

Tutorial:

Pushing the PCM5102A Dual Audio DAC Beyond Audio

Author: Greg Kovacs (code by Zachariah Magee with inputs from Greg)

Date: 9/18/26

Revision: 0.3

Introduction

The PCM5102A dual audio DAC by Burr-Brown (Texas Instruments) is a fabulous, low-cost (about \$2 at 2kU, \$4 in singles), high-performance chip typically used for consumer audio applications. One of the best things about it is that it has an onboard charge pump that generates a negative power rail, allowing the output to swing symmetrically around ground, DC-coupled, almost 6VPP! This matters for users who might want low frequencies, which would require gigantic output coupling capacitors.

Much has been written about interfacing it to Arduino-family processors including the Teensy, which was used here. An excellent online resource is the tutorial and library put together by Phil Schatzmann (<https://github.com/pschatzmann/arduino-audio-tools/wiki/External-DAC>).

The main purpose of this tutorial is to show how, by trading off some signal-to-noise ratio, the PCM5102A can be pushed to at least 60kHz while retaining the ability to generate very low frequencies, essentially down to DC (an example 0.1Hz sinewave – very clean – is shown in Figures 1 below, and then 0.01Hz and 0.001Hz in Figures 2 and 3).

As a secondary concept, the use of “on-the-fly” DDS (direct digital synthesis) is demonstrated, where instead of computing a waveform LUT (look-up-table) in RAM, the Teensy 4.1’s incredible floating-point computation speed allows for the waveform amplitude values to be computed on-the-fly, effectively creating an infinitely long LUT (as a side note, it is common practice for a DAC of “N” bit resolution to have a DDS LUT of two to four times deeper resolution... in other words for a 12-bit DAC with 4096 possible output levels, a 14-

bit LUT is nice, but for 16-bit words, that requires 32kB!). The Teensy 4.1 can compute the values for both DACs, on-the-fly (e.g., for two frequencies, or quadrature output) without breaking a sweat (based on my experience with 1MHz, 1 μ s ISRs), using an interrupt-driven routine. Code-driven amplitude control, amplitude modulation, and generation of other waveforms (noise is demonstrated here) is also possible with this very low-cost approach.

Note: a great tutorial on DDS can be found here (a basic read for sure):

<https://www.analog.com/en/resources/analog-dialogue/articles/all-about-direct-digital-synthesis.html>

A much more technical tutorial, digging into the more nuanced bits and beyond can be found here:

<https://www.ieee.li/pdf/essay/dds.pdf>

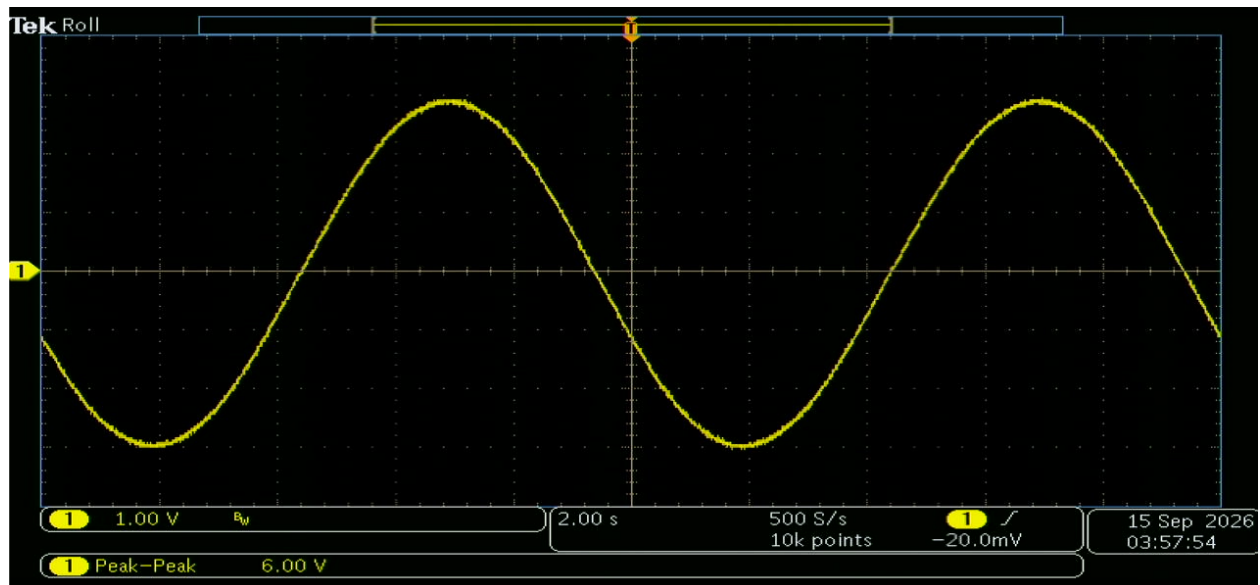


Figure 1: Example of a clean 0.1Hz sinewave using the PCM5102A. No realistic AC-coupled DAC can do this. I did test it down to 0.01 Hz (below), but got tired of waiting so I did not go lower. Point made. But then...

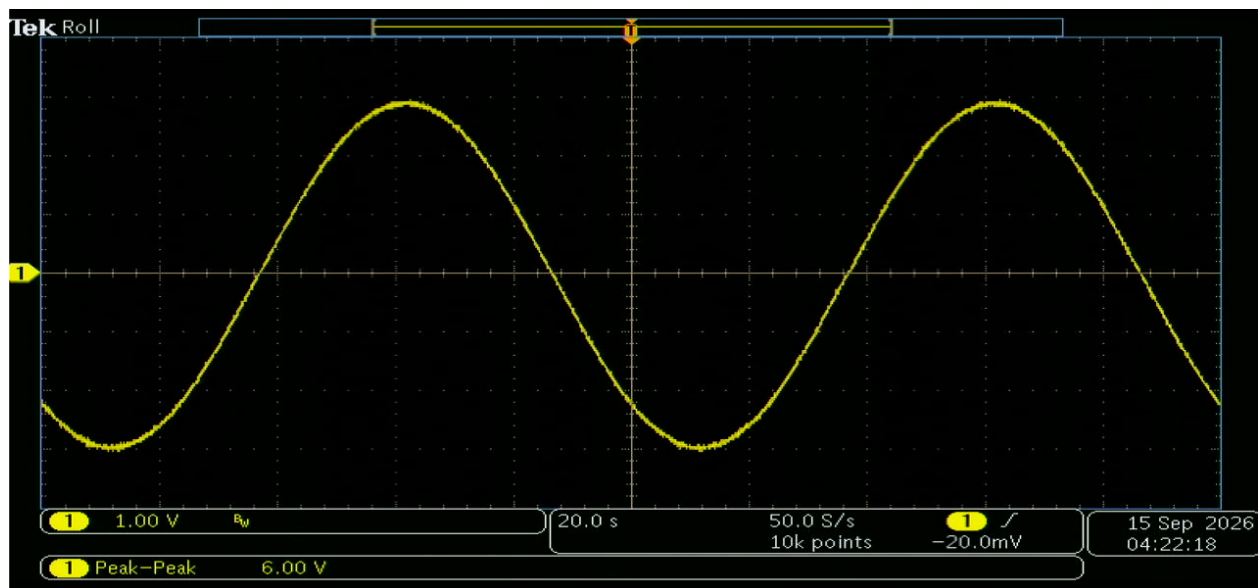


Figure 2: Really, it can go low... here is 0.01Hz at a scope setting of 20s/division!

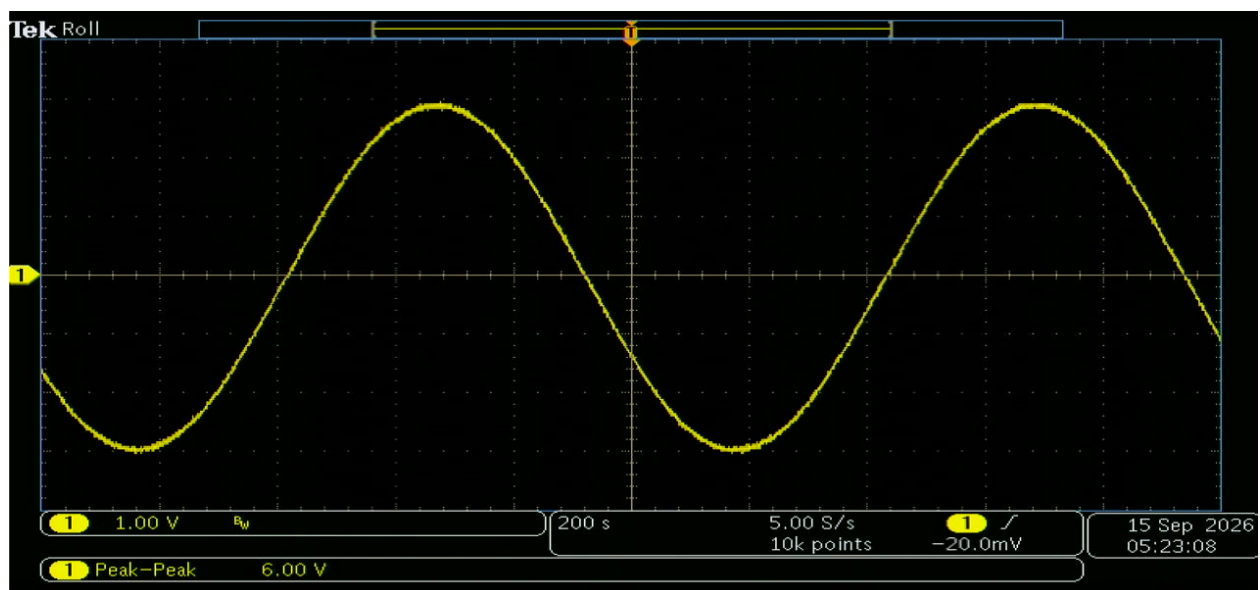


Figure 3: Ok – I have other things to do, but here is 0.001Hz.

