

Tutorial: VT-5 On-the-Fly DDS Meets Parallel I/O

Author: Greg Kovacs (with his own crappy code and debug with Grok)

Date: 10/5/26

Revision: 0.2

Introduction

Well, I know full well that when I share a tutorial, there are folks who will chime in to say stuff like, “you need to use the `hamsterNoggin()` function because it will save 0.0001% CPU load.” I always think – this is a tutorial with simple goals and a time-constrained authorship cycle – but I also appreciate the input and learn.

Recently, on this thread of “On-the-Fly DDS,” someone pointed out the `sinf()` function (single precision float) versus `sin()` (double precision). That got me thinking and reading. It turns out that there are nearly as many ways to compute the sine of an argument in c++ on an ARM as there are words for ice in Inuit. The thing is, some take variable amounts of time to return, based on the argument (but are really precise) and others take constant time to return, but bring higher noise. Who would have known (not me).

So this whole idea, which I’ve been playing with since the Teensy 3.6 is that if the CPU is fast enough, one can compute the sinewave samples on-the-fly and keep up with the sample rate. Cool. Floating point – rounded to the ADC resolution, and no LUT (look-up-table) taking up memory and, by its size, introducing spurs in the synthesized waveform’s spectrum. So I dug in more.

Of course – and I’ve heard this all before – it is bad to do code-timed loops for waveform synthesis. But, if there is no jitter, one simply measures the loop cycle time and obtains an effective sample rate that is stable from chip to chip to chip. That’s actually ok. You can see that it yields pretty nice signals on a spectrum analyzer.

Then it comes down to “how fast can it go without compromising signal quality?”

Exploration of Options

First, it is important to understand the timing and jitter questions when computing sinewave values on-the-fly. The simple jitter test code in Appendix 1 (which uses the `arm.math.h` library) is used to measure the jitter of a loop-timed DDS, with interrupts disabled on the Teensy 4.1 (be careful – can hang-up some USB ports and hubs and always requires pressing the manual program button on the Teensy 4.1) and allows a clear look at the effects of the different sine computation methods on timing – but not yet on ultimate signal quality (noting that they return vastly different sine precisions). The sinewave samples are not used, but the time to complete one DDS cycle is presented on a bit-banged pin to be measured with an oscilloscope. Please note that the “sampleRate” in the code is just a placeholder – the code timing determines it.

The `arm_sin_f32()` and `arm_sin_q31()` functions are designed to return a result in a fixed time (“strict execution time”) regardless of their (valid) argument. The trade-off is lower precision because they use table look-ups with interpolation rather than `sinf()` and `sin()`, which use argument reduction and polynomial approximations with branching logic. A good reference can be found here:

https://arm-software.github.io/CMSIS-DSP/main/group_sin.html

The results are presented here in “hindsight” order, starting with the fastest and lowest jitter arm library sine functions, and continues through and higher-jitter native floating-point sine functions, in Figures 1 through 4, respectively. All measurements were done at the default Teensy 4.1 clock of 600MHz, and the code is in Appendix 1.

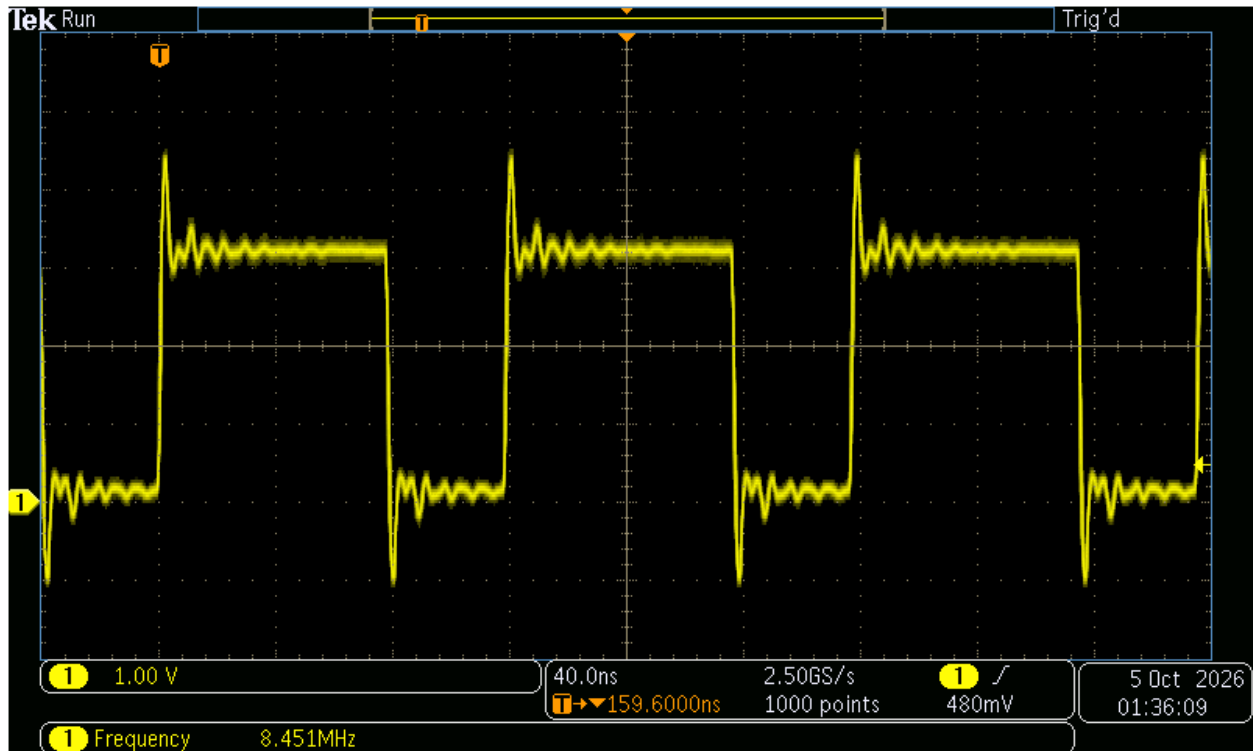


Figure 1: Test of the `arm_sin_q31()` function, showing no visible jitter and a loop-determined sample rate of 8.45 MSPS. Test done with Tektronix MDO3054 oscilloscope with 3.00s persistence, using TPP0500B, 500MHz probe, no damping resistors (hence ringing).

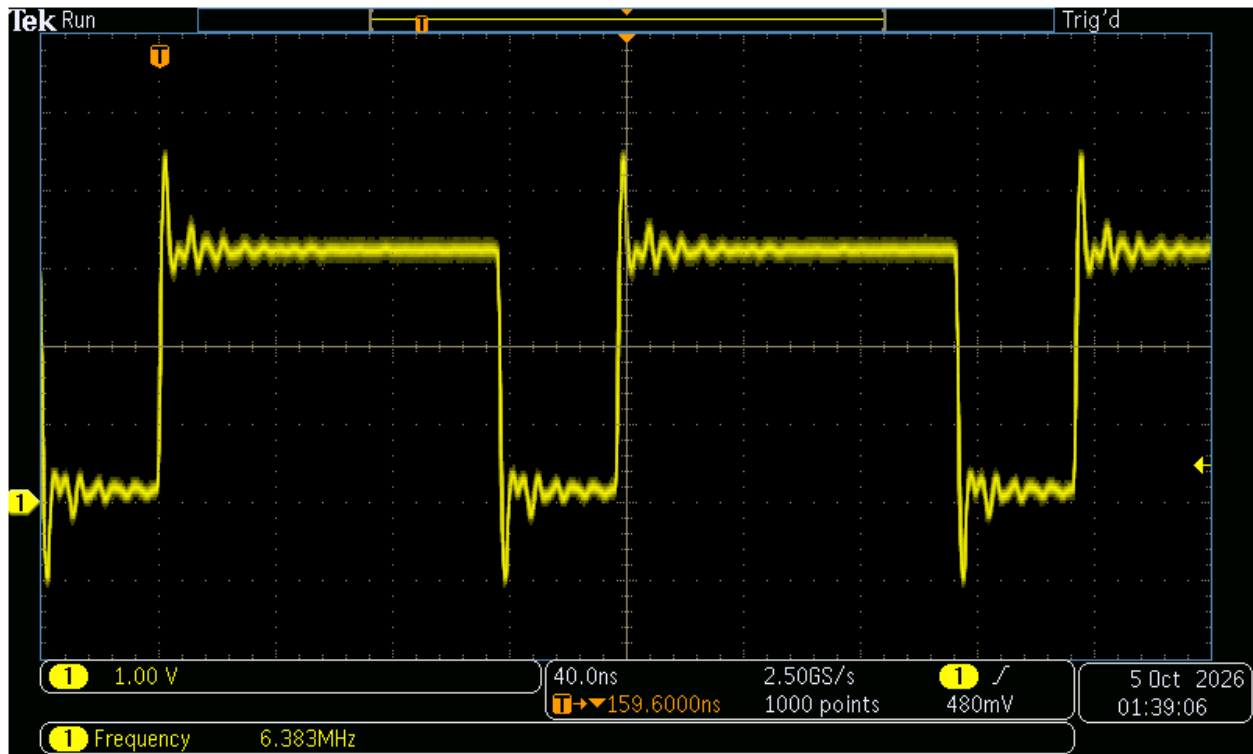


Figure 2: Test of the `arm_sin_f32()` function, showing no visible jitter and a loop-determined sample rate of 6.38MHz. Test conditions identical to those in Figure 1.

