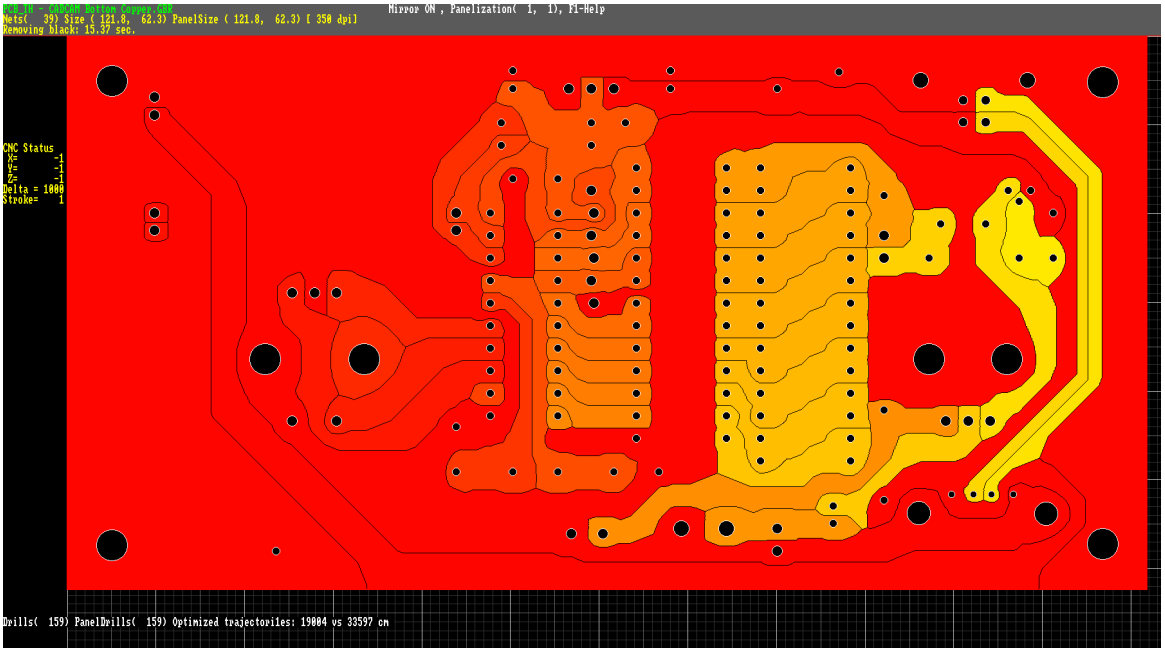
J. Arias



1 Introduction

This is by no means the first AVR DDS function generator around. Similar designs were found on the Net some years ago, but I decided to do it in my own way, mostly with the intention of building a practical function generator for my bench using mainly scrap components I had in stock, like an LCD numerical display with an I2C interface and an ATTINY2313 microcontroller that a decade ago was used, and probably abused, by students in the lab. In addition to these parts, the other very important block was a digital to analog converter (DAC), that I wanted to build using only resistors. Also, in this design the output signal is low-pass filtered in order to remove sampling artifacts (signal steps) This filter was built using just a single PNP transistor and a few passive components. It was very simple but demanded a supply voltage higher than that of the microcontroller, so, a voltage regulator is also included in order to reduce the MCU supply that now is only 3.3V. My idea here was to use an USB port as a power supply for the generator, thus, providing 5V to the filter amplifier and the LCD display while the MCU runs at 3.3V.

Some components had to be ordered. These were a potentiometer for the control of the output amplitude and a rotary encoder with switch for the selection of the signal frequency and waveform.