The bottom line is that with a simple, inexpensive analog filter designed to suppress folded harmonics from the DAC, the SFDR can be held to about 60dB at 60 kHz, which is comparable to decent benchtop function generators (good ones, like the Keysight 33600B series, can do about 76dB at 1kHz – just checked a 33622B, but remember that we can get

90dB at lower frequencies with this chip). Here I should mention the iconic SRS DS360, which has insanely low harmonics/spurs and is very handy to have for testing ADCs.

For frequencies below 20kHz, the much better SNR of the PCM5102A is preserved (in these simple prototypes) to about 90dB with a good choice of opamps for the output filter used to get us clean signals to 60kHz (described below). So if you don't need the full bandwidth, you can just get better SFDR (still improved over the basic PCM5102A modules). Of course (this is a tutorial) you could implement a steeper-cutoff filter and get better performance at the higher frequencies. If you only need lower-frequency signals, this is a pretty good way to get them, or you can use our code with just the basic PCM5102A modules – our pleasure to help! It's all up to you.

Some Details

The PCM5102A datasheet is an essential learning reference (always read them, please) and can be found here:

<https://www.ti.com/lit/gpn/PCM5102A>

The chip is specified to produce waveforms with a dynamic range of 112 dB, an SNR of 112 dB and a THD+N (total harmonic distortion + noise) of -93dB – truly impressive specs. While it takes 32-bit data, it performs roughly like an 18-bit DAC. As mentioned above, it has a built-in charge pump so that it can operate at sub-Hertz signals – basically down to DC. The sample rate can be run as high as 384kSPS and for a typical DAC, that means one could use DDS to about 60 – 70% of the Nyquist frequency so that the (low-pass) reconstruction filter can roll off to at least the depth required by the theoretical DAC noise floor in dB, for N bits, approximated by:

$$SNR \approx 6.02N + 1.76$$

For high-resolution DAC's, it is difficult to construct a suitable analog filter without too much complexity. If we max it out at 8th-order and (for now) approximate the slope from the simple Butterworth, maximally flat passband filter, we have a slope of (8)(20dB/decade), or 160 dB/decade. Sounds impressive, but if one has the -3dB (cutoff) frequency at, say, 60 kHz, by the 384kSPS Nyquist of 192kHz, the roll-off will be only -80.8 dB from this approximation:

$$Atten = 160 \log_{10} \left(\frac{60kHz}{192kHz} \right)$$

(Note that the log of the ratio of frequencies is a handy trick to calculate how many “decades” of frequency the filter can attenuate over.) Of note is that ≈ 81 dB of attenuation is only enough for (use the SNR formula to find N) a DAC of about 14-bit resolution! This is a good example of the fundamental trade-space in design of DDS systems, and one can certainly use filter types like Chebyshev or elliptical (Cauer) which trade ripple in the pass-band for steeper roll-off. For this tutorial, we’ll stick with Butterworth, but a nice graphical representation from Bliley Technologies (<https://blog.bliley.com/filter-typology-face-off-a-closer-look-at-the-top-4-filter-types>) is below in Figure 4.

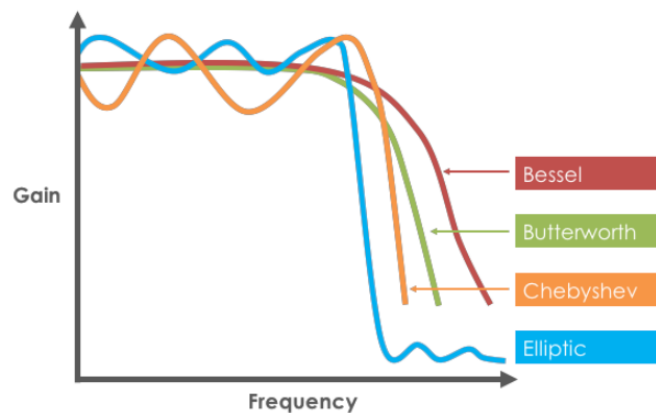


Figure 4: Illustration of common low-pass filter responses, showing the trade-off between pass-band and stop-band ripple and steepness of roll-off. Phase is also a consideration, not covered here. The illustration is from the Bliley web site (thanks) cited above.

So here, we are not going to get that aggressive – just go to 31% (60kHz) of the Nyquist frequency, which for the maximum sample rate of the PCM5102A of 384kSPS is 192kHz. It is worth noting that the PCM5102A’s basic circuit includes a first-order RC filter (470 Ω resistor and 2.2nF capacitor) with cutoff at 154kHz (Figure 40, PCM5102A datasheet). That helps.

Also – remember – amplitude matters. Sometimes if you digitally drive a DAC to less than full-scale, harmonics and spurs decrease. Pushing the silicon less hard helps because it is basically nonlinear stuff, beaten into submission to be linear. Often worth repeating the FFT at lower amplitudes.

Hardware

There are many commercial PCM5102A modules available at low cost. The ubiquitous Chinese modules are of moderate build quality but easy to find online. A much higher-quality module is available from Adafruit at a comparable cost. Both are shown below in Figure 5 below. In my opinion, the Adafruit board is far superior because the pins are aligned to 0.1” “DIP” spacing and will fit on any protoboard or plugboard. The Adafruit version also has a clean +3.3V output from the LDO on the module. Schematics and connections for both types of module are in Appendix 1.

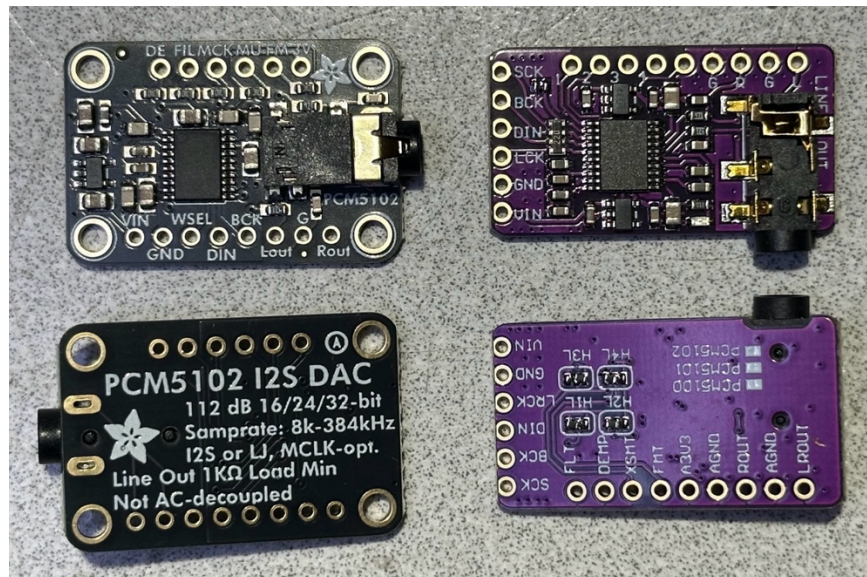


Figure 5: Left – Adafruit PCM5102A module, right – generic Chinese PCM5102A module, either of which will work fine for this tutorial, but the Adafruit version is much better build quality and has pins that will align with 0.1” holes in a white “plugboard” or PCB protoboard.

Test Code

Our test code, based on a lot of work by the community and thanks to initial PCM5102A exploration by Mark Flamer and fleshing out by our very own Zachariah Magee, is below in Appendix 2. It can be used to generate fixed frequencies or sweeps and, because it is meant for testing, it outputs the same frequency on both channels with a defined phase shift (e.g., 90° for quadrature). This is the easy-to-understand version. A “full-featured” version that does much more is included in Appendix 3.

Of educational note here is that the DDS output values are computed on-the-fly on each interrupt (which can be as high as 384kHz, or 2.60µs) for both channels in double-precision and converted to `int32_t` before being sent to the PCM5102A. *How cool is that?* I first demonstrated on-the-fly (no LUT) DDS using the Teensy 3.6 in 2020, so the idea had some time to ripen. Using this in the test code, the entire DDS accumulator is used (not just the top N bits, which are usually truncated to match the LUT size) to minimize DDS LUT-related spurs (sort of like having an “infinite” LUT relative to other sources of DDS spurs). This is just another example of the amazing computational power of the MIMXRT1062DVJ6A chip that powers the Teensy 4.1. Very (nerdy) cool!

Initial Testing – Mapping the Potential

It is very important to point out that while there are several well-intentioned sources on the internet showing the harmonic content of these modules, most were done on inexpensive digital oscilloscopes that, at best, have 8-bit resolution. Revisiting the dynamic range equation above, that means, at best a dynamic range of about 50dB. So when the “FFT” function is used, there is no way to see harmonics and spurs down where we need to go, around -90dBc (dBc means relative to the carrier, or main frequency’s amplitude). Please don’t do that or ingest that kind of result unless you have a high-resolution oscilloscope as is used below (in this case it is a 12-bit scope).

Instruments that can actually do the job include the Stanford Research Systems SR1, Pioneer Hill’s Spectra DAQ-200, Quantasylum’s QA403, Audio Precision’s several models, and others. All but the AQ403 are quite expensive though... Of course there are tools that use the audio inputs of computers, but those are generally limited to the 20Hz – 20kHz audio band, and we need more here.

I have the luxury of using the SR1 (thank you John Willison and SRS!) – it really does the job with precision, and also keeps me fresh on Windows XP Embedded...

Data Rate Validation

I used the test code in Appendix 2 (for advanced use, we recommend the enhanced code in Appendix 3, but the first version is far simpler and easier to understand).

Just to be sure (always a good thing) the “DIN” signal was checked. This was using the “fly-wire” connection, so the signal is not clean (would also usually put a series “damping” resistor of maybe 49.9 or 100Ω on each digital line, on the end sending the signal), but should be (32 bits per channel)(2 channels)(384kSPS), or about 24.57MHz... Figure 6 below shows that we are, in fact, getting the desired sample rate (the error in frequency displayed on the scope is just because of the way it is estimating it... this model, Tektronix MDO3054 does not have an actual frequency counter, but estimates frequency from the waveform). Interestingly, all of the overshoot/ringing in the signal makes the 3.3V logic level look like almost 4.7VPP!