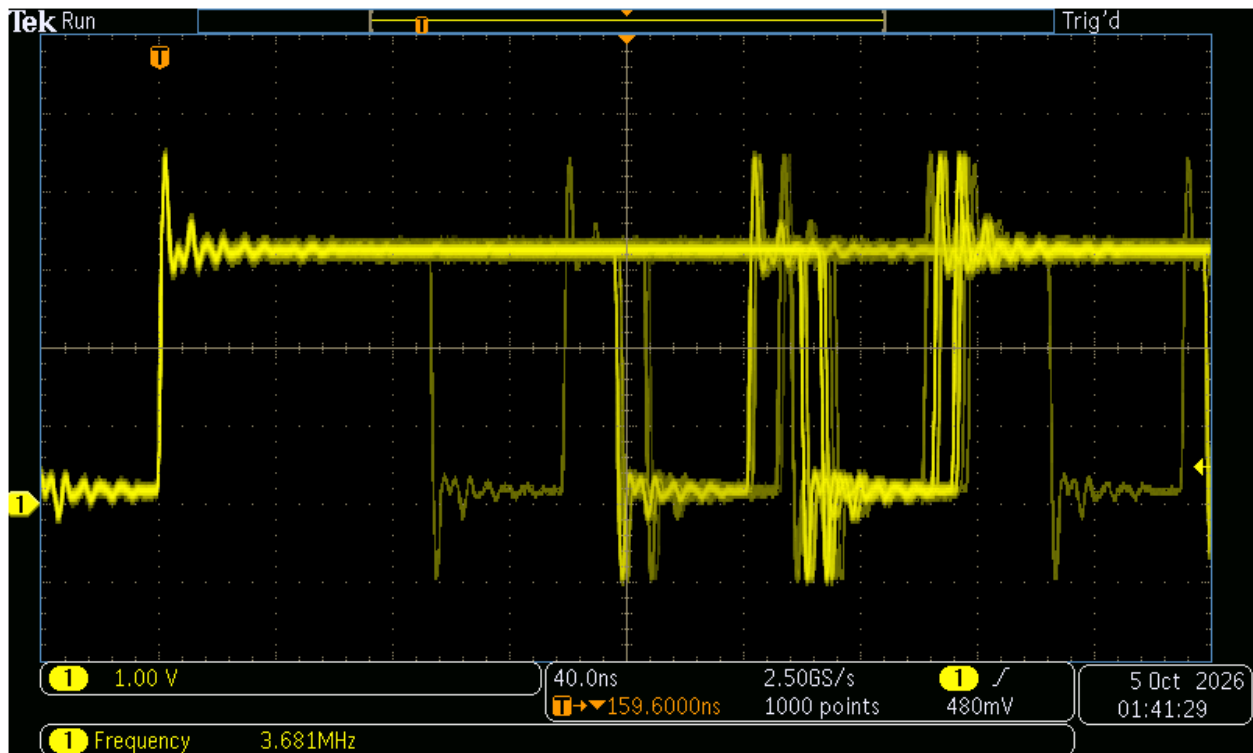


Figure 3: Test of the `sinf()` function, showing considerable jitter with occasional very large jitter excursions captured by the scope's persistence mode, and a loop-determined sample rate of ≈ 3.7 MSPS. Note that the `sinf()` return time is measured as rising-edge to rising-edge, so sometimes it is faster than average, and sometimes much slower.

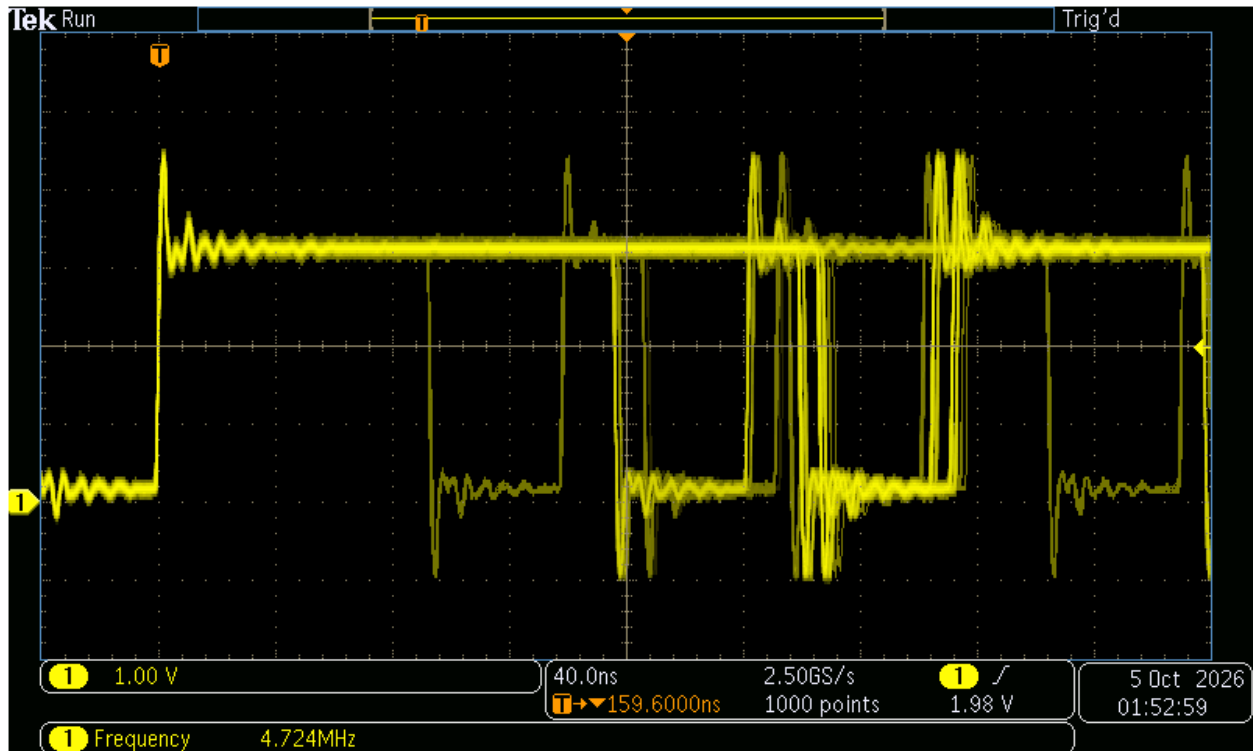


Figure 4: Test of the double-precision `sin()` function – exactly the same as `sinf()` - unless the argument is explicitly cast as a double (Figure 5). This appears to be the result of compiler optimization. Here again, considerable jitter is observed and a loop-determined sample rate of ≈ 3.7 MSPS.

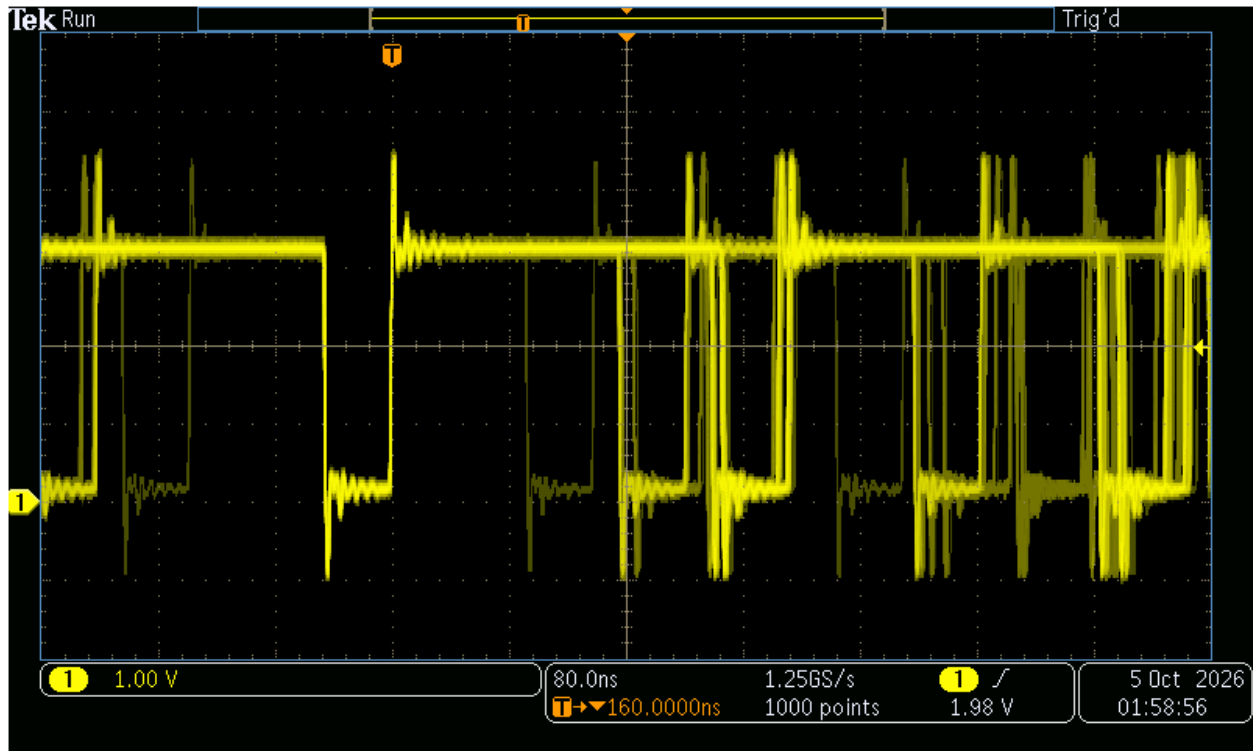


Figure 5: Test of the double-precision `sin()` function with the argument explicitly cast as a double, showing (on a slower sweep speed), considerable jitter an a sample rate of ≈ 2.3 MSPS.

So, we can see that the timing, if it matters, is fastest and argument-agnostic for the arm versions, and has significant jitter for the native `sinf()` and `sin()` functions. Having said that, if one is doing interrupt-driven synthesis (sadly limited to 1 MSPS using `intervalTimer()` due to interrupt servicing delays, etc.), the jitter in sine computation probably doesn't matter at all, so one can focus on accuracy. However, we can see that the accuracy – which could affect the purity of the synthesized signal, might be affected. Table 1 below, including peak error estimates obtained via a well-specified prompt to Grok, shows the relative errors in dBFS (dB full-scale). Bottom line is they are all good enough for realistic 16-bit DAC DDS.

Method	Format	What limits the error	Peak absolute error	Peak error, dBFS	RMS error (one cycle)
arm_sin_f32	float32, radians	512-point linear interpolation	1.89×10^{-6}	-94.5 dBFS	$\sim 9.7 \times 10^{-6}$ (-100.3 dBFS)
arm_sin_q31	Q31, input scaled so $1.0 \rightarrow 2\pi$	same table, plus a 2^{-31} LSB	1.88×10^{-6}	-94.5 dBFS	$\sim 9.7 \times 10^{-6}$ (-100.3 dBFS)
sinf	float32, radians	~ 1 ULP of the float32 result	$\sim 1.2 \times 10^{-7}$	-138.5 dBFS	about the same order
sin	float64, radians	~ 1 ULP of the float64 result	$\sim 2.2 \times 10^{-16}$	-313 dBFS	about the same order

Table 1: Grok-output comparison of the different sine computation functions being compared here. Note that `sin()` with a float argument will, on the Teensy 4.1, produce the same result and performance as `sinf()`.

Sine Function	Loop Frequency (MHz)	Execution Time (s)	Jitter	Peak Error (dBFS)
arm_sin_q31()	8.45	118E-9	not measurable	-94.5
arm_sin_f32()	6.38	157E-9	not measurable	-94.5
sinf()	3.7	270E-9	considerable	-138.5
sin()	3.7	270E-9	considerable	-138.5
sin(double)	2.3	435E-9	considerable	-313

Table 2: Summary of time-domain and amplitude-domain performance estimates.

One might well ask, why not use a 64-bit phase DDS accumulator, or a floating point accumulator? The 64-bit case would bring better frequency resolution, but with 32 bits we already get 233 μHz resolution at 1 MSPS (64 bits would yield 5.4 pHz, but that's not needed here). However, if we go with the fastest sine computation method, `arm_sin_q31()`, it uses a 512-entry lookup table with interpolation based on the top nine of the top 32 bits of 64. Bottom line is that this would not lower the spur levels.

For the floating-point case, the float mantissa is 23 bits, so the lower bits get dropped, and a double, while it can hold 32 bits completely, but is considerably slower. Not worth it here.