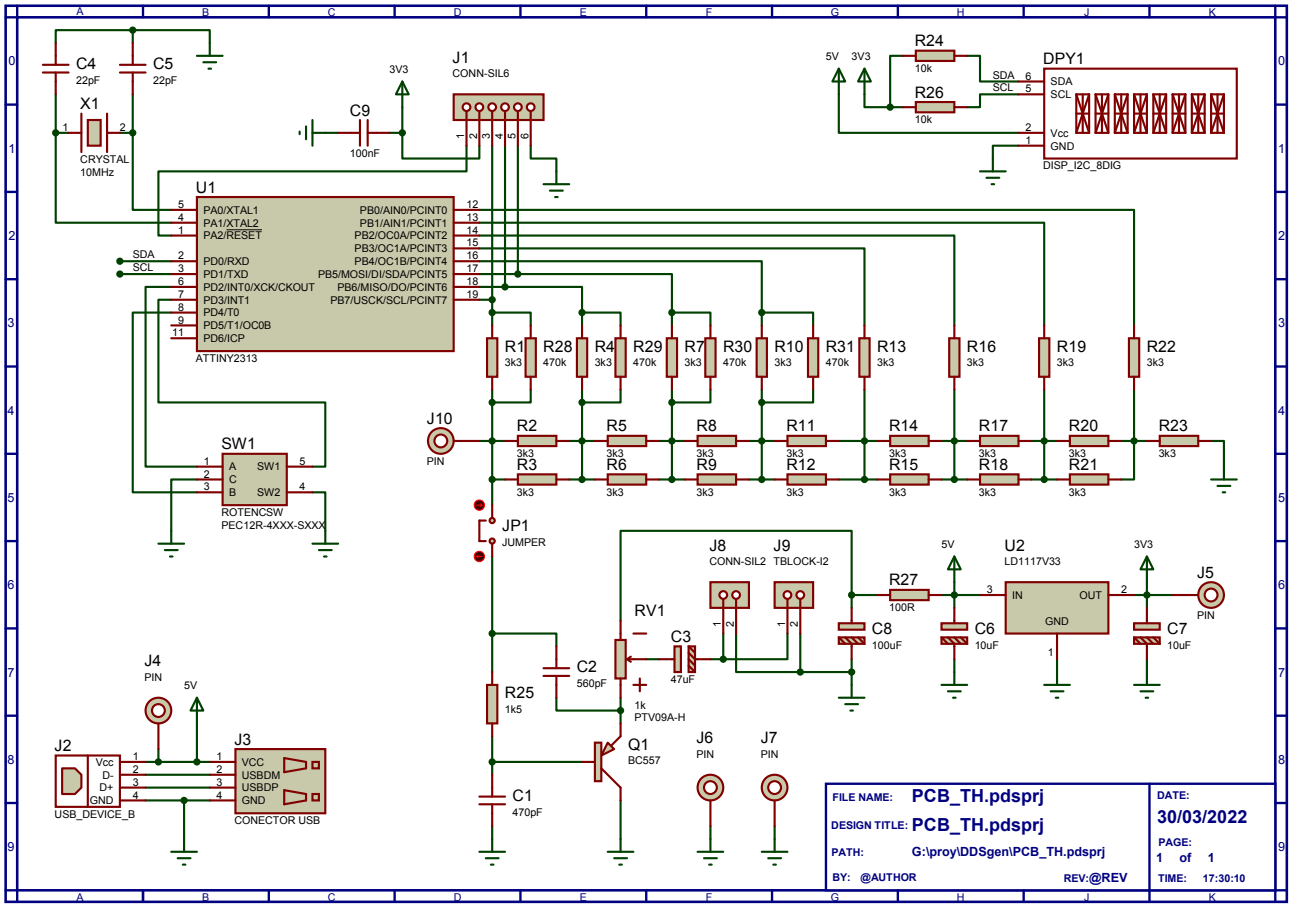


Only one prototype was going to be built and it made no sense to order a batch of PCBs for it. So, I managed to do all the routing of the prototype in a single side PCB (without any bridges!) and made a PCB for it using a CNC drilling and milling machine (see the screenshot of the milling program with the layout).

But first, let me present the circuit and its critical building block: the DAC.