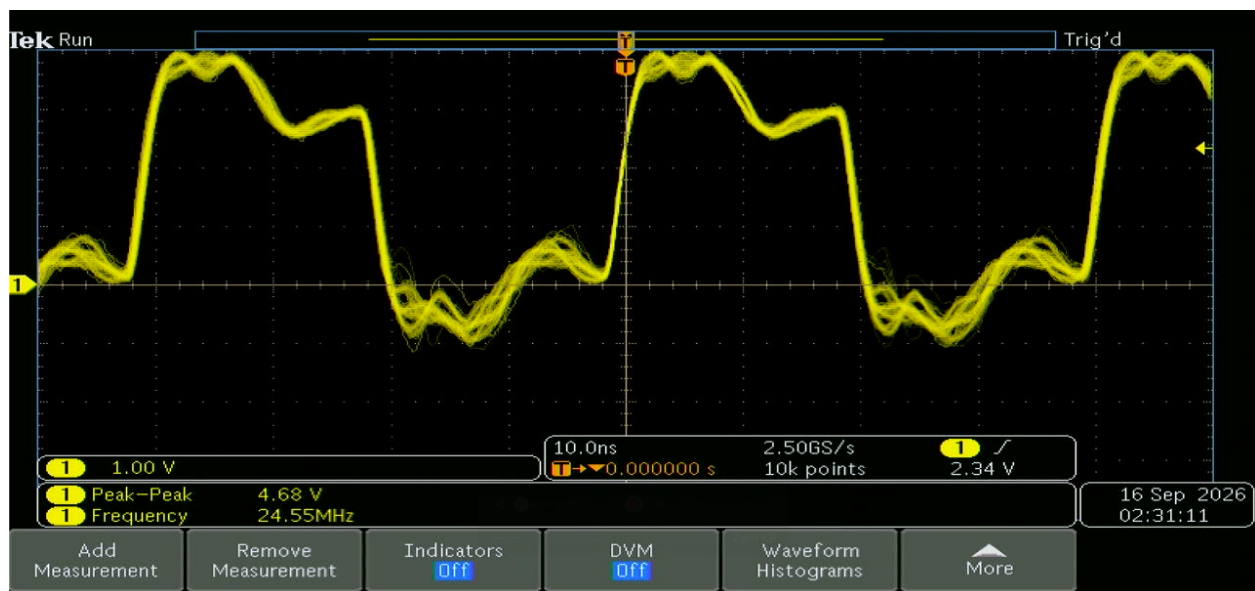


Figure 6: PCM5102A DIN signal measured at the Adafruit PCM5102A module (some ringing present due to “fly-wire” hookup). Good to know we have the correct frequency of bits flying to the chip!

Evaluation of Output Spectrum

The first test was with a 1.0kHz output signal, generated using the Adafruit module connected to a Teensy 4.1 with “fly wires” – not ideal but good for basic testing. It can be seen in Figure 7 below that the all spurs up to the displayed range of 28.8kHz are at or below about ≈ 90 dBc, not at spec-sheet level, but this is very basic prototyping. Just fine.

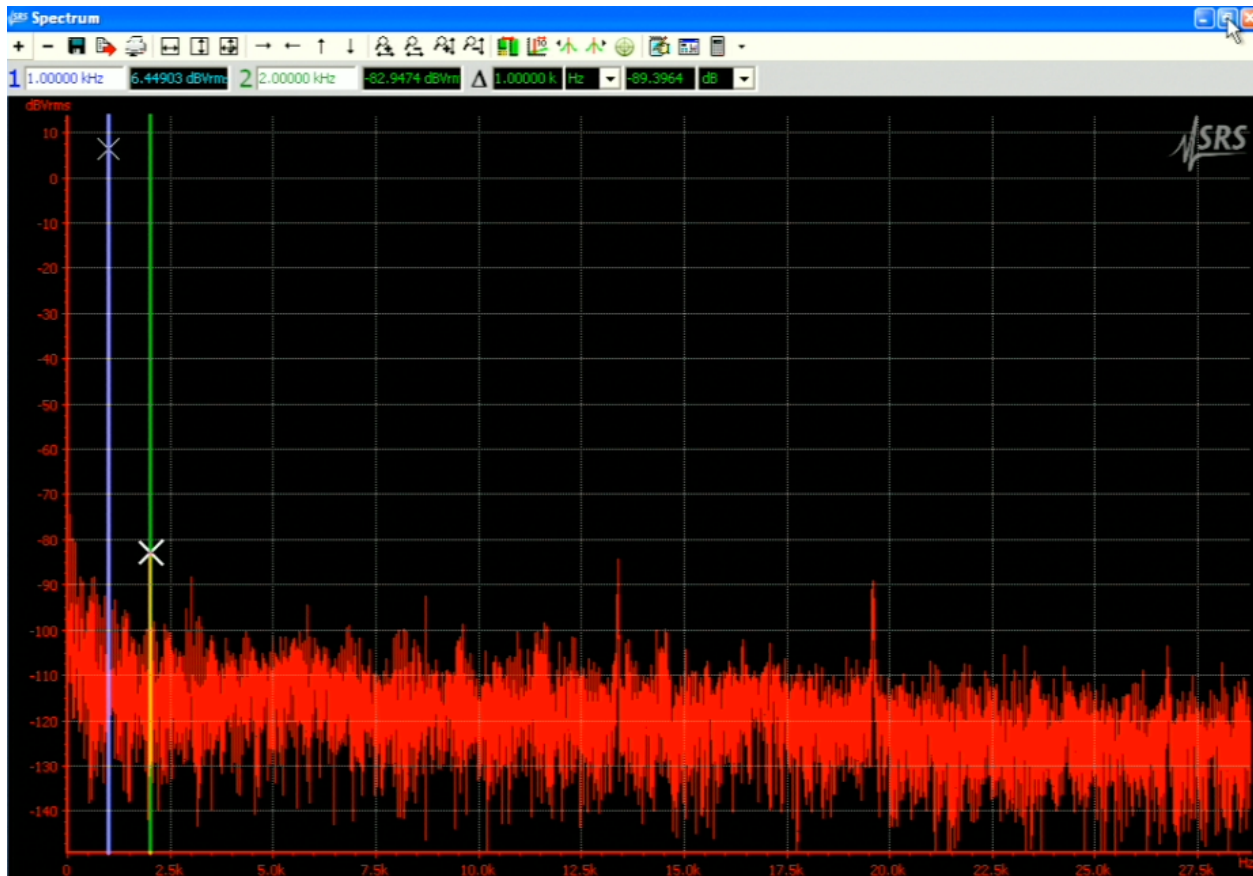


Figure 7: Lowest noise-floor view of 1 kHz output and accompanying spurs, with the SR1 in high-resolution mode, showing frequencies up to 28.8 kHz.

With the SR1 in “high-bandwidth” mode, displaying frequencies out to 199.75kHz, we can see more spurs, at about -89dBc (Figure 8 below). This is interesting because if one is generating audio (as are 99.99999% of the PCM5102A users, I suspect), harmonics this high won’t matter (you would not hear them even if your amp and speakers could reproduce them) – but they are real and would be unacceptable in some types of test equipment. Overall, this is very, very good performance for a few dollars!

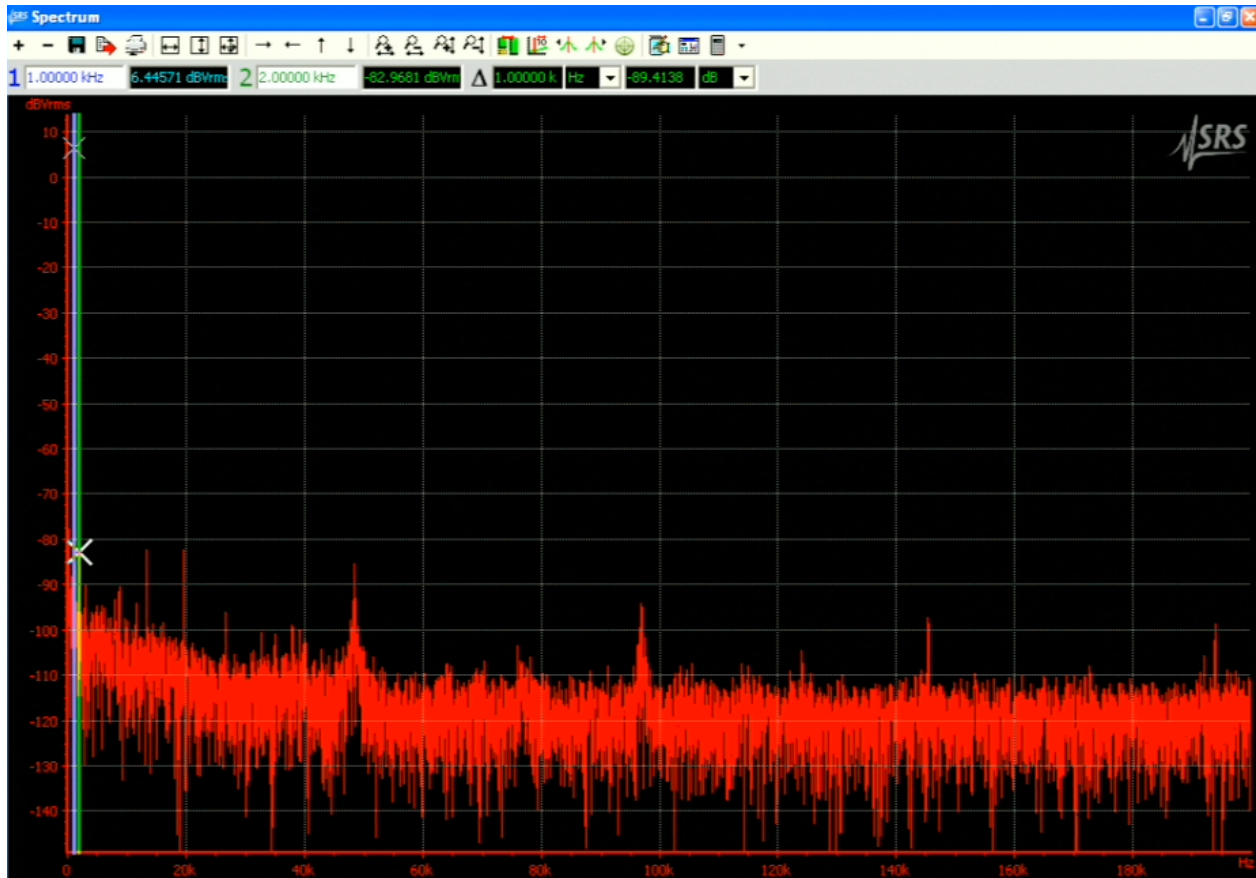


Figure 8: Higher frequency view of 1 kHz output and accompanying spurs, with the SR1 in high-bandwidth mode, showing frequencies up to 128kHz... now we can see the higher-frequency spurs, but they are all quite low.