All of this stuff does not yet consider “how fast” the DAC receiving the data can go. If you are using a fast parallel DAC, as shown below, port I/O is readily possible up to 16 bits, so the speeds above are almost what you could get. Of course, “fast” here means that the DAC can settle at or less than the sample period, but for realistic systems, the maximum “clean” output signal with a strong reconstruction filter is perhaps 40% of the Nyquist frequency. If one considers a low-ripple filter such as Butterworth and a maximum order of 8th, to hit a tone that is clean relative to the quantization noise floor of $6.02N + 1.76 \text{ dB}$, Table 3 shows the approximate % of Nyquist that is practical. Of course, you can push it with other filter types, such as Cauer, but then one has to deal with ripple in the pass- and stop-bands, often requiring digital self-calibration.

DAC bits	Quantization Floor (dB)	Highest Clean Tone (% of Nyquist)
8	50	67%
10	62	61%
12	74	54%
14	86	48%
16	98	43%
24	146	26%

Table 3: Approximate fraction (%) of Nyquist frequency that can be cleanly synthesized versus DAC resolution. From this, one can see that for realizable filters and a constant sample rate, higher resolution (and ENOB) DACs yield lower maximum clean sinewave frequencies.

If you are using an SPI DAC for synthesis, you are limited by the Teensy 4.1 hardware. If not “overclocking” (oh, dear God no!) it is 30MHz SPI clock, which, with overhead of chip-select assertion (digitalWriteFast(), of course) and such, gives an update rate of about 828 kSPS for 16-bit, single-lane SPI, as seen below in Figure 6.



Figure 6: Testing “On-the-Fly” DDS with hardware SPI on Teensy 4.1 – this is not limited appreciably by the way sine() is computed, by the way. With the constant-time arm_sin_f32() function it edges up to 1MSPS with 30MHz SPI clock.

Of course, one can overclock the hardware SPI, which tops out around 40MHz SPI clock, so maybe a bit above 1MSPS on a good day. Ok. Cool. So we have a rough hardware limit for serial DACs on the Teensy 4.1 (noting that many single-lane DACs these days support SPI clock speeds of 50MHz or higher). The way to break that is more “lanes” of data per clock, perhaps QSPI or – OMG, old-school – parallel I/O!

The same code that yielded Figure 6 above (in Appendix 1), with the SPI transactions commended out, hits 3.333MSPS, which assumes a zero-time parallel port push and using **arm_sin_f32()** to do the computation without jitter. Still, just checking it out. Please see Figure 7, which is no SPI, but still the code-timed loop setting the DDS sample rate. This roughly represents “infinitely fast” SPI.



Figure 7: Code-timed (I can hear the CS-bro’s screaming) DDS doing on-the-fly floating-point sine calculations but without SPI transactions, roughly simulating parallel I/O maximum sample rate of about 3.333 MSPS. Some bro-grammers will also call for DMA, which does not work for DDS (except for pushing pre-computed blocks of data), but works like a champ for fixed-table synthesis (coming soon to a tutorial near you!).

If we use fast parallel port I/O, which I and a few others have decoded for the Teensy 4.1, there is a full sixteen contiguous (!) bits exposed on pins, which happen also to be the high 16-bits of a 32-bit word that can be pushed the port. The pins are scrambled like crazy, but physical wiring can fix that – a PCB! So we’re in business. I tested the port with LUT-based DDS (more on that in the future) and could hit well over 60 MSPS update rate. Figure 8 below shows the painfully mapped parallel I/O pins for Teensy 4.0 and 4.1.

Teensy 4.0 and 4.1 Parallel I/O Mapping Guide

CONTIGUOUS but scrambled 12-bit port I/O on Teensy 4.0

G. Kovacs
7/20/20

Green boxes represent relevant bits of a 16-bit word one wants to get into or out of a Teensy 4.X as a parallel word, mapped to digital output pins shown at left.
"X" marks represent the bits of the internal 32-bit word that are output to each digital output pin shown at left.
Shifts in code "move the green box to the X" representing shifting a bit of 16-bit data to the right place for a write. It is the opposite shift direction for a read.

Note that this assumes that for FAST reads or writes, the data is bit-reordered as needed either before or after the port I/O action, respectively.

Port 6 Bit->		uint16_t data																																uint16_t			
Bit Order	Port Pin	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	32-bit Hex Mask	Bitmask	Bitshifts	
LSB B0	D0																																	0x00000008	0x0010 >>1		
	D1																																	0x00000004	0x0020 >>3		
	D27 back																																				
	D26 back																																				
	D25 back																																				
MSB B11	D24 back																																				
	D23																																		0x02000000	0x0800 <<10	
	D22																																		0x01000000	0x4000 <<10	
	D21																																		0x08000000	0x2000 <<14	
	D20																																		0x04000000	0x1000 <<14	
	D19																																		0x00010000	0x0800 <<5	
	D18																																		0x00020000	0x0400 <<7	
	D17																																		0x00400000	0x0200 <<13	
	D16																																		0x08000000	0x0100 <<15	
	D15																																		0x00080000	0x0080 <<12	
B2	D14																																	0x00040000	0x0040 <<12		

For reads, "AND" with 0x0FCF000C to mask out unused bits.

Teensy 4.1 16-bit port COMPATIBLE WITH TEENSY 4.0 (shares same MSB and goes down from there) for PCB compatibility.

Note that this assumes that for reads or writes, the data is bit-reordered as needed either before or after the port I/O action, respectively.

Shifts in code "move the green box to the X" representing shifting a bit of 16-bit data to the right place for a write. It is the opposite shift direction for a read.

Port 6 Bit->										uint16_t data																											
Bit Order	Port Pin	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	32-bit Hex Mask	uint16_t	Bitmask	Bitshifts
B4	D0																														X			0x00000008	0x0010	>>1	
B5	D1																															X		0x00000004	0x0020	>>3	
LSB B0	D27	X																																0x80000000	0x0001	<<31	
B1	D26		X																															0x40000000	0x0002	<<29	
B2	D25																				X													0x00001000	0x0004	<<11	
B3	D24																				X													0x00001000	0x0008	<<9	
MSB B15	D23								X													X												0x20000000	0x8000	<<10	
B14	D22								X																									0x01000000	0x4000	<<10	
B13	D21					X																												0x08000000	0x2000	<<14	
B12	D20					X																												0x40000000	0x1000	<<14	
B11	D19																X																	0x00010000	0x0800	<<5	
B10	D18															X																		0x00020000	0x0400	<<7	
B9	D17																																	0x00400000	0x0200	<<13	
B8	D16									X																								0x08000000	0x0100	<<15	
B7	D15													X																				0x00800000	0x0080	<<12	
B6	D14														X																			0x00400000	0x0040	<<12	
spare	D41											X																						0x02000000			
spare	D40												X																					0x01000000			
spare	D39			X																														0x20000000			
spare	D38			X																														0x10000000			

For reads, "AND" with 0xCFCF300C to mask out unused bits.

Example for 12-bit writes: GPIO6_GDIR = 0xCFCFFFFF; //All set to outputs for 12-bit operation. Bits set to 1 = output, 0 = input with this command.

TEENSY 4.1 ONLY 16-Bit Port without bit scrambling - handle with physical wiring, not Teensy 4.0 compatible.

The green boxes [physical bit mapping of the internal 32-bit word] are already mapped to output pins: No bit masking required for 16 bit data, just shift uint16_t <<16 or scale data to uint32_t

Port 6 Bit->		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	uint16_t			
Bit Order	Port Pin																																	32-bit Hex Mask	Bitmask	Bitshifts	
spare	D0																														X			0x00000008			
spare	D1																															X		0x00000004			
MSB B15	D27				X																													0x80000000	N/A	<<16	
B14	D26					X																												0x40000000	N/A	<<16	
spare	D25																				X													0x00002000			
spare	D24																					X												0x00001000			
B9	D23									X																									0x02000000	N/A	<<16
B8	D22										X																								0x01000000	N/A	<<16
B11	D21																																		0x08000000	N/A	<<16
B10	D20						X																												0x40000000	N/A	<<16
LSB B0	D19																																		0x00100000	N/A	<<16
B1	D18																																		0x00020000	N/A	<<16
B6	D17																																		0x00400000	N/A	<<16
B7	D16										X																								0x08000000	N/A	<<16
B3	D15																																		0x00800000	N/A	<<16
B2	D14																																		0x00040000	N/A	<<16
B5	D41																																		0x02000000	N/A	<<16
B4	D40																																		0x00100000	N/A	<<16
B13	D39						X							X																					0x20000000	N/A	<<16
B12	D38							X																											0x10000000	N/A	<<16

For reads, "AND" with 0xFFFF0000 to mask out unused bits.

Figure 8: Pin map for parallel I/O on Teensy 4.X. Contiguous bit pins, physically scrambled, is my choice (lower of the three blocks).

A quick test yielded an update rate of 6.5MSPS with interrupts off (later, we get to how to handle interrupt priority so your Teensy 4.X does not freeze USB on your computer!). Remember, this is code-loop-timed (bracing for criticism!), on-the-fly, sine computation using `arm_sin_f32()`, so not too bad (and it gets better with `arm_sin_q31()`, but of course the bro-grammers already know that!).

So... where the heck is this going? I'll tell you. We are going to build a tutorial circuit with the vintage LTC1597 parallel DAC and squeeze the most performance out of it that we can. Why that chip? Because my dear friend Leonard Shtargot, at Linear Technology, now ADI, found 600 of them in a student-friendly DIP package and – gosh darn it – we are going to use them!

It's actually a really great 16-bit DAC, but not designed for waveform synthesis. Well, I'm not designed for getting up early, but I do it. VoidLoop also has a ton of AD8052 opamps, so we try those first and compare to see if anything we don't have in bulk is better (spoiler – generally not). So let's see what we can do.

If you look at Table 3, it is easy to believe one can actually hit -98dBFS noise floor and about 40% of Nyquist, but there are many other factors, like slew rate, ENOB (actual performance of the DAC), other noise sources, the performance of the reconstruction filter and on and on. So expectations should be modest here – let's aim for 100kHz, fairly flat output band for the frequency range assuming we will (for now) go with interrupt driven DDS (and prove the jitter is ok) at 1MSPS. Here “fairly flat” means we can use a Chebyshev filter with perhaps 0.05dB theoretical ripple. (The attentive reader may note that the theoretical range of about 40% of Nyquist should be 400kHz here, but the DAC performance cannot support it.)

Engineering The Thing

Our job, as engineers is typically to make what we have (stock, affordability, boss says so) work against some constraints that we can handle in production, given variations in parts and so on.