2 System design and description

2.1 R-2R DAC, tolerances and mismatch



The schematic of the generator circuit is shown above, where the DAC is built with resistors R1 to R23 and

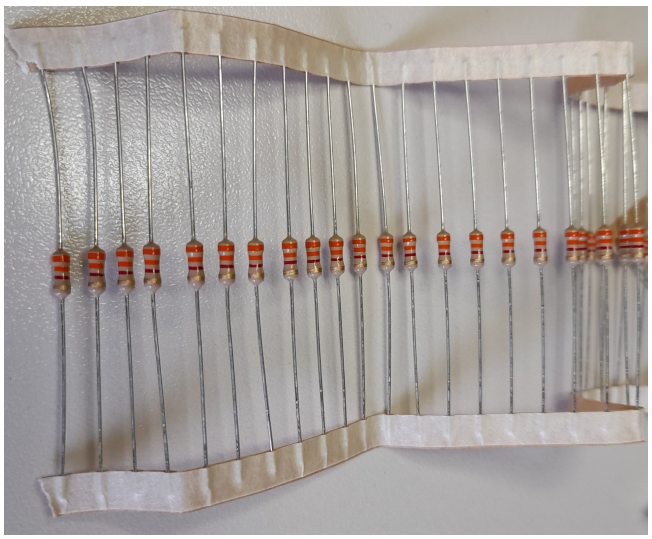
also resistors R28 to R31. These last 4 resistors are included for fine tuning the DAC linearity and could be removed resulting in a classical R-2R DAC built with identical resistors. In this circuit R is the equivalent of two parallel-connected resistors, or 1650Ω, while 2R is 3300Ω.

I was very concerned about the linearity of such a DAC. So much, in fact, that I wrote a simulation program to analyze the non-linearity of lots of DACs built with resistors exhibiting variations. A resistor with a 5% tolerance can have a value ranging from 0.95 to 1.05 times its nominal value, and a DAC built using these resistors has a very poor linearity. Many DACs simulated with a 5% variation in their resistors had a maximum differential non-linearity much higher than one LSB, meaning they were non-monotonic, and, in summary, a heap of junk.

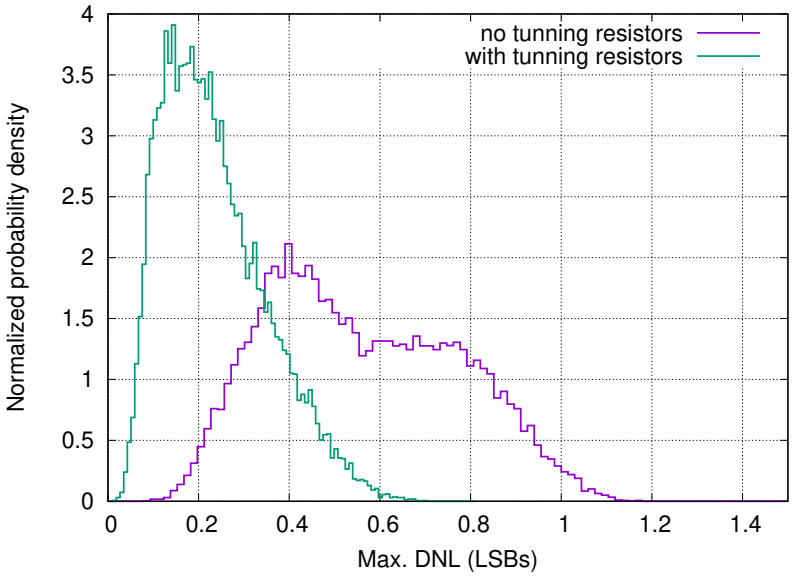
Just to clarify things let me present some definitions:

- 1 LSB is the change in the output voltage of an ideal DAC when the input code is increased by one, in our case it is 3.3V/256, or 12.89mV.
- The differential non-linearity is the error in the voltage step when the input code is increased by one. In an ideal DAC the increment is 1 LSB, so the DNL is the absolute value of the output step minus one LSB.
- A DAC is non-monotonic if there are some cases when, while increasing the input code, the output voltage decreases. This implies a DNL value higher than 1 LSB, and the most problematic code is usually located midrange. In our case this happens when the input code changes from 127 to 128. A non-monotonic DAC is usually considered unacceptable.