Now we look at it for the target upper frequency of 60kHz (covers, for example, the 50 – 60kHz range used for biomedical impedance measurements, and conveniently allows synthesis of signals to annoy cats, dogs and bats). The results are shown in Figure 9 below, where the spectrum of a 60kHz output signal is shown. Note that there are spurs below the desired frequency – there is nothing we can do about those with filter (in-band spurs are the worst!). In-band, the worst spur is at -60dBc, which defines the best we can do for SFDR (spur-free dynamic range), but that is still comparable to decent bench-top function generators, but hopefully we can get here for just the price of a latte at Starbucks (depends on your choice of opamp), if you already have a Teensy 4.1 (spoiler – we can).



Figure 9: High-frequency view of 60 kHz output and accompanying spurs, with the SR1 in high-frequency mode, showing frequencies up to 128 kHz. Note many spurs above the band of interest (top image) and in-band (lower image). At this point the PCM5102A is way out of its designed-for frequency range of operation, but still doing a great job!

The closest out-of-band spur is at 84kHz at about -55dBc. That is something we can work with. A carefully designed low-pass filter could easily attenuate that spur down to -60dBc, hitting our target. Of course, in-band spurs might be worse than this for some frequencies, so a swept measurement is required. Using the test basic in Appendix 2, it was clear that the target -60dBc was achievable through the full range of near-DC to 60kHz. Good.

The Filter

We can easily design a filter using one dual-opamp per channel, to achieve our goal of about 60dBc SFDR. Now when one implements a filter at the required frequencies – assume cutoff above the maximum desired output frequency – the minimum opamp gain-bandwidth product (GBP) is roughly given by, for a maximum filter section Q (quality factor) and cutoff, f_c , by:

$$GPB > 20Qf_c$$

The “textbook fallacy” of all of this is that here we are working in “large signal” mode, versus using signals so small in amplitude, maybe 100mVPP, that the ugly nonlinearities of the chips don’t show themselves. Ultimately, the slew rate of the opamp limits the actual *large signal bandwidth*, based on the slew rate (rate of change of output voltage, usually V/μs) of:

$$FPBW \approx \frac{SR}{2\pi V_p} = \frac{SR}{\pi V_{pp}}$$

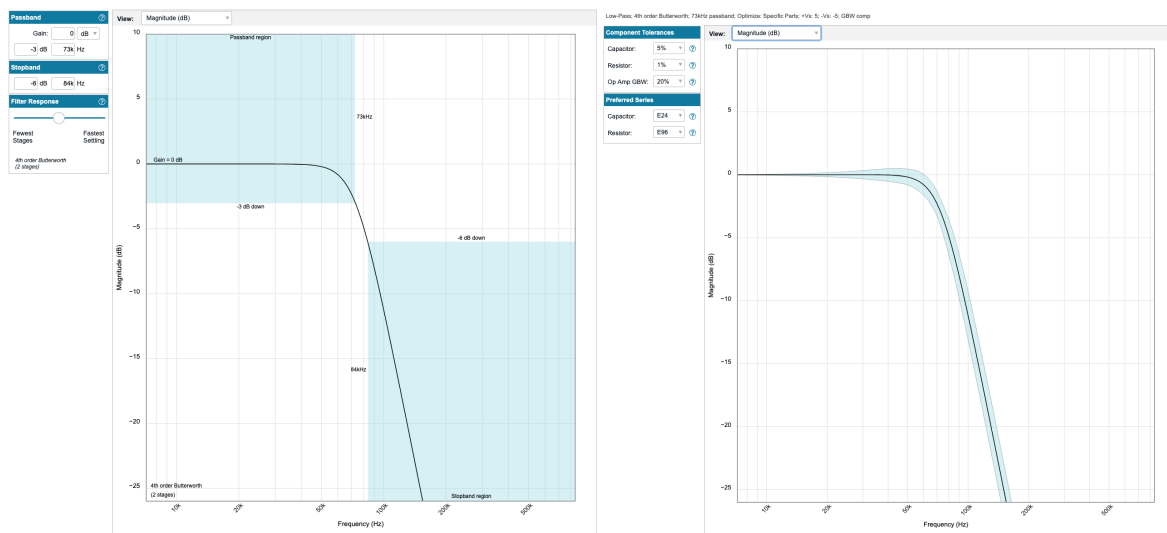
What this really means is that if you go with the GPB rule-of-thumb, and the signals get large, you will face a lot of distortion (slew-rate limitation tends to produce odd harmonics because it makes you sinewave look like a triangle wave, roughly, but even ones appear when slew rates differ – often!). So it means, net-net, that one needs to pick an opamp with not only the raw GBP, but also a slew-rate that will be well above spec. So assuming 70 kHz cutoff for now, the required SR for 6VPP (note that the scaling for SR is to multiply by 1E6) is roughly 1.32V/μs, which is a fast, but not insanely expensive opamp. For reference, the classic 741 (Appendix 4) has a slew rate of 0.5V/μs, which is only about 2.6 times too slow, but the designer, Dave Fullagar, released the design at Fairchild in 1968, so the 741 is still amazing!

The opamp we use for most things at VoidLoop is the AD8052, which has an SR of 145 V/μs, so is more than fast enough. The LM833 audio opamp and the NE5532 audio opamp were also quickly tested after the design was done, and showed comparable attenuation results, but worse second harmonic performance. In general, high slew-rate, low-distortion opamps are best for such applications, but chip makers know this and price accordingly.

Using the ADI “Analog Filter Wizard” tool (<https://tools.analog.com/en/filterwizard/>), and some iteration to choose real component values, the final filter was a fourth-order Butterworth with unity gain and a cutoff tuned to give the > 5dB attenuation at 84kHz. The approximation was a cutoff of 73kHz for an attenuation at 84kHz of 6 dB. The ideal response, Monte-Carlo spread with the specified component tolerances, and schematic are shown in

Figure 10 below, verifies that with 6dB designed attenuation, it should still work well given the chosen component tolerances.

Reminder – this is a tutorial. For a production design, one would allow for plenty of margin in production tolerances before declaring a spec like “SFDR $\geq$ 60dB.” The Monte-Carlo plot shows maybe ± 0.5 dB error at 60kHz, so all we can say here is that the SFDR will be “about” or “typically” 60dB, which is pretty darned good for a design that costs just a few dollars (Teensy 4.1 not included 😊). One might choose a lower-frequency cutoff and accept more attenuation at the maximum frequency of 60 kHz (here it is only about 0.82dB).



Low-Pass; 4th order Butterworth; 73kHz passband; Optimize: Specific Parts; +Vs: 5; -Vs: -5; GBW comp

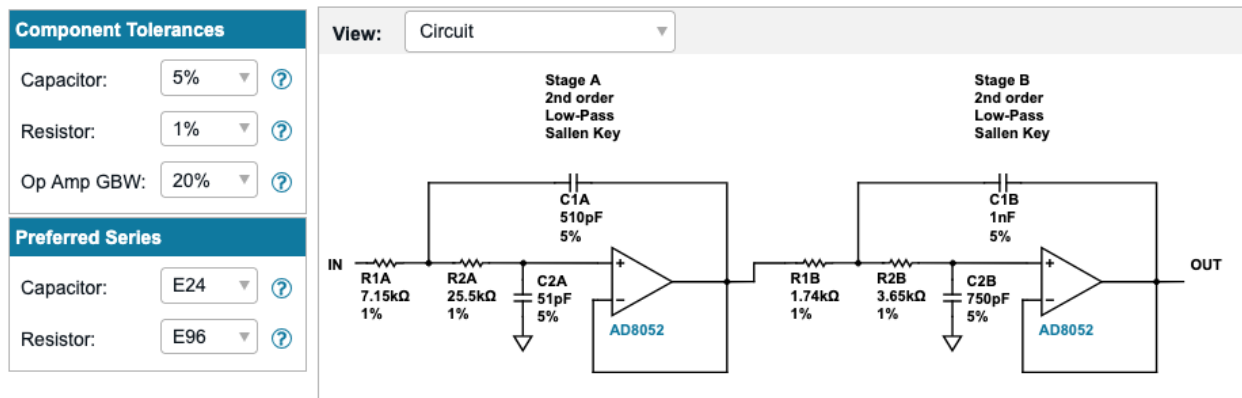


Figure 10: Filter design developed using the Analog Filter Wizard tool from Analog Devices. Note here that a fairly fast opamp (AD8052, 110MHz GBP, 145V/ μ s slew rate) was used to avoid distortion from the filter itself. Care is required when selecting other opamps, bearing in mind the full-power bandwidth (FPBW) which is a function of slew rate and signal amplitude – a very common mistake (explained above)! The plot at bottom shows the expected spread in frequency responses due to component variations (1% resistors and 5% capacitors).

The physical construction of the filter (not amenable to white “plug-boards” at these frequencies) is shown in Figure 11 below. Please note that an isolated, 5V-to-5V DC-to-DC converter was used to generate the -5V rail for the opamps (in this case the inexpensive B0505S-1W was used). USB +5V was used for the +5V rail, and each rail was decoupled at the AD8052 opamp with an 0.1 μ F capacitor to ground. In Figure 10, the DC-to-DC converter (black rectangle) is shown at lower right.