For any kind of mixed-signal synthesizer, it makes sense to look over the whole intended/specified output frequency range, at **in-band spurs**, which you cannot fix with an output filter (although theoretically by ensuring that the DDS sample rate is an integer multiple of the desired output frequency). The maximum SFDR achieved here will define the best spec for the system, even with a perfect (yeah, right) reconstruction filter above the band. This is a vital thing to understand. Of course – ugh – in-band spurs could be due to LUT truncation in DDS, so on-the-fly might be a different beast in some situations. And... remember that component datasheets are a game – they optimize for the best numbers regardless of typical use – *you need to build and test the actual system you are designing, physically.*

So, schematic below, I wired up the LTC1597 to the Teensy 4.1 (and, yet again, for TTL levels, 3.3V CMOS can drive them just fine in realistic situations, so no level-shifters are needed. But if you have 5V CMOS logic driving the Teensy inputs, you will dump current through the protection diodes to the 3.3V rail and – generally – fry the Teensy. Clear?

Quick checks on a 12-bit scope (R&S MXO34) suggested that the above-band spurs could be rendered lower, relative to in-band spurs using a 130kHz, 0.04dB ripple Chebyshev reconstruction filter as shown in Figures 9, 10 and 11 below, set to use the AD8052 and standard passive component values, using the ADI Analog Filter Wizard. The hand-built prototype is shown in Figure 12 below, schematic in Appendix 2 (note the programmable full-scale amplitude achieved by using a DAC8512, 12-bit voltage output DAC as reference for the LTC1597.

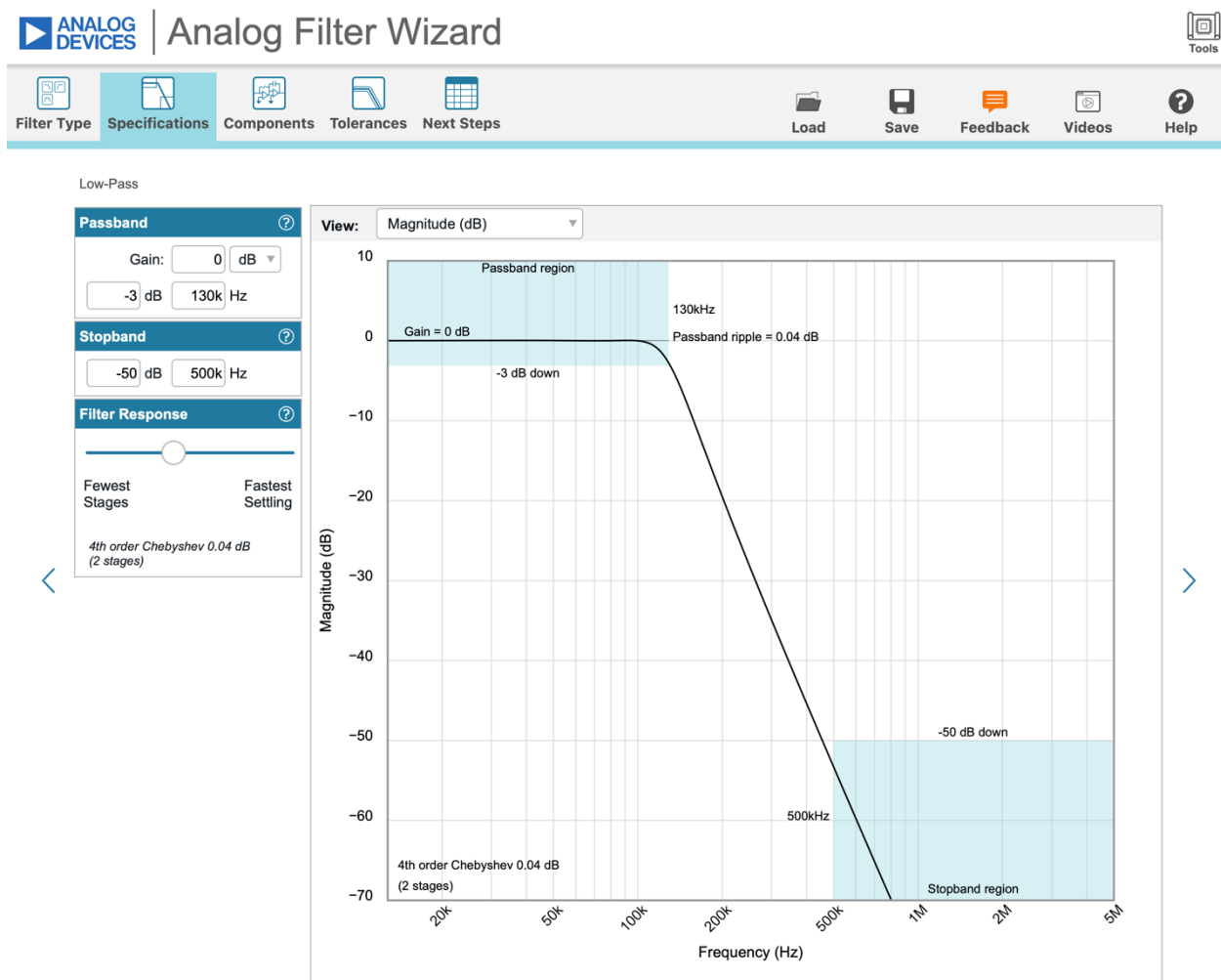


Figure 9: Theoretical response of the 130kHz, 4th-order, 0.04dB ripple Chebyshev filter to be tested.

Low-Pass; 4th order Chebyshev 0.04dB; 130kHz passband; Optimize: Specific Parts; +Vs: 5; -Vs: -5

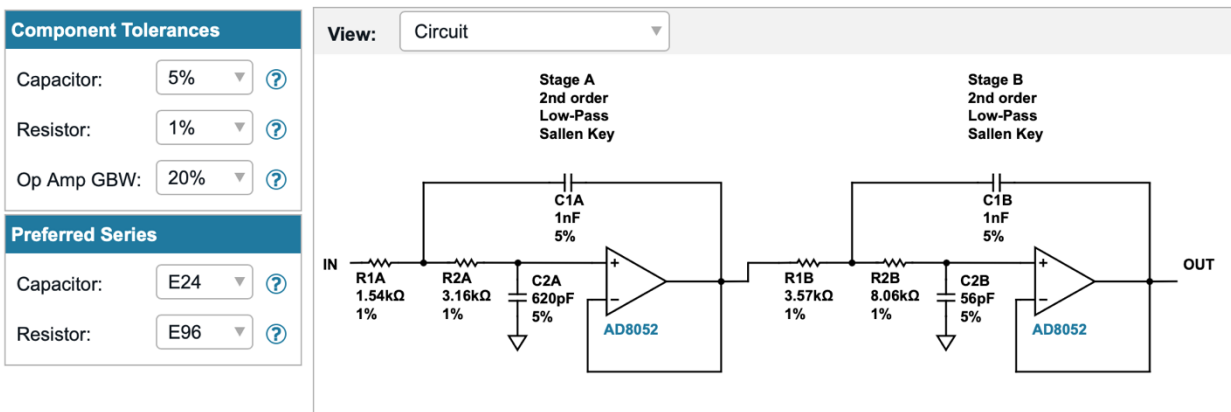


Figure 10: Filter schematic, showing 1% resistor and 5% capacitor choices.

Low-Pass; 4th order Chebyshev 0.04dB; 130kHz passband; Optimize: Specific Parts; +Vs: 5; -Vs: -5

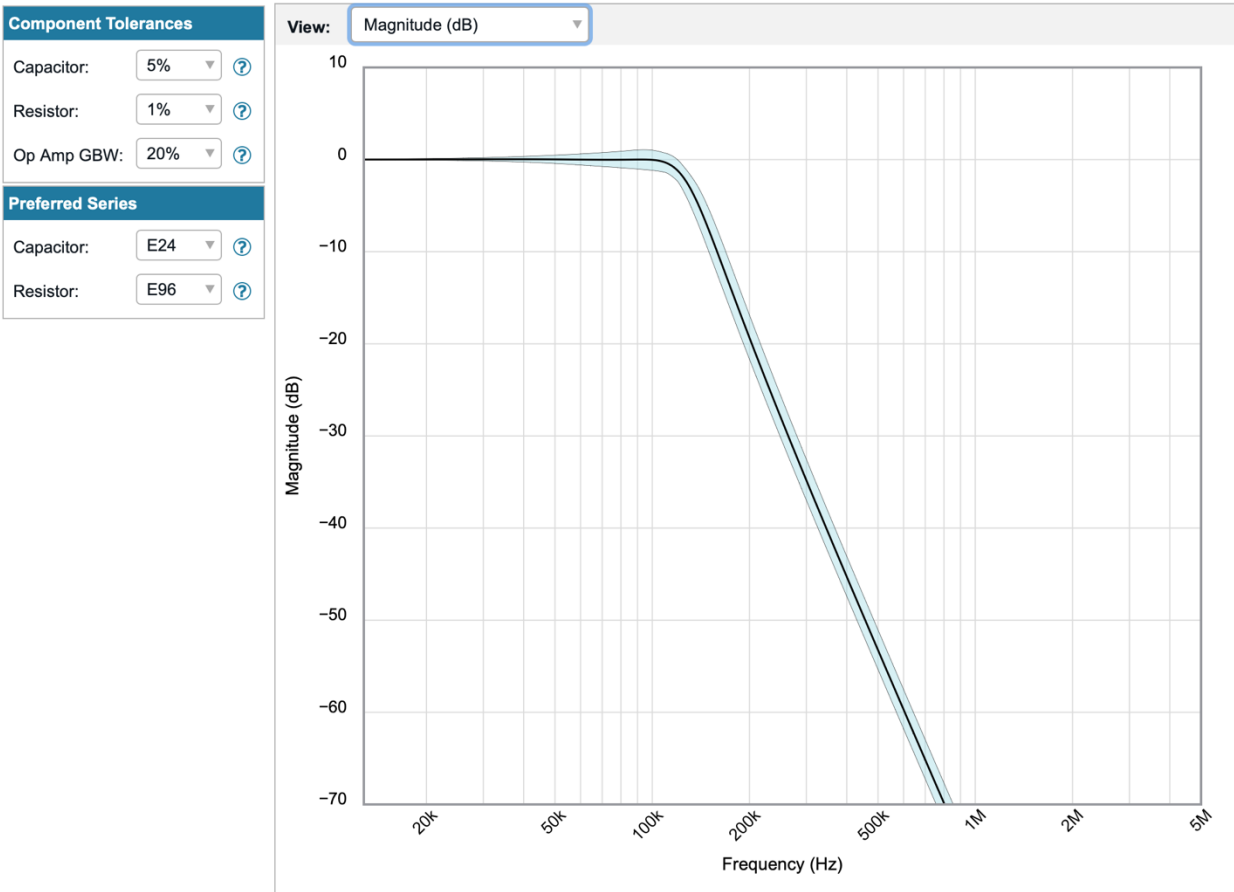


Figure 11: Monte-Carlo analysis of transfer function variations based on 1% resistors and 5% capacitors (looks decent).