So, the discrete resistor DAC was unusable, or so I was thinking until I realized the tolerance of resistors is one thing and the matching between resistors something completely different. In fact, no accuracy is required at all in the DAC resistors, it doesn't matter if the resistor value is higher or lower than its nominal value, what is really important is to have all the resistors as close to each other as possible. Well, the accuracy of resistors is stated with a color band, but the matching between resistors isn't specified in any place, so I took a multimeter and a batch of brand new resistors with a 5% accuracy and I measured 23 of these resistors:



The value measured was 3.25kΩ for 16 resistors, 3.26kΩ for 4 resistors, and 3.24kΩ for 3 resistors. I would like to have a better multimeter, but anyway, the variation was less than 10Ω for a 3250Ω resistor or only a 0.3%. On the other hand the measured average was an 1.5% off from its nominal value (or maybe, the multimeter was asking for calibration).



With these results, with only a 0.3% mismatch, the DAC is expected to be much more linear than before. In the former graph the magenta line shows an histogram for 20000 DACs simulated with these kind of resistors, and only a few have DNLs over 1 LSB.

Yet, There is still another source of error due to the resistance of the microcontroller pins. The ATTINY2313 datasheet include some nice curves showing the variation of pin voltages with output current, and from these graphs an estimate of the output resistance can be extracted. In our case the pin resistance is about 21Ω when the output is low, and 24Ω when the pin is high. These values were higher than the expected mismatch and they can affect the linearity of the DAC, so, they were included in simulations. In order to reduce their effect, some 470kΩ resistors (R28 to R31) were placed in parallel with the 2R branches of the DAC with the intention of reducing the resistor value from 3300Ω to 3277Ω. This was done only in the four most significant bits of the DAC because the error due to pin resistance is negligible in the lower bits. With these four fine tuning resistors the linearity of the DAC is clearly improved, as we can see in the green curve of the preceding plot.

In summary, the key recipe for the DAC design is to pick all its resistors up from the same batch. The mixing of resistors from different manufacturers is really a very bad idea.

2.2 The filter

Sampled signals need a low-pass reconstruction filter with a cut-off frequency below the Nyquist frequency, or half the sampling rate. In the former schematic a second-order Sallen-Key filter is built around an emitter-follower amplification stage (Q1 , RV1, R25, C1, and C2). An input resistor seems to be missing, but it is in fact the equivalent output resistance of the DAC, or 1650Ω. I started the design with a Butterworth filter with C2 being double the value of C1. It was fine for sinewaves, but it showed some unwanted overshoots for square waves, so, I departed from the Butterworth standard and lowered the Q of the resonator by means of making C2 more close to C1 and the overshoots were gone.

The transistor Q1 provides a lower output impedance than the DAC and the potentiometer RV1 allows the reduction of the waveform amplitude. The output waveforms are AC coupled through C3.

2.3 The direct digital synthesis (DDS)

In order to generate a sampled sinewave it is a convenient way to follow the DDS approach where, first, a phase accumulator variable holds the phase of the current sample. This value can be used (or at least its more significant bits) as an index for a table storing the values of a single sine cycle. Then, a constant value is added to the phase variable periodically, with the overflow of integer variables simulating the equivalent phase angle

change from 360º to 0º. The phase increment is proportional to the ratio between the generated frequency and the sampling rate:

$$\frac{f_{sine}}{f_{sampling}} = \frac{\Delta\varphi}{2^N}$$

Where N is the number of bits of the phase variable. In the firmware of the AVR MCU three registers were used to store the phase, so N is 24 bits and the resulting minimum frequency step ($\Delta\varphi$=1) is 0.059Hz for a sampling rate of 1MS/s. The code for the generation of the sinewave is:

```
        ldi     r31,hi8(sine)
1:      add     r21,r20         ; phase+=frequency
        adc     r23,r22
        adc     r30,r24
        nop                     ; delay adjust
        lpm                     ; r0=progmem[r31:r30] (sine table)
        out     0x18,r0         ; PORTB=r0
        rjmp    1b
```

Where “sine” is a program memory address with its lower 8 bits in zero (multiple of 256). This code is a never ending loop that takes 10 cycles to execute (3 cycles for LPM, 2 cycles for RJMP, and 1 cycle for any other instruction). The 8 MSB bits of the phase (R30) are used to address the sine data stored in the program memory by means of the LPM instruction. With a clock frequency of 10MHz the resulting sampling rate is 1MHz, and therefore signals up to 500kHz could be generated, but only if a very sharp reconstruction filter is included after the DAC. From a more practical point of view it is better to reduce the maximum signal frequency to 1/4 of the sampling rate and to relax a little the filter requirements.