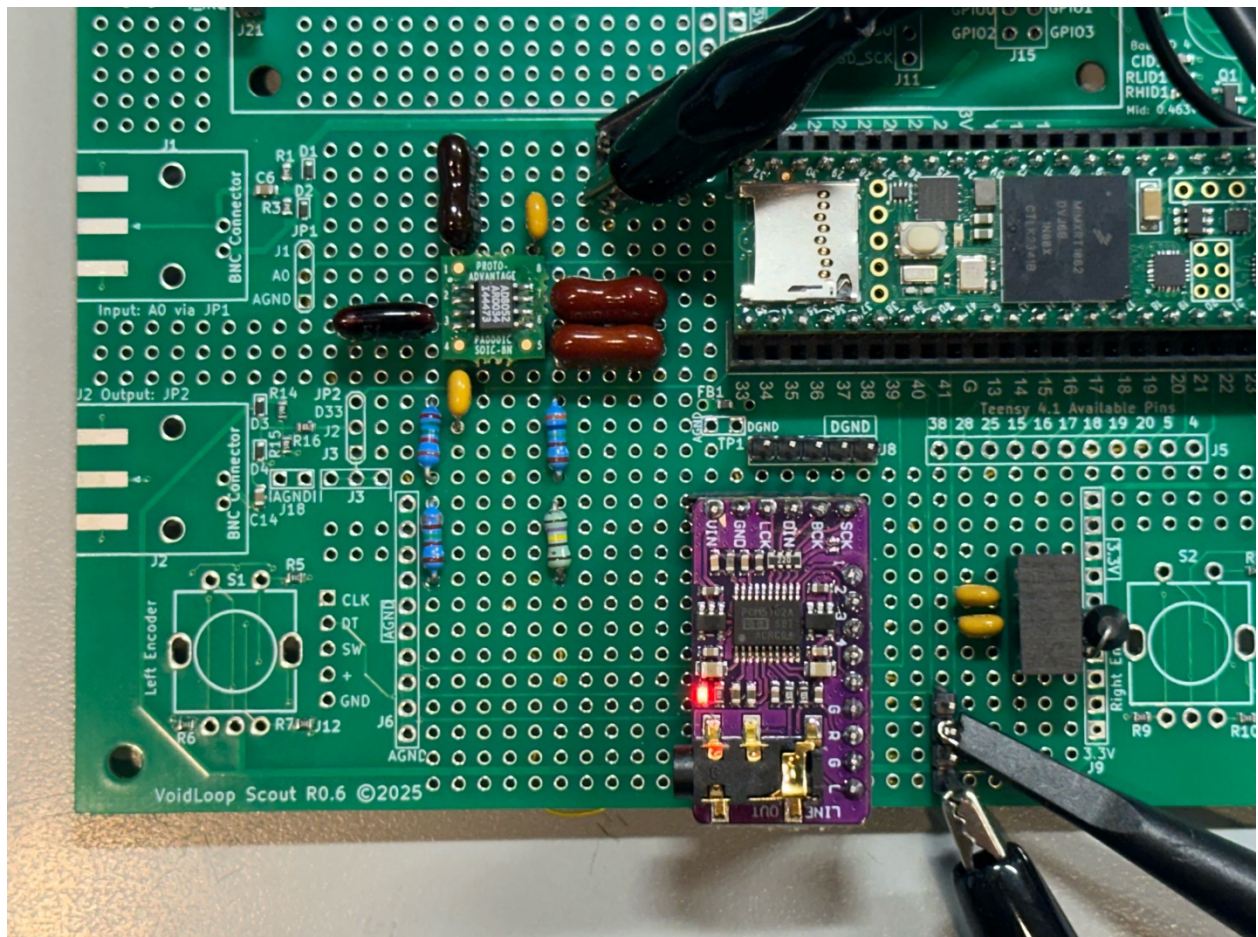


Figure 11: Physical construction of output filter, shown with Chinese PCM5102A module (lower center) and teensy 4.1 (upper right), built upon one of VoidLoop’s basic boards for prototyping. Only one filter channel was built, so that a direct comparison could be made to the unfiltered output (both channels were set to the same frequency and phase). The bottom-side includes a ground-plane (not shown), to which all ground points were soldered.

Next, the spur-suppression performance was validated using the SR1, and is shown in Figure 12 below (success!).

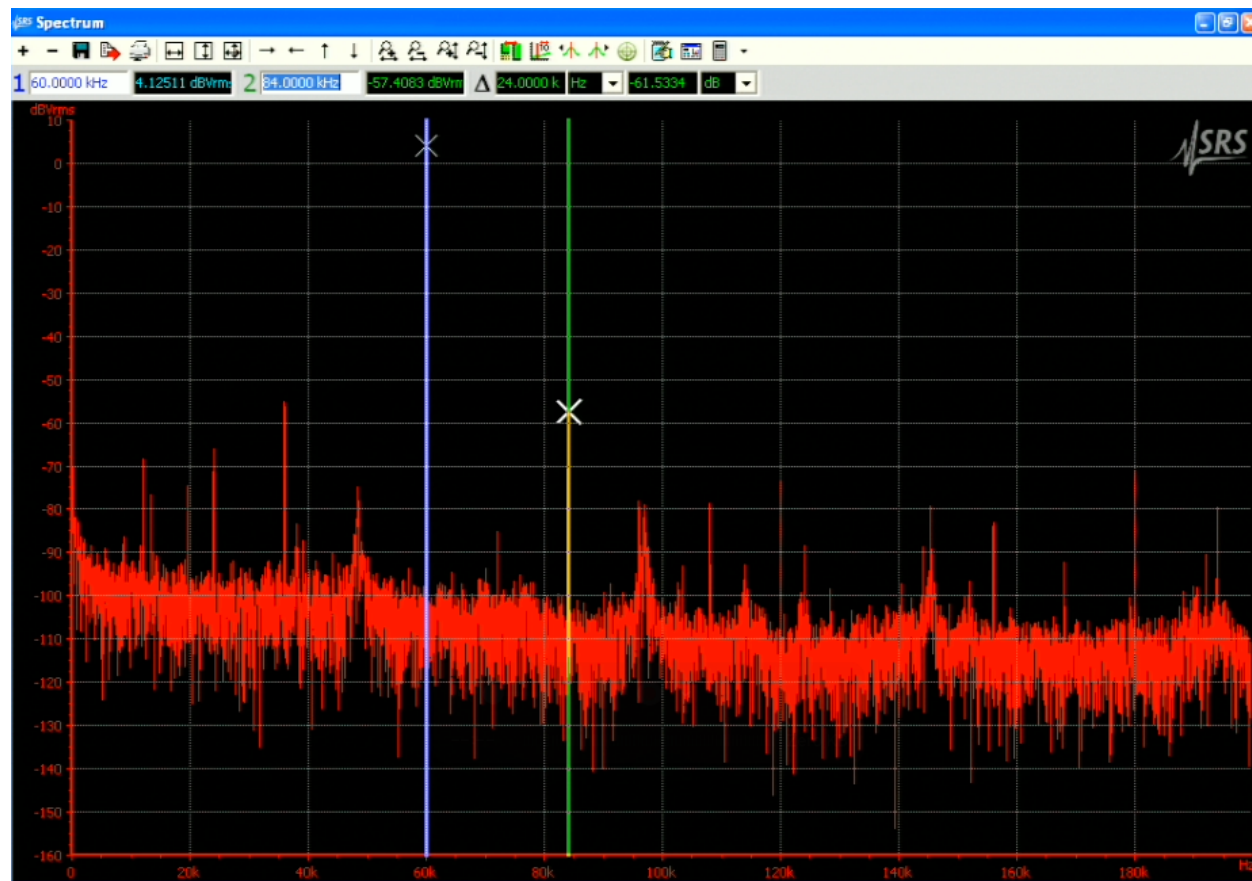


Figure 12: Validation of the filter design, showing 60kHz output frequency and 84kHz spur reduced to -61.5dBc. Good enough! There is nothing to be done about in-band spurs, so they define the best SFDR.

Swept Harmonic Testing

The next step was to use a **12-bit** oscilloscope (Rohde & Schwarz MXO-34 – we need 12-bit or higher scopes here to see the required FFT dynamic range, in theory about 74dB for 12-bits) with peak-hold on the FFT, to show the peak heights of spurs when the output frequency was swept from a few Hz to the 60kHz target. It is shown in Figure 12 below. The white traces capture the maximum spur levels over many frequency sweeps and help define the filter needed to suppress them. The lower trace in Figure 12 is the result with the filter described below. It works very well.

It is worth noting that the MXO-34 has exceptionally fast and good FFT's and includes the critical “peak-hold” feature seen below – few digital oscilloscopes offer that feature.