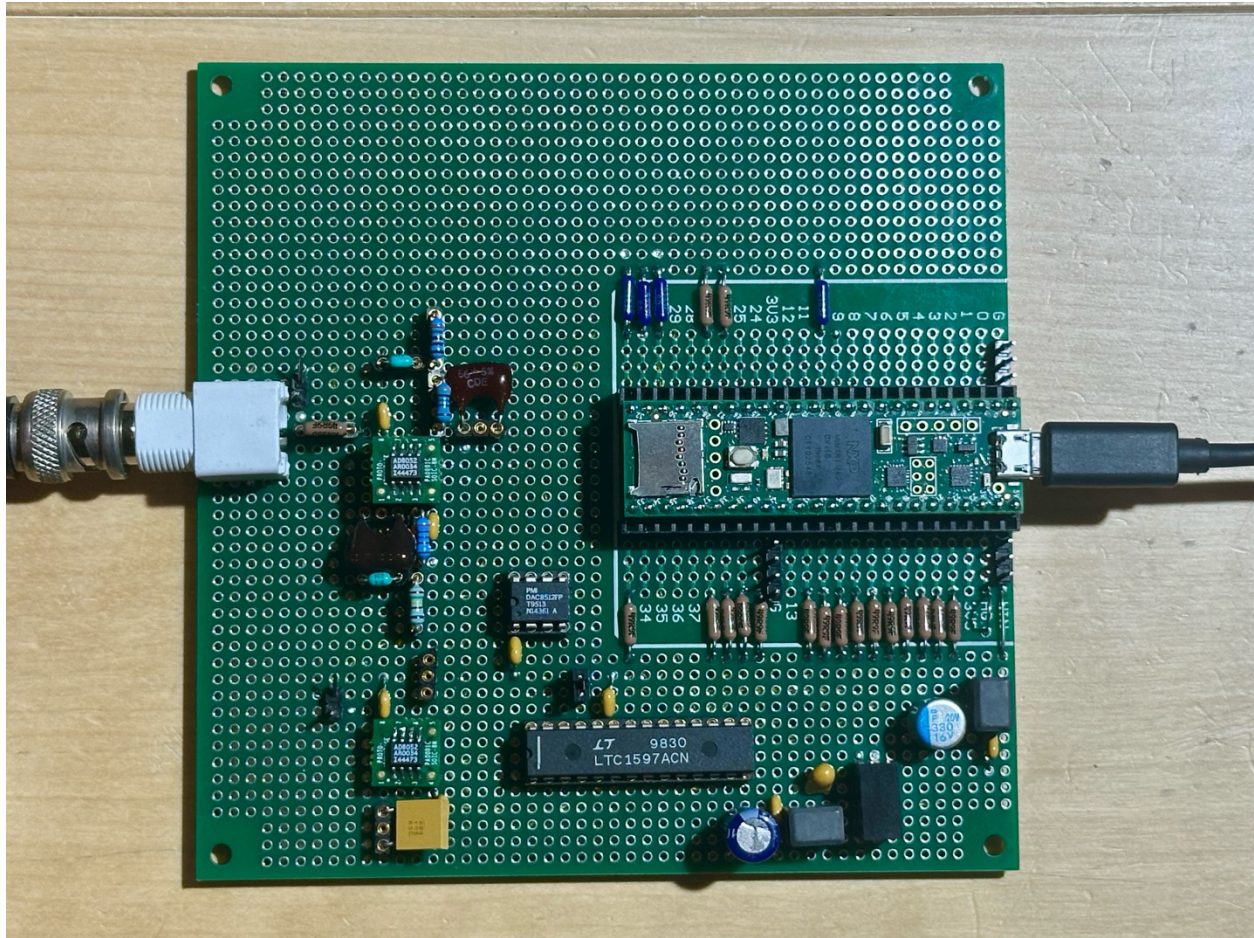


Figure 12: Photo of the hand-built prototype. At upper left is the AD8052 and passive components forming the reconstruction filter. At bottom is the LTC1597 DAC and AD8052, the latter providing both reference drive and I-to-V conversion at the output. Above the LTC1597 is the DAC8512 voltage-output SPI DAC, which can output 0.0 to 4.095V and sets the PP output voltage level into 50 Ω at that value (due to division by two via the 49.9 Ω series output resistor). At lower right are the power filters and the DC-to-DC converter that generates the -5.0V rail.

Results? Still on 12-bit scope before moving on to the SRS SR1, the bottom line is pretty good! SFDR is near the measurement limit (therefore somewhat suspect) but still pretty good. This testing was done using the `arm_sin_f32()` function (code in Appendix 3).



Figure 13: 10kHz, 3VPP, 76.2 dB SFDR, not bad! But never believe an SFDR close to that of the ADC in the scope... 😊

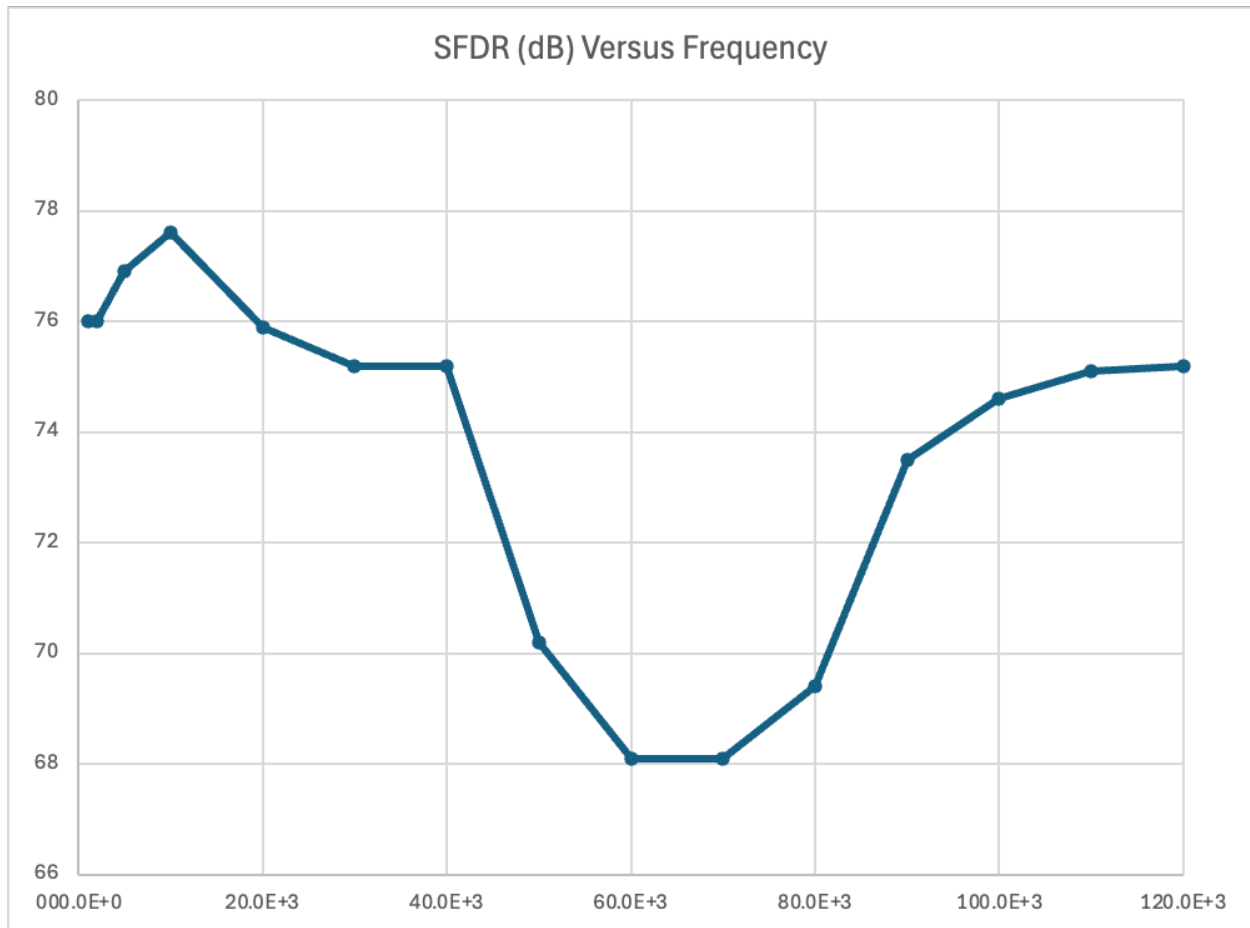


Figure 14: Rough SFDR versus output frequency on 12-bit scope with 3VPP output.

Swinging over to the trusty Stanford Research Systems SR1, we can see that the 10kHz SFDR is 84.9 dB, which is pretty good (Figure 15). This was measured at 2.00VPP into a 50 Ω load, which turned out to have the best SFDR.

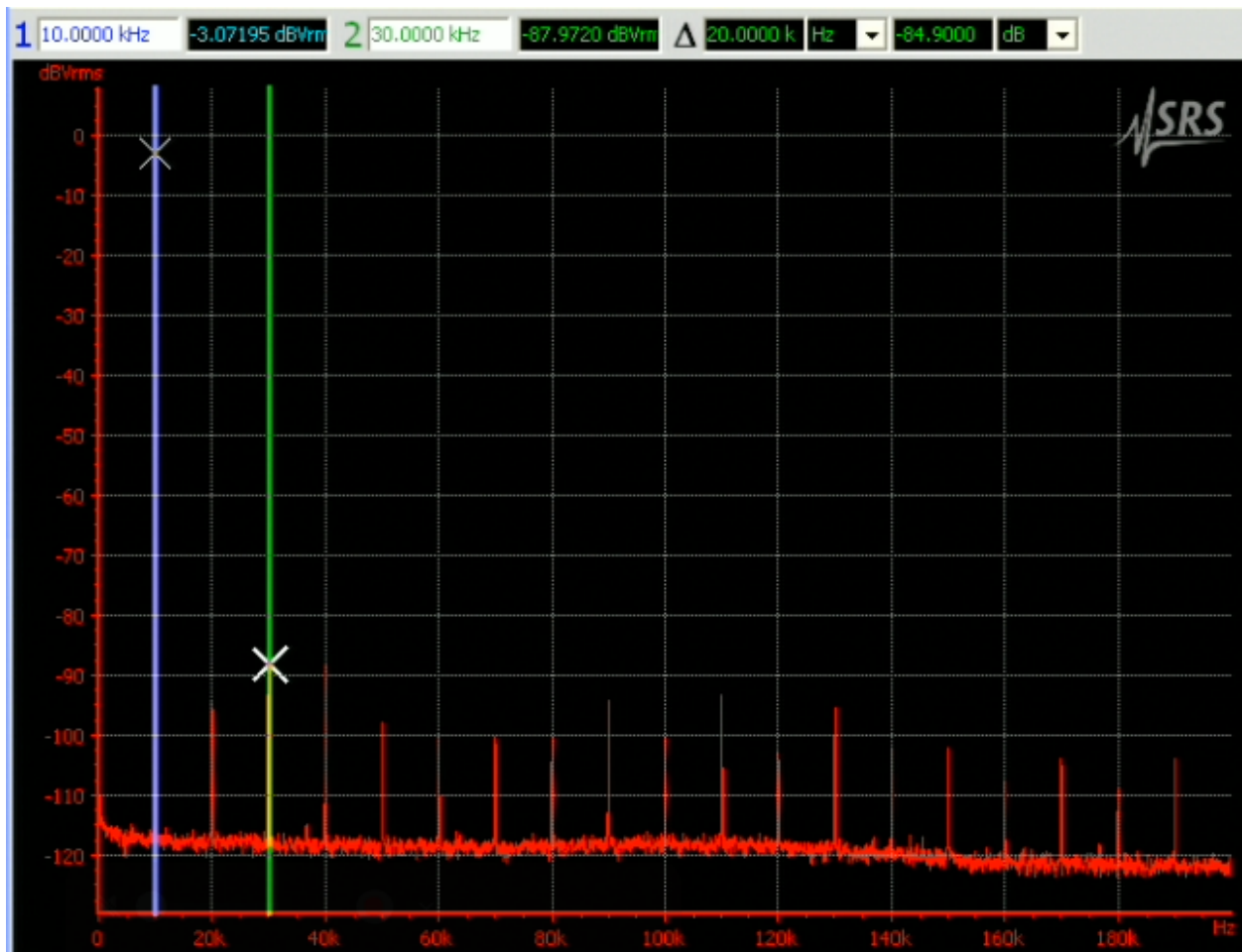


Figure 15: Example SFDR measurement on SR1 at 10kHz, 2VPP output.