Another interesting thing about the previous code is the impossibility of doing any other task in the MCU when the sinewave is being generated. Unless this code is run with interrupts enabled and any user action stops the code and jumps to an interrupt service routine. This is precisely what the firmware does: the encoder rotation and pushing results in the activation of the external interrupt pins of the ATTINY2313. This allows the change of the frequency being generated and its waveform.

And BTW, some simple signals can also be generated in a similar fashion without the need to use any lookup table. Take for instance the generation of a sawtooth waveform, where the 8 MSB of the phase variable are routed to the DAC directly:

```
1:      add     r21,r20         ; phase+=frequency
        adc     r23,r22
        adc     r30,r24
        nop                     ; delay adjust
        nop
        nop
        nop
        out     0x18,r30        ; PORTB=phase[23:16]
        rjmp    1b
```

Or the generation of a square wave, where the MSB of the phase is used to select between two possible output values:

```
1:      add     r21,r20         ; phase+=frequency
```

```
        adc     r23,r22
        adc     r30,r24
        ldi     r16,0xff
        sbrs    r30,7           ; skip if phase[23]==1
        ldi     r16,0
        nop
        out     0x18,r16        ; PORTB=r16
        rjmp    1b
```

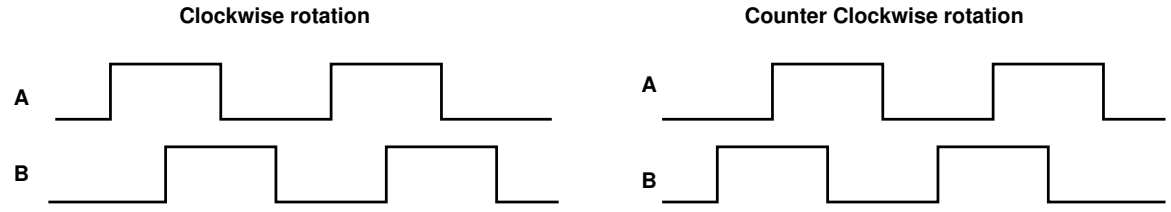
Also, a symmetric triangle wave can be generated by complementing the LSBs of the phase when its MSB is one:

```
1:      add     r21,r20         ; phase+=frequency
        adc     r23,r22
        adc     r30,r24
        mov     r16,r30         ; r16=phase[23:16]
        sbrc    r16,7           ; skip if phase[23]==0
        com     r16             ; r16=~r16
        add     r16,r16         ; r16*=2
        out     0x18,r16        ; PORTB=r16
        rjmp    1b
```

2.4 Human interface

For human interface a seven digit LCD display is used along with a rotary encoder that also includes a push switch. The main idea here is to use the encoder to increment or decrement the frequency of the generated signal and the switch to change the waveform. An analog control is also provided for the waveform amplitude by means of a potentiometer.

The rotary encoder is just two switches that generate pulses in quadrature :



The switch A is connected to the INT0 pin, while the B switch is connected to GPIO pin PD4. Therefore, a falling edge at INT0 will cause an interrupt and the interrupt service routine (ISR) will also check the PD4 pin in order to decide what to do with the frequency (decrement if PD4 is zero or increment otherwise). Actually, the ISR is a bit more complex because these are mechanical switches and they could bounce resulting in dirty edges and unwanted interrupts that have to be filtered out by software. The same applies to the INT1 pin that is connected to the push switch of the encoder.

After any change the ISRs can make the code has to jump to one of the four possible signal generation loops, but before this happens the stack pointer must be reset in order to avoid an stack underflow due to interrupt nesting, and the interrupts have to be enabled again. Also, the new frequency and waveform has to be displayed on the screen, and this requires the activation of the required segments of the digits in a cache buffer in the MCU and the sending of the final buffer contents to the display via an I2C frame. The ATTINY2313 lacks a hardware I2C peripheral, so its functions are implemented via bitbanging.