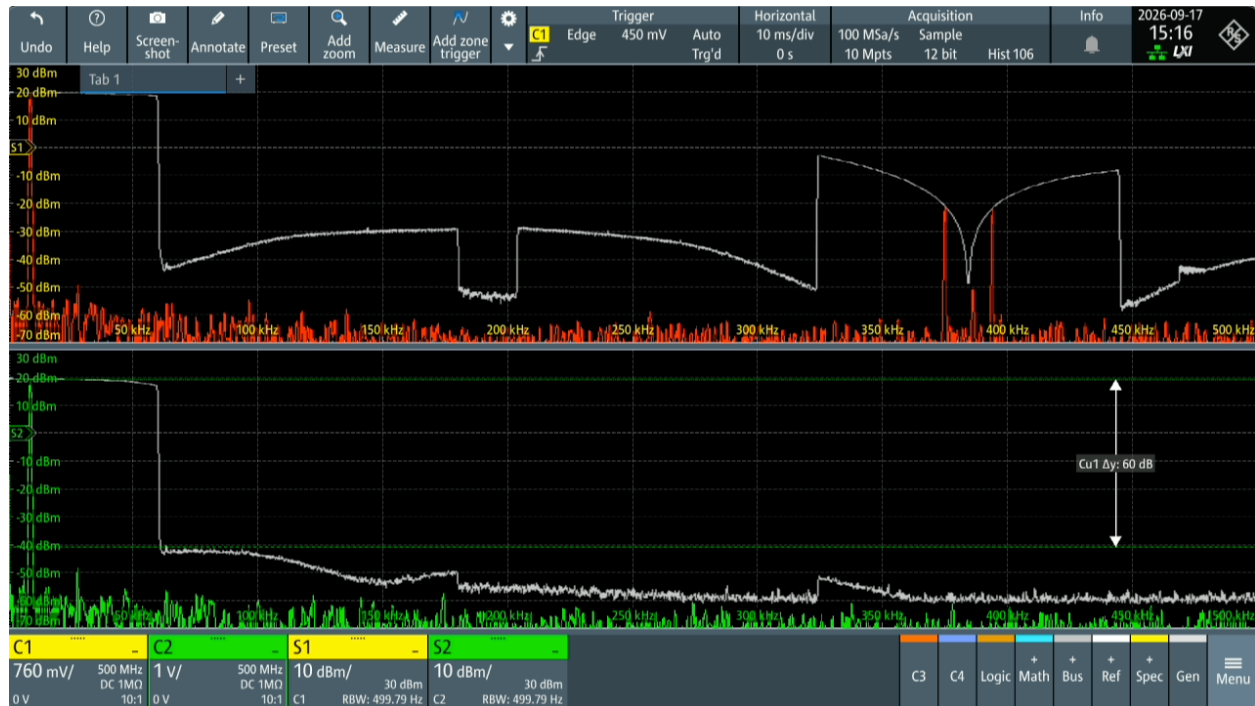


Figure 12: At top, unfiltered output of PCM5102A module, and at bottom, output through prototype filter (described below), showing SFDR $\approx$ 60dBc, using Rohde & Schwarz MXO-34 oscilloscope's FFT function with peak hold (white traces) to capture maximum spur amplitude over a sweep of output frequency from near DC to 60kHz. (Note the “Tab” rectangle at upper left cannot be moved and obscures signal information – this is an unfortunate issue with this type of oscilloscope.)

Improved Code: Adding “No-Hardware” Amplitude Control, Modulation and Noise

Despite sending 32-bit words to the DAC, it is not capable of anywhere near that kind of resolution. But, we can do on-the-fly amplitude control using only code, by scaling the data flowing to the DAC for each channel by a static (gain) or modulation signal. So we don't have to add any form of analog gain control.

The catch? Well, if one added a potentiometer, for example, for manual gain control. The signal and noise would both be attenuated together, for a more-or-less constant SNR or SFDR. If one digitally “attenuates” the DAC data, the noise floor will remain the same, and the signal amplitude will be reduced, so the SNR and SFDR will degrade. Still, this approach comes at zero cost, and after all, this is a tutorial!

Into a high-impedance load, the PCM5102A produces almost exactly 6.0VPP at full-scale. The final prototype used a 49.9Ω series resistor at the output of the final opamp stage to properly match a typical 50Ω load as used in test equipment (into high-Z loads it will still be 6.0VPP, but into 50Ω will be 3.0VPP). In any case, it is easy to “calibrate” the code for either use case. As for the DDS amplitude levels (“on-the-fly” versus fixed LUT), the amplitude control values for each channel can be updated as often as once per sample, allowing for static or dynamic amplitude control (e.g., “AM”).

Figure 13 below shows an example 10kHz signal with AM modulation at a depth of 100%, with the expected frequency spectrum. Figure 14 below shows the random noise mode, which provides a noise spectrum that is fairly flat (to 1 dB) out to 50kHz and 3dB down at approximately 62kHz. Figure 15 below shows (cool) the same AM modulation in Figure 13 applied to the noise. Figure 16 below shows the broadband (to 500kHz) performance of the unmodulated noise.



Figure 13: Example of 10kHz carrier, 1kHz AM frequency and 100% modulation depth using the code in Appendix 3. The top trace is the AM spectrum, the bottom is the time-domain signal. Note that no visible spurs appear out to 100kHz.



Figure 14: Random noise output, unmodulated (bandwidth is shaped by the output filter). The white trace is the average of the spectrum, which is very flat out to the target 60kHz bandwidth (detailed measurements showed that the noise is down 1dB by 50 kHz).



Figure 15: Example of 100% AM modulated random noise, with the same modulation parameters as in Figure 13.

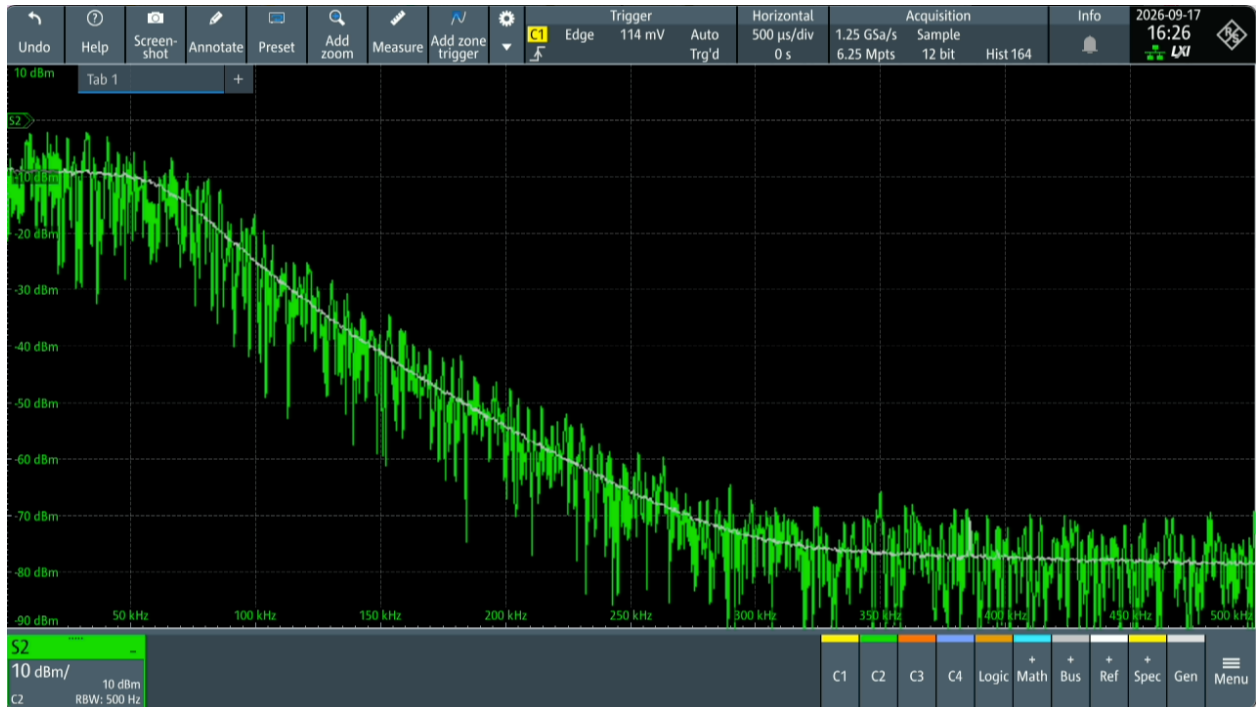


Figure 16: Wide-band frequency spectrum of random noise, out to 500kHz. The white trace is the average of the spectrum.

Bottom Lines

It works! We have shown that it is possible to push the PCM5102A well beyond audio (to 60kHz), generate clean waveforms down to mHz (due to the chip's DC-coupled output) and maintain a respectable SFDR of about 60dB at 60kHz, decreasing to ≈ 90 dB in the audio range.

Notably and nerdily, we have also demonstrated “on-the-fly” DDS without a pre-computed look-up-table, as well as “on-the-fly,” software-controlled amplitude modulation (including dynamic modulation – to generate AM signals). No pre-computed LUTs, but floating point until the moment of casting as 32-bit numbers to send to the DACs, and the phase accumulator is the whole phase accumulator, not some truncated bit used to address the LUT. This is the power of a 600MIPS/450MFlops processor influencing standard design practice. The chip itself (i.MX RT1062) is \$6 at 1kU. New world. Cool.

Lastly, generation of pseudo-random noise (maybe notice the “Entropy.h” library) within the bandwidth of the output filters was demonstrated.

VoidLoop will soon have a fully-assembled, two-channel version of the demo circuit, which can drive 50 Ω loads to about 3.0VPP or high-Z loads to 6.0VPP and later a board with a color touchscreen and a UI for control. It will not be expensive. We are not out to make money, but are out to empower great engineers, hobbyists and scientists.

The concepts and code discussed here work with off-the-shelf modules and parts if you want to do a DIY build. Just remember that “on-the-fly” DDS computation requires a powerful CPU – our code is only tested with the Teensy 4.1.