Unexpectedly, when the `sin()` and `sinf()` functions were tested, SFDRs plummeted (by tens of dB). Ultimately it took the use of `Grok` to find out that the arm compiler (GCC, or Gnu Compiler Collection), which it described as “lazy,” could well schedule tasks out of the programmed sequence. In other words, when I tried to write the previous sample immediately upon interrupt servicing (minimum jitter) the compiler was putting the `sinf()` or `sin()` ahead of that, causing jitter in when the actual sample was written to the DAC. Holy smokes!

With that fixed, and a few more code clean-ups, the “final” SFDR was measured at several frequencies using the `arm_sin_q31()` function and some tricks (Appendix 3) to force the compiler to behave.

You may note that in the code, pin 9 is **toggle**d with each DDS cycle, and can be used with a frequency counter to examine sample rate and jitter if you want to.

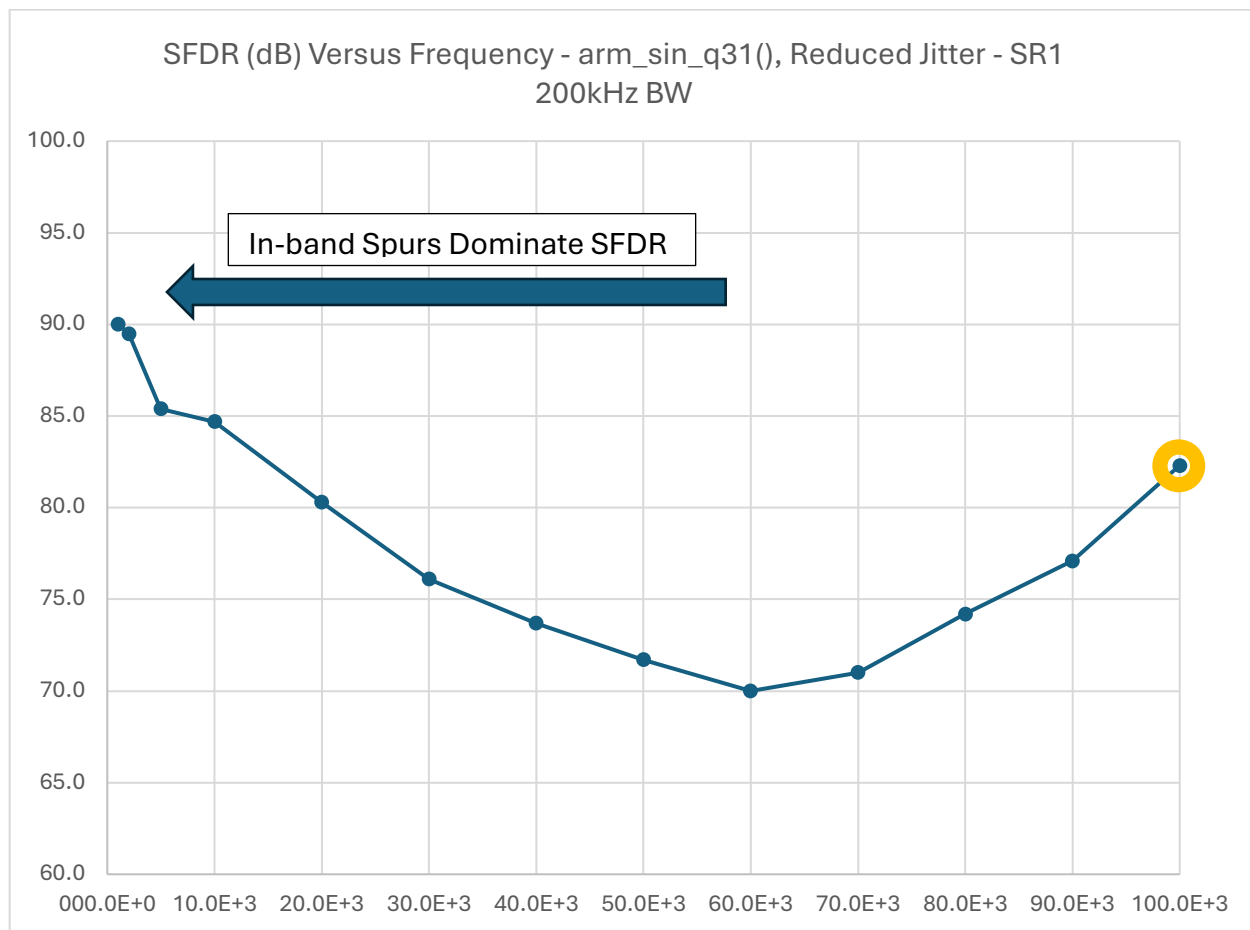


Figure 16: SFDR for the tutorial circuit measured on the SR1 up to the maximum of 90kHz given the SR1's maximum bandwidth of 200kHz. This data was obtained using the `arm_sin_31()` function after reducing code jitter as mentioned above. This might be a little generous at the higher end, but is believed to be correct after checking with the SR1 option to "show aliased lines," which did not show any higher/larger spurs up to the full SR1 Nyquist of 256 kHz, which showed an SFDR of 81.0 at 100kHz (orange circle) – the improvement at higher frequencies is due to the reconstruction filter's attenuation of higher harmonics/spurs.

So... for a DAC not designed for DDS applications, this is pretty good indeed. The SFDR (at 2.00VPP) bottoms out at 60kHz, coming in at 70dB. The best SFDR is about 92dB, at 1kHz, so it may not matter which sine function is used (as noted above, for these measurements it was `arm_sin_f32()`). There was no visible difference (and there should not be) with `arm_sin_q31()`.

Next, a look at the 100kHz output and a comment about spurs. This was done with 2.0VPP output seen on the SR1 – noting that the second and higher harmonics cannot be seen due to the maximum frequency the SR1 can display (200kHz). In figure 17 is the output for 100.000kHz and in Figure 18 is the output for 100.010kHz. You can see that the spurs are much worse for exact submultiples of the sample clock. DDS spur management is complex, and beyond the scope of this tutorial.

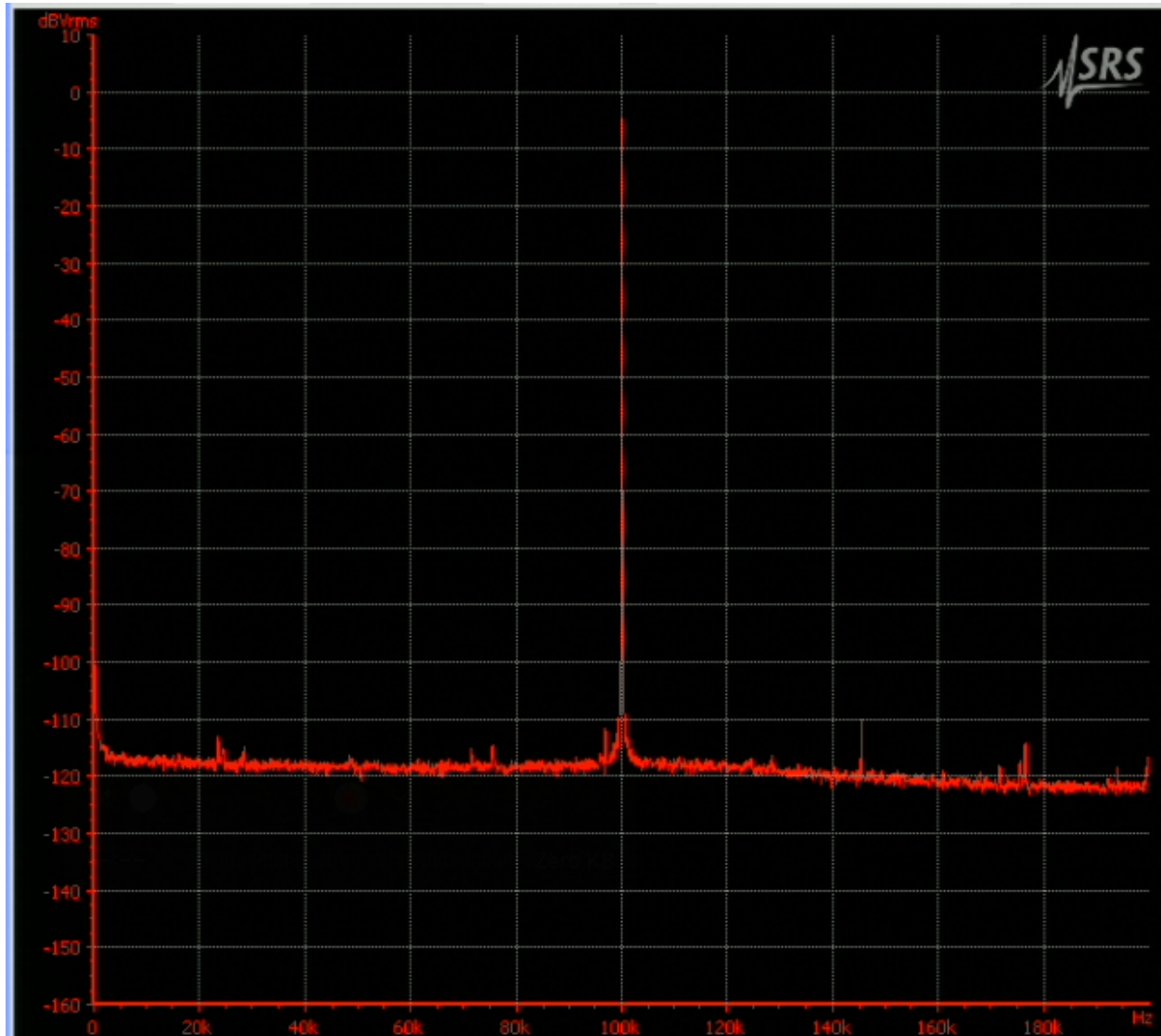


Figure 17: Output spectrum (to 200kHz) for an output frequency of 100.000kHz at a sample rate of 1MSPS.