2.5 USB power

The generator is powered from an USB connector wich provides 5V. An additional linear regulator lowers the voltage to 3.3V, that is the supply of the MCU. The filter amplifier and the LCD display are powered by the 5V directly. Well, the filter amplifier also includes an RC (R27, C8) filter in its supply to reduce the noise of the USB power.

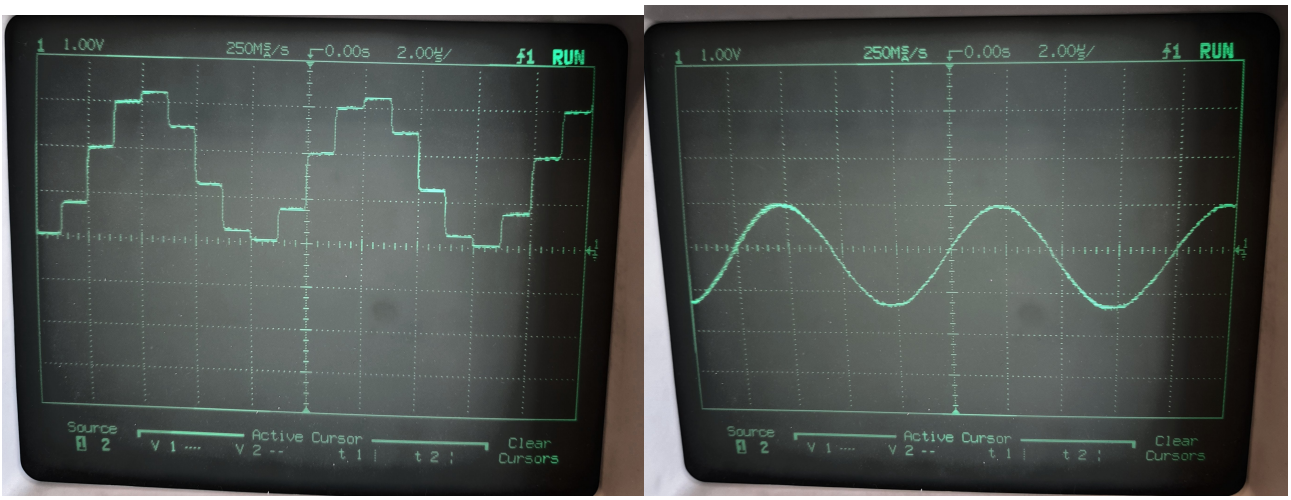
The I2C lines, SDA and SCL, have their respective pull-up resistors (R24 and R26) tied to 3.3V, thus avoiding voltages higher than the supply in the MCU.

The measured current consumption was only 12.6mA, and almost half of this current is consumed as a quiescent current in the voltage regulator (The actual regulator was a LD1086, an obsolete part in stock with the same pinout as the one shown in the schematic that, BTW, also has a quite large quiescent current). The current consumtion is much less than the minimum 100mA provided by any USB port.