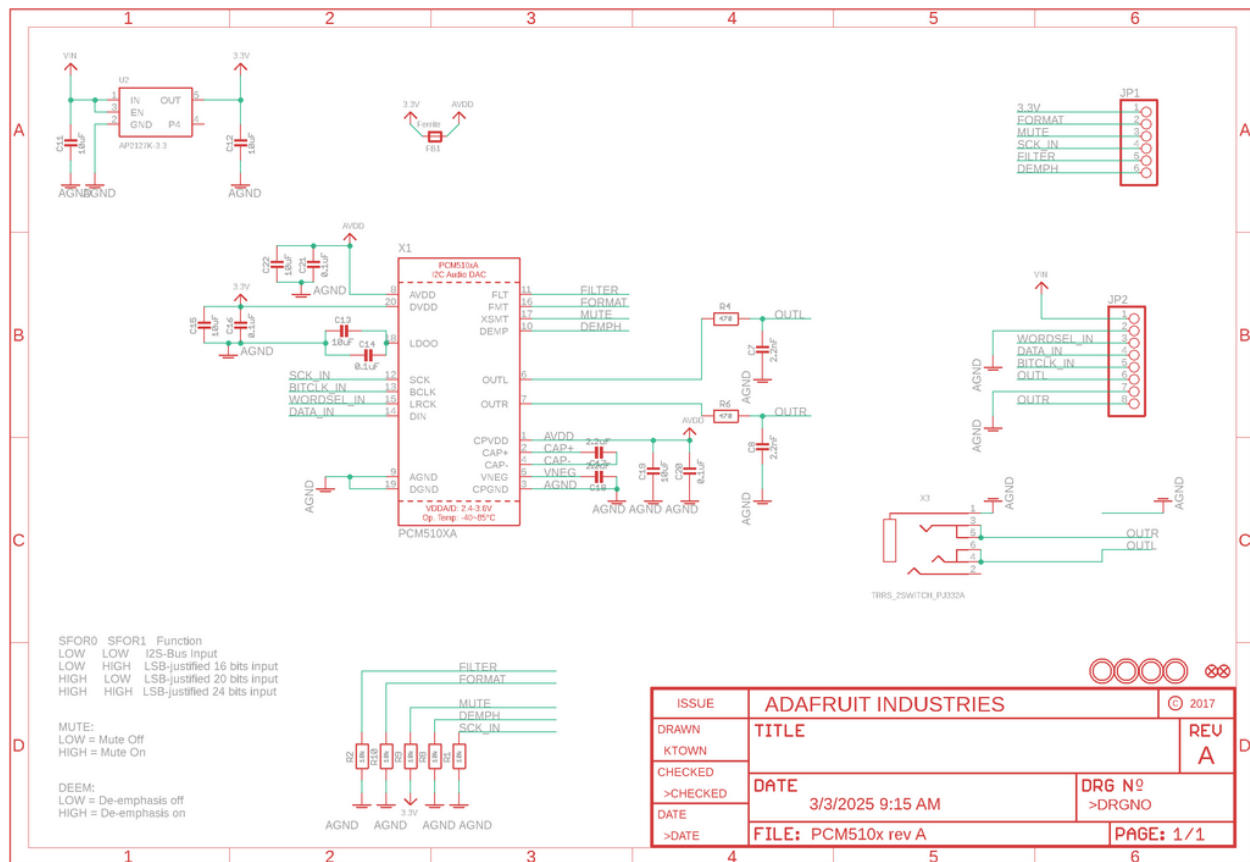
Appendix 3 shows the modified code for independent DDS frequencies per channel and amplitude control via software scaling. **This is the code to use for your own projects if you have a good handle on c++.**

Disclosures:

VoidLoop does not shill for T&M or chip companies. Greg funds it and pays cash for test equipment except in the rare instances where they are unconditionally gifted. John Willison, founder of SRS, gifted me (thank you so much) the SR1, which powered two of my PhD students and myself through many, many circuits – I am extremely grateful. Otherwise, cash, so the opinions expressed are genuine.

Appendix 1: Module Schematics and Connections to Teensy 4.1

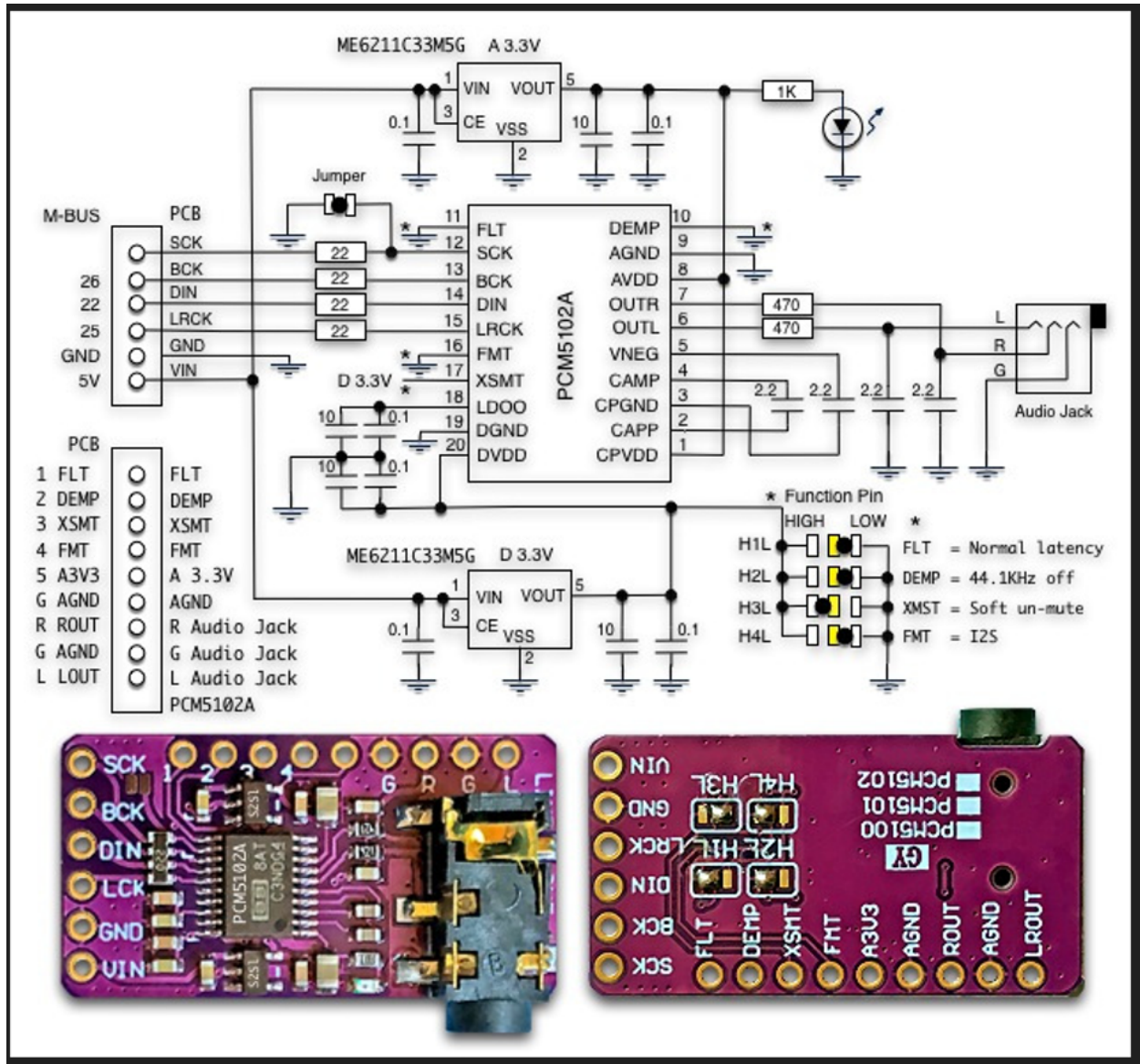
Adafruit Module Schematic:



Adafruit PCM5102A Module Connections:

- * Hardware (Adafruit PCM5102A module):
- * - Teensy pin 36 → BCK (SAI1 TX bit clock)
- * - Teensy pin 37 → WSEL (SAI1 TX frame sync / LRCLK)
- * - Teensy pin 7 → DIN (SAI1 TX data)
- * - N/C → SCK (grounding SCK selects the DAC's internal PLL, so no master clock wire is needed)
- * - 5V → VIN, GND → GND

Chinese PCM5102A Module Schematic:



Chinese PCM5102A Module Connections:

* Hardware (Chinese PCM5102A module):

- * - Teensy pin 36 -> BCK (SAI1 TX bit clock)
- * - Teensy pin 37 -> LCK (SAI1 TX frame sync / LRCLK)
- * - Teensy pin 7 -> DIN (SAI1 TX data)
- * - GND -> SCK (grounding SCK selects the DAC's internal PLL, so no master clock wire is needed)
- * - 5V -> VIN, GND -> GND

Appendix 2: Basic Test Code (Easier to Understand)

```
/**
 * Basecamp_PCM5102_Sine.ino
 * @version 1.0.0
 * @date 08-28-26
 *
 * VoidLoopEngine is brought to you by VoidLoop. See www.voidloop.com for details.
 * Stereo DDS sine generator driving a PCM5102A I2S DAC from SAI1.
 *
 * Board: Basecamp (Teensy 4.1). Any Teensy 4.x wired the same way works.
 *
 * A 32-bit phase accumulator advances by a fixed increment once per audio
 * frame; the top 14 bits of the accumulator select a point on one sine
 * cycle. Output frequency = (increment / 2^32) x sample rate, so retuning
 * only changes the step size – the phase, and the output, never jump.
 *
 * The left channel carries sin(phase) and the right sin(phase + 90 deg).
 * Looped into an X/Y display, that quadrature pair draws a circle.
 *
 * This demonstrates:
 * – Register-level SAI1 (I2S) master transmit on the Teensy 4.1
 * – Audio PLL (PLL4) configuration for an arbitrary sample rate
 * – Direct digital synthesis with a phase accumulator
 *
 * Hardware (PCM5102A module):
 * – Teensy pin 36 -> BCK (SAI1 TX bit clock)
 * – Teensy pin 37 -> LCK (SAI1 TX frame sync / LRCLK)
 * – Teensy pin 7 -> DIN (SAI1 TX data)
 * – GND -> SCK (grounding SCK selects the DAC's internal PLL,
 * so no master clock wire is needed)
 * – 5V -> VIN, GND -> GND
 * – Module solder jumpers (back of the purple boards): 1(FLT)=L, 2(DEMP)=L,
 * 3(XSMT)=H, 4(FMT)=L. XSMT=H unmutes the output; FMT=L selects I2S.
 *
 * The output is line level, about 2.1 Vrms at full scale and centered on
 * ground. Start at low volume on headphones or a powered speaker. Do not
 * wire it straight to a Teensy ADC pin – it swings below ground.
 *
 * Sketch outline:
 * – setup() programs the audio PLL and SAI1, muxes the three pins, and
 * enables a FIFO interrupt that pushes one stereo frame per sample.
 * – loop() prints a heartbeat, or steps the frequency when SWEEP_DEMO is on.
 *
 * Things to try:
 * – Change toneHz and listen for the pitch change.
```

```

* - Set SWEEP_DEMO to 1 for a repeating 20 Hz - 50 kHz sweep.
* - Put left and right on an X/Y scope display and watch the circle.
*
* Based on PCM5102 bring-up code by G. Kovacs and M. Flamer (2023).
*/

#include <math.h>

// ---- Knobs -----

// Output frequency in Hz. Values below ~15 kHz are easy to hear.
static const float toneHz = 60000.0f;

// Set to 1 to sweep sweepMinHz..sweepMaxHz instead of holding toneHz.
#define SWEEP_DEMO 1
static const float sweepMinHz = 20.0f;
static const float sweepMaxHz = 60000.0f;
static const float sweepStepHz = 100.0f;
static const int sweepStepMs = 50; // at least one full cycle of the lowest step

// Frames per second on the I2S bus. The SAI interrupt fires once per frame,
// so raising this raises interrupt load one-for-one.
static const int sampleRate = 384000;

// ---- DDS state -----

static volatile uint32_t ddsAccum = 0; // current phase, full 32-bit circle
static volatile uint32_t ddsIncrement = 0; // phase step per frame

// The top 14 accumulator bits address one sine cycle, as if reading a
// 16384-entry lookup table. The sine is computed on the fly here, but
// keeping the 14-bit step means the harmonic content matches what a real
// 16 K table (the eventual non-blocking version) will produce.
static const uint32_t lutBits = 14;
static const uint32_t lutSize = 1u << lutBits;
static const uint32_t ddsShift = 32 - lutBits;

// I2S data words are signed 32-bit two's complement; the int32_t casts in
// the interrupt handler produce exactly that.
static const int32_t fullScale = 2147483647;

static void setFrequency(float hz) {
    ddsIncrement = (uint32_t)((hz / (double)sampleRate) * 4294967296.0 + 0.5);
}

// Runs every time the SAI1 TX FIFO drains to its watermark - once per audio
// frame in steady state. Pushes one stereo frame, then advances the phase.
static void sai1_isr(void) {