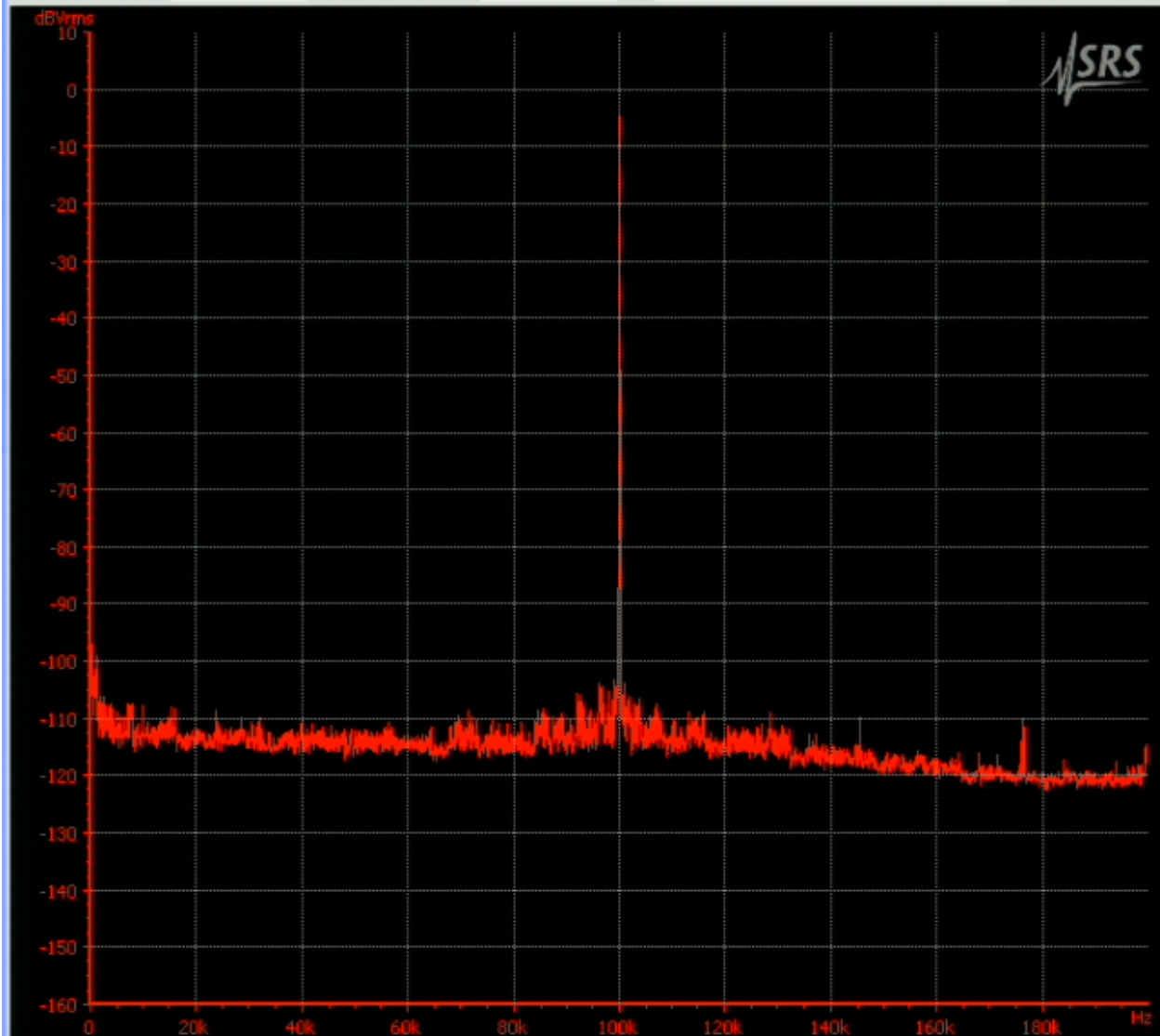


Figure 17: Output spectrum (to 200kHz) for an output frequency of 100.010kHz at a sample rate of 1MSPS, showing a slightly raised noise floor and wider skirt around the fundamental.

Possible improvements? Going beyond the intent of this basic tutorial, one could experiment with clock timing of the LT1597, including re-sampling the clock or try a higher-performance DAC, better designed for dynamic operation as in DDS. Of course, a dedicated DDS chip (more on that later) could be used, but then the internal mechanics would not be programmable nor very educational.

Appendix 3 contains the final test code for the tutorial, which includes hardware amplitude control and also includes the option of using SPI to drive the fast, 12-bit AD5452 DAC, which should be able to get close to 1 MSPS. More on that later.

Summary

The main take-aways from this tutorial are:

- Execution time and jitter comparisons of four possible sine computation functions that can be used on the Teensy 4.1 to probe the basic questions of “how fast can it go,” and “what is the best waveform quality possible?”
- Demonstration of fast parallel port I/O up to 16-bits, which can be applied to many projects.
- Design and testing of a basic demonstration circuit showing on-the-fly DDS synthesis, using parallel port I/O and hardware amplitude control (versus the floating point, mathematically-controlled amplitude demonstrated previously).

Hopefully this provides some good background and seeds some further explorations and learnings!

Appendix 1: Sine Computation Jitter Test Code (Sine_Jitter_Test_R0_0.ino)

```
//Sine value computation jitter test.
//G. Kovacs, 10/5/26

//#include <math.h> //note that arm.math.h is not already included with the Teensy
board library
#include <arm_math.h>

const int outputPin = 33;

uint32_t phaseAccumulator = 0;    //For now use a 32-bit accumulator, but consider
uint64_t - here 2^32 counts per rotation ("turn")
uint32_t sample;                  //Holds the 32-bit sample for each DDS clock cycle

static const float countsPerRotation = 4294967295.0;
// 2^32 - 1, maximum value of phase accumulator before roll-over to zero
static const float fullScaleAmplitude = 2147483647.5;
// 2^31 - 0.5, to allow for rounding
static const float radiansPerPhaseCount = 2.0f * M_PI;
// Scale 0 to 0.999999... phase accumulator to 0 to 2PI for sine functions that need
it (arm_sin_q31(), does not!)

float    outputFrequency = 10.0E3;
float    sampleRate = 10.00E6; //placeholder - will measure the actual sample rate
uint32_t phaseIncrement = (uint32_t) outputFrequency * countsPerRotation /
sampleRate; //Phase step per sample

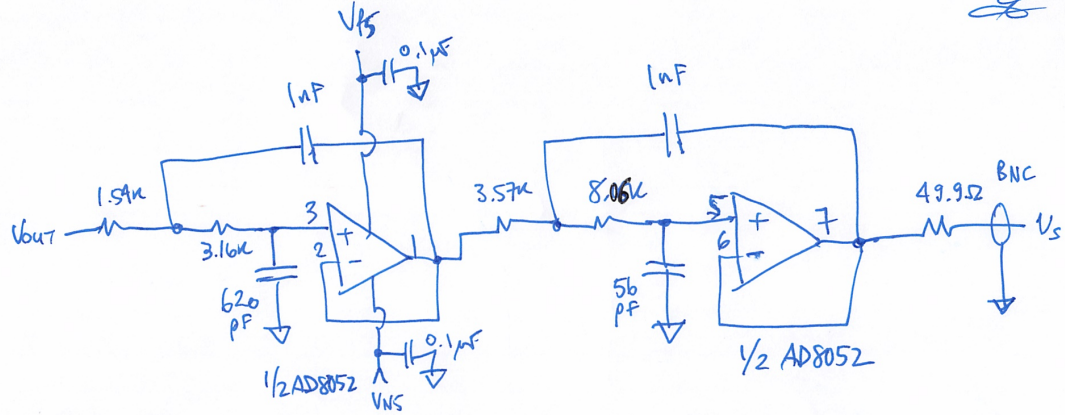
void setup() {
    pinMode(outputPin, OUTPUT);
    noInterrupts();
}

void loop() {
    digitalWriteFast(outputPin, HIGH);
    //Uncomment the sine function you want to test and look at the output pin to see the
    execution time and jitter.
    //sample = (uint32_t) ((fullScaleAmplitude * sin((double) radiansPerPhaseCount *
    phaseAccumulator/countsPerRotation)) + fullScaleAmplitude);
    //sin(), double precision with argument explicitly cast as double, shows significant
    jitter based on argument, which is treated modulo 2pi
    //sample = (uint32_t) ((fullScaleAmplitude * sin(radiansPerPhaseCount *
    phaseAccumulator/countsPerRotation)) + fullScaleAmplitude);
    //sin(), double precision, shows significant jitter based on argument, which is
    treated modulo 2pi
    //sample = (uint32_t) ((fullScaleAmplitude * sinf(radiansPerPhaseCount *
    phaseAccumulator/countsPerRotation)) + fullScaleAmplitude);
    //sinf(), single precision and 2X faster than sin() shows significant jitter based on
    argument, which is treated modulo 2pi
    //sample = (uint32_t) ((fullScaleAmplitude * arm_sin_f32(radiansPerPhaseCount *
    (phaseAccumulator/countsPerRotation))) + fullScaleAmplitude);
    //arm_sin_f32() returns at a constant time regardless of argument, but argument must
    be 0..2pi
    sample = (uint32_t)(fullScaleAmplitude * (1.0f +
    (float)arm_sin_q31((q31_t)(phaseAccumulator >> 1)) * (1.0f / 2147483648.0f)));
    //arm_sin_q31() returns at a constant time and the argument rolls over at 1.0000 so
    not scaled as a multiple of pi

    digitalWriteFast(outputPin, LOW);
    phaseAccumulator += phaseIncrement;
}
```