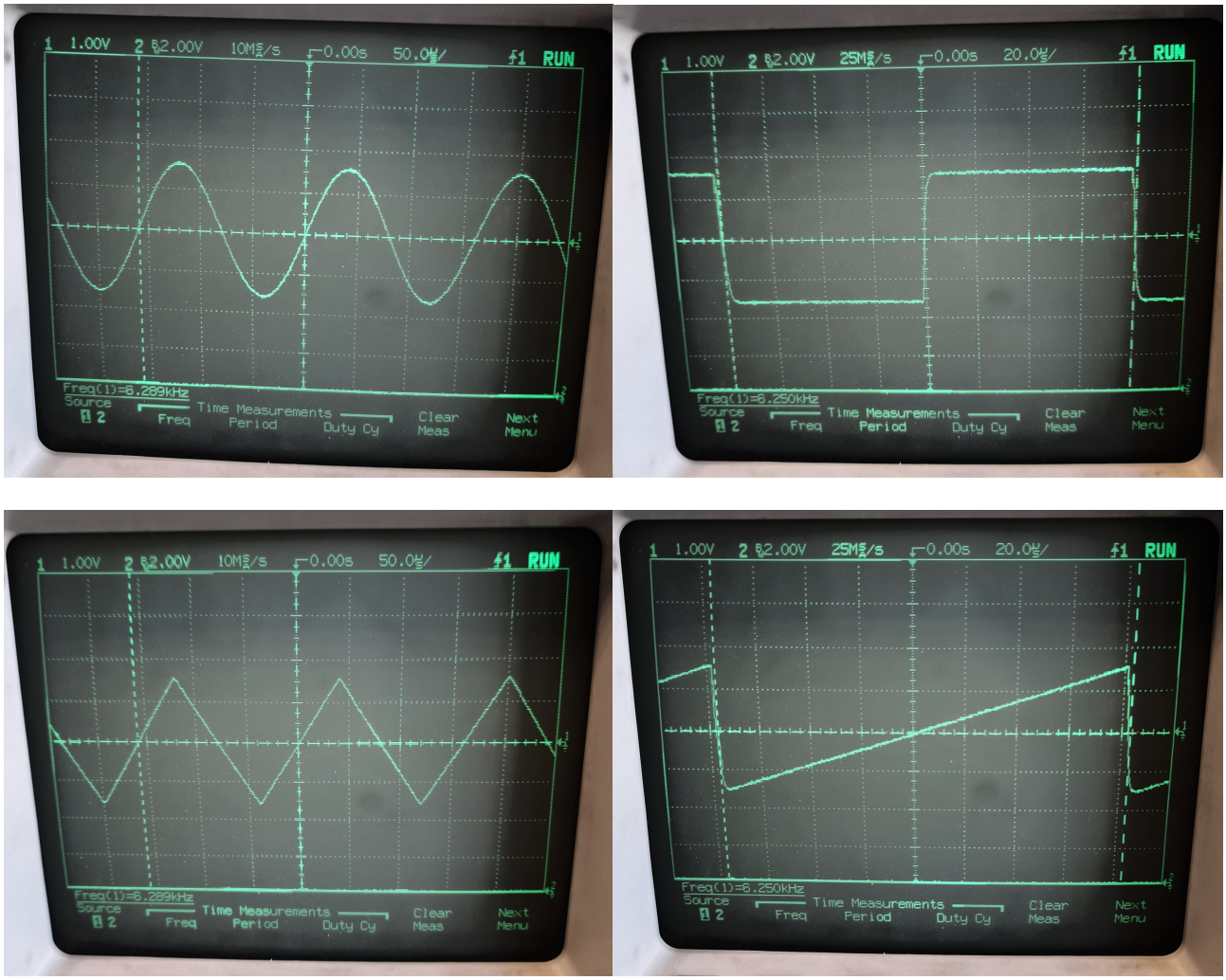
The prototype includes an USB-B connector for power input, but it also includes an USB-A that is connected as a pass-through. This allows the connection of other USB devices to a port that, otherwise, would be only used for powering the generator. The power consumption of the generator is so small that almost all the USB power is still available for the device connected to the USB-A port.

3 Results

The waveform generation worked flawlessly at the first try on the real hardware. The human interface, on the other hand, needed some debugging starting with the fact that the display only has 7 digits instead of the 8 I was supposing it had, and ending working a convenient way to modify the frequency steps in order to be able to cover all the range from 1Hz to 250kHz with a reasonably low number of turns in the encoder.



But lets start presenting some waveforms captured in a scope screen. Here on the left, the output filter was disconnected (JP1 off) and the signal was obtained directly from the DAC. This signal shows very clearly the steps of a 125kHz sampled sinewave (8 samples per cycle). On the right snapshot the same signal is shown after being passed through the reconstruction filter and what remains is just a sine wave without steps and with a little attenuation (the cutoff frequency of the filter is about 200kHz).



Now, in the above snapshots , the four possible waveforms are shown. The signal frequency was way lower (6.25kHz) in order to have a near perfect waveforms for the non-sinewaves. The square wave shows on its edges the step response of the reconstruction filter with no overshoots.