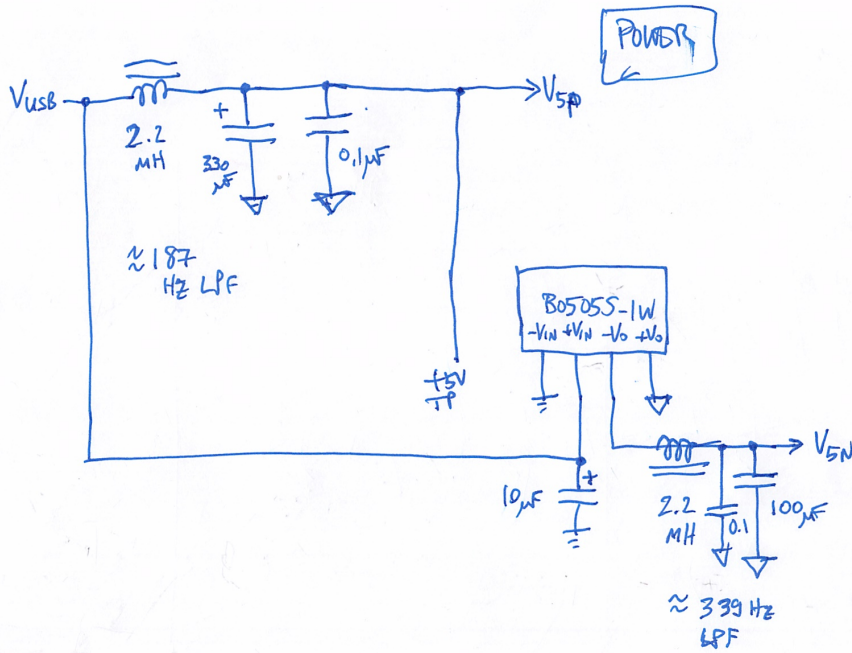

RECONSTRUCTION FILTER

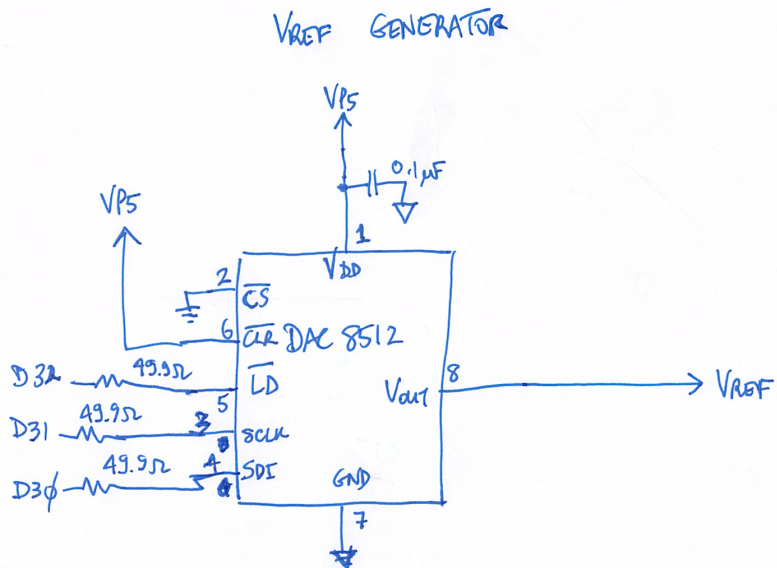
9/30/26

130 KHz, 4th - ORDER, CHEBYSHEV 0.04 dB .

3/27/26
~~8~~





SPI PINS :

SCLK	13 (LED)
MISO	12
MOSI	11
OPTIONAL → CS	10

Appendix 3: Code to Demonstrate Hardware-Amplitude-Controllable On-the-Fly DDS Using Parallel Port I/O

```
/**
 * On-the-Fly_DDS_Speed_Parallel_R1_0.ino
 * @version 1.00
 * @date 10/6/26
 * G. Kovacs, VoidLoop
 * Grok used for testing, compiler issue mitigation, and cleanup.
 *
 * Basic speed testing of on-the-fly DDS. Output can be the 16-bit parallel
 * port (LTC1597), hardware SPI0 (AD5452), or bit-banged SPI (DAC8512).
 * The test circuit used LTC1597 as the DAC, with the DAC8512 controlling amplitude.
 * The sample clock is an IntervalTimer, not a calibrated code loop.
 *
 * arm_sin_f64() does not exist in CMSIS-DSP. arm_sin_q31() is the low-harmonic
 * path: the phase accumulator is already the angle, so there is no modulo-2pi
 * step and no float rounding of the phase word.
 *
 * Select the sine with SINE_MODE at the top. The latch sequence does not change.
 * 16 contiguous Teensy 4.1 port pins, MSB to LSB:
 * 27, 26, 39, 38, 21, 20, 23, 22, 16, 17, 41, 40, 15, 14, 18, 19
 */

#include <arm_math.h>
#include <SPI.h>

// 0 arm_sin_q31 integer phase, lookup table + linear interpolation. Lowest
harmonics.
// 1 arm_sin_f32 same table, but the angle is built in float radians.
// 2 sinf libm, single precision. Variable execution time.
// 3 sin libm, double precision. Variable execution time.
#define SINE_MODE 0

IntervalTimer ddsTimer;

const int parallelCSpin = 33;
const int spi0CSpin = 10;
const int bitbangLDPin = 32; // "-LD" pin on the DAC8512
const int isrAlivePin = 9; // toggles every ISR; not part of the DAC port

// Bit-banged SPI, any free pins
const int DAC8512clock = 31;
const int DAC8512sdi = 30;

// Hardware SPI0: SCK = 13 (LED), MISO = 12, MOSI = 11, CS = 10 (or any GPIO)

uint32_t phaseAccumulator = 0; // 2^32 counts per turn. Wraps on its own.
uint32_t phaseIncrement; // counts added each sample
uint32_t sample; // offset-binary word latched into the LTC1597

static const float countsPerRotation = 4294967296.0f; // 2^32, exact in float. Not
2^32-1.
static const float fullScaleAmplitude = 2147483647.5f; // 2^31 - 0.5, float paths only
static const float radiansPerPhaseCount = 2.0f * M_PI; // used only by modes 1, 2, and
3

//float outputFrequency = 100.0E3;
float outputFrequency = 100.000E3;
float clockFrequency = 1.000E6; // IntervalTimer rate
```

```

float isrIntervalMicroseconds = 1.0f;      // 1E6/clockFrequency, written out so it
cannot collapse to 0
float PPamplitude = 2.0000;                // DAC8512 peak-to-peak into 50 ohm (49.9 ohm
series divides by 2)

// 30 MHz, MSB first, mode 0 (AD5452 supports mode 0 or 3, up to 50 MHz)
SPISettings AD5452Settings(30000000, MSBFIRST, SPI_MODE0);

// Build the next DAC word from the phase accumulator.
// Mode 0 never leaves integer arithmetic. The other three share one float
// radian value, so a difference between them is the sine routine, not the phase word.
uint32_t sineSample(uint32_t phase) {
    #if SINE_MODE == 0
        // Top 31 bits are the CMSIS Q31 angle: 0 .. almost 1 turn maps to 0 .. almost 2pi.
        // arm_sin_q31 returns a signed Q31 value. Adding 2^31 makes offset binary
        // (0x00000000 = -full scale, 0x80000000 = 0, 0xFFFFFFFF = +full scale).
        return (uint32_t)arm_sin_q31((q31_t)(phase >> 1)) + 0x80000000u;

    #elif SINE_MODE == 1
        // phase/countsPerRotation is float, so counts finer than 2^(N-24) are lost
        // once the accumulator is above 2^24. That quantization is periodic and becomes
        harmonics.
        return (uint32_t)(fullScaleAmplitude * (1.0f + arm_sin_f32(radiansPerPhaseCount *
        (phase / countsPerRotation)))));

    #elif SINE_MODE == 2
        return (uint32_t)(fullScaleAmplitude * (1.0f + sinf(radiansPerPhaseCount * (phase /
        countsPerRotation)))));

    #elif SINE_MODE == 3
        // The multiply is still float. sin() then promotes that already-rounded radian
        value.
        return (uint32_t)(fullScaleAmplitude * (1.0f + sin(radiansPerPhaseCount * (phase /
        countsPerRotation)))));

    #else
        #error SINE_MODE must be 0, 1, 2, or 3
    #endif
}

void setup() {
    pinMode(isrAlivePin, OUTPUT);
    digitalWriteFast(isrAlivePin, LOW);

    // Idle levels used by the sketch that produced a waveform
    pinMode(spi0CSpin, OUTPUT);
    digitalWriteFast(spi0CSpin, LOW);          // deassert SPI0 CS
    pinMode(parallelCSpin, OUTPUT);
    digitalWriteFast(parallelCSpin, LOW);      // deassert parallel -WR
    pinMode(bitbangLDpin, OUTPUT);
    digitalWriteFast(bitbangLDpin, HIGH);      // deassert DAC8512 -LD
    pinMode(DAC8512clock, OUTPUT);
    pinMode(DAC8512sdi, OUTPUT);

    phaseIncrement = (uint32_t)((outputFrequency / clockFrequency) * countsPerRotation);

    bitBangWriteDAC8512(PPamplitude);

    SPI.begin();
    GPIO6_GDIR = 0xFFFFFFFF;                  // 16-bit parallel port, all outputs

```

```

    phaseAccumulator = 0;
    sample = sineSample(0); // first edge must be a real code, not an
uninitialized word
    writeParallel(sample);

    ddsTimer.begin(DDSIsr, isrIntervalMicroseconds); // 1 us
    ddsTimer.priority(0); // highest, to keep the sample clock from
being delayed
    SCB_SHPR3 = 0x20200000; // SysTick below this timer
}

void writeParallel(uint32_t value) {
    GPIO6_DR = value; // upper pins are the ones bonded out;
write them first
    digitalWriteFast(parallelCSpin, LOW); // LTC1597 starts taking data
    delayNanoseconds(90); // minimum -WR low time
    digitalWriteFast(parallelCSpin, HIGH); // latch. High time is whatever remains
of this ISR.
}

void writeAD5452(uint32_t value) {
    // Top 12 bits, two control bits held 0: (value >> 18) & 0x3FFF
    uint16_t spiWord = (value >> 18) & 0x3FFF;
    SPI.beginTransaction(AD5452Settings);
    digitalWriteFast(spi0CSpin, LOW);
    SPI.transfer16(spiWord);
    digitalWriteFast(spi0CSpin, HIGH);
    SPI.endTransaction();
}

void bitBangWriteDAC8512(float PPvoltage) {
    // Assumes a 50 ohm load. High-Z gives twice the voltage passed in.
    // DAC8512: 0x000 = 0.000 V, 0xFFF = 4.095 V
    uint16_t dataDAC8512 = (uint16_t)((PPvoltage / 4.095) * 4095.0);
    digitalWriteFast(bitbangLDpin, LOW);
    delayNanoseconds(10);
    shiftOut(DAC8512sdi, DAC8512clock, MSBFIRST, byte(dataDAC8512 >> 8));
    shiftOut(DAC8512sdi, DAC8512clock, MSBFIRST, byte(dataDAC8512 & 0x00FF));
    delayNanoseconds(10); // -LD hold time
    digitalWrite(bitbangLDpin, HIGH);
}

void loop() {
    //Nothing to do here (yet)... :)
}

void DDSIsr() {
    digitalWriteFast(isrAlivePin); //For example, 500 kHz on pin 9 while 1 us timer
is running

    // Strobe the sample computed last time, before any sine work. sinf() and sin()
    // take a data-dependent number of cycles; if that ran first, the -WR edge
    // would move with the phase and the spectrum would grow harmonics.
    writeParallel(sample);

    // Compiler barrier. volatile on sample is not enough: GCC (Gnu Compiler Collection)
may still evaluate
    // the next sine call before this store, then only write the variable later.
    // The empty asm cannot be deleted ("volatile"), and "memory" tells GCC that
    // memory may have changed, so it must not move loads or stores across it.
    asm volatile("" ::: "memory");
}

```

```
    phaseAccumulator += phaseIncrement;  
    sample = sineSample(phaseAccumulator);  
}
```