```

```

uint32_t idx = ddsAccum >> ddsShift; // 0 .. lutSize-1
double phase = 2.0 * M_PI * ((double)idx / (lutSize - 1));
I2S1_TDR0 = (int32_t)(fullScale * sin(phase)); // left
I2S1_TDR0 = (int32_t)(fullScale * sin(phase + 0.5 * M_PI)); // right
ddsAccum += ddsIncrement;
}

// Sets PLL4 (the audio PLL) to 24 MHz x (nfact + nmult/ndiv). A PLL that is
// already enabled is left untouched.
static void set_audioClock(int nfact, int32_t nmult, uint32_t ndiv) {
    if (CCM_ANALOG_PLL_AUDIO & CCM_ANALOG_PLL_AUDIO_ENABLE) return;

    CCM_ANALOG_PLL_AUDIO = CCM_ANALOG_PLL_AUDIO_BYPASS | CCM_ANALOG_PLL_AUDIO_ENABLE
        | CCM_ANALOG_PLL_AUDIO_POST_DIV_SELECT(2)
        | CCM_ANALOG_PLL_AUDIO_DIV_SELECT(nfact);
    CCM_ANALOG_PLL_AUDIO_NUM = nmult & CCM_ANALOG_PLL_AUDIO_NUM_MASK;
    CCM_ANALOG_PLL_AUDIO_DENOM = ndiv & CCM_ANALOG_PLL_AUDIO_DENOM_MASK;

    CCM_ANALOG_PLL_AUDIO &= ~CCM_ANALOG_PLL_AUDIO_POWERDOWN;
    while (!(CCM_ANALOG_PLL_AUDIO & CCM_ANALOG_PLL_AUDIO_LOCK)) {}

    CCM_ANALOG_MISC2 &= ~(CCM_ANALOG_MISC2_DIV_MSB | CCM_ANALOG_MISC2_DIV_LSB);

    CCM_ANALOG_PLL_AUDIO &= ~CCM_ANALOG_PLL_AUDIO_BYPASS;
}

// Clocks SAI1 from PLL4 and configures it as I2S master: two 32-bit words
// per frame, MSB first, data one bit clock after each frame-sync edge.
static void config_i2s(int fs) {
    CCM_CCGR5 |= CCM_CCGR5_SAI1(CCM_CCGR_ON);

    // PLL4 must run between 648 MHz (24 MHz x 27) and 1296 MHz (x 54); the
    // n1/n2 dividers scale the target so the multiplier C lands in range.
    int n1 = 4;
    int n2 = 1 + (24000000 * 27) / (fs * 256 * n1);
    double C = ((double)fs * 256 * n1 * n2) / 24000000;
    int c0 = C;
    int c2 = 10000;
    int c1 = C * c2 - (c0 * c2);
    set_audioClock(c0, c1, c2);

    CCM_CSCMR1 = (CCM_CSCMR1 & ~(CCM_CSCMR1_SAI1_CLK_SEL_MASK))
        | CCM_CSCMR1_SAI1_CLK_SEL(2); // 2 = PLL4
    CCM_CS1CDR = (CCM_CS1CDR & ~(CCM_CS1CDR_SAI1_CLK_PRED_MASK |
        CCM_CS1CDR_SAI1_CLK_PODF_MASK))
        | CCM_CS1CDR_SAI1_CLK_PRED(n1 - 1)
        | CCM_CS1CDR_SAI1_CLK_PODF(n2 - 1);
}

```

```

// MCLK1, selected as the bit-clock source by MSEL below, is the SAI1
// clock root. The MCLK pad itself is never muxed out; the DAC recovers
// its clock from BCK.
IOMUXC_GPR_GPR1 = (IOMUXC_GPR_GPR1 & ~(IOMUXC_GPR_GPR1_SAI1_MCLK1_SEL_MASK))
                  | (IOMUXC_GPR_GPR1_SAI1_MCLK1_DIR |
IOMUXC_GPR_GPR1_SAI1_MCLK1_SEL(0));

// These SAI1 TX pins are free on Basecamp; on Ascent the same pins belong
// to the parallel sampler.
CORE_PIN36_CONFIG = 3; // pad B1_02 -> SAI1_TX_BCLK (BCK)
CORE_PIN37_CONFIG = 3; // pad B1_03 -> SAI1_TX_SYNC (LCK)
CORE_PIN7_CONFIG  = 3; // pad B1_01 -> SAI1_TX_DATA0 (DIN)

// Transmitter as asynchronous master: it generates BCK and LRCLK itself.
// The FIFO watermark of 8 fires the refill interrupt with room to spare.
I2S1_TMR = 0;
I2S1_TCR1 = I2S_TCR1_RFW(8);
I2S1_TCR2 = I2S_TCR2_SYNC(0) | I2S_TCR2_BCP
            | I2S_TCR2_BCD | I2S_TCR2_DIV(1) | I2S_TCR2_MSEL(1);
I2S1_TCR3 = I2S_TCR3_TCE;
I2S1_TCR4 = I2S_TCR4_FRSZ(2 - 1) | I2S_TCR4_SYWD(32 - 1) | I2S_TCR4_MF
            | I2S_TCR4_FSD | I2S_TCR4_FSE | I2S_TCR4_FSP | I2S_TCR4_FCONT;
I2S1_TCR5 = I2S_TCR5_WNW(32 - 1) | I2S_TCR5_W0W(32 - 1) | I2S_TCR5_FBT(32 - 1);

// Prime the FIFO with silence so the transmitter has data the moment it
// starts.
for (int i = 0; i < 32; i++) {
    I2S1_TDR0 = 0;
}

attachInterruptVector(IRQ_SAI1, sai1_isr);
NVIC_ENABLE_IRQ(IRQ_SAI1);
I2S1_TCSR = I2S_TCSR_TE | I2S_TCSR_BCE | I2S_TCSR_FRIE;
}

void setup() {
    Serial.begin(115200);
    setFrequency(SWEEP_DEMO ? sweepMinHz : toneHz);
    config_i2s(sampleRate);
}

void loop() {
    #if SWEEP_DEMO
        static float hz = sweepMinHz;
        hz += sweepStepHz;
        if (hz > sweepMaxHz) hz = sweepMinHz;
        setFrequency(hz);
        delay(sweepStepMs);
    #endif
}

```

```

#else
    // Heartbeat, so a silent DAC is distinguishable from a board that never
    // booted or never took the flash.
    static uint32_t lastMs = 0;
    if (millis() - lastMs >= 2000) {
        lastMs = millis();
        Serial.printf("PCM5102 DDS: %.1f Hz at %d frames/s, up %lu s\n",
                      (double)toneHz, sampleRate, millis() / 1000);
    }
#endif
}

```

Appendix 3: Enhanced Code with Independent Channel Frequencies and Amplitude Control (“Full-Featured”)

```
/**
 * PCM5102A_VoidLoop_R0_2.ino
 * @version 1.2.0
 * @date 08-28-26, 9/16/26, 9/17/26
 *
 * VoidLoopEngine is brought to you by VoidLoop. See www.voidloop.com for details.
 * Two-channel DDS signal generator driving a PCM5102A I2S DAC from SAI1.
 *
 * Board: Teensy 4.1 only (Basecamp). Pins 36 and 37 carry the SAI1 transmit
 * clocks on the 4.1; on the Teensy 4.0 the same pin numbers are SD card pads.
 *
 * Each channel (right and left) is set up independently, at compile time, as
 * one of three sources:
 *   SOURCE_FIXED   one sine frequency, with an optional starting phase
 *   SOURCE_SWEEP   a stepped sine sweep from a start to a stop frequency,
 *                   rising or falling
 *   SOURCE_NOISE   a new random number every sample, no DDS
 * A fixed sine or a noise source can also be amplitude modulated (AM) by a
 * second, slower sine.
 *
 * Sine frequencies are limited to 60 kHz. Noise is not band-limited
 * by the sketch. At 384 kSPS, the generated noise spans frequencies
 * up to 192 kHz. The DAC's filtering and any external output filter
 * shape the analog noise spectrum.
 *
 * A 32-bit phase accumulator represents one complete sine cycle.
 * Each sample advances the phase by a fixed increment, and sin()
 * calculates the sample value from that phase.
 *
 * Output frequency = (phase increment / 2^32) x sample rate.
 *
 * Using only the upper phase bits to index a lookup table introduces
 * phase truncation, which can produce spurious tones. This sketch
 * uses all 32 phase bits in the sine calculation, avoiding that
 * source of error.
 *
 * Changing frequency changes the phase increment without resetting
 * the accumulator, so the waveform's phase remains continuous.
 *
 * This demonstrates:
 * - Register-level SAI1 (I2S) master transmit on the Teensy 4.1
 * - Audio PLL (PLL4) configuration for an arbitrary sample rate
 * - Direct digital synthesis with a phase accumulator, computed on the fly
```

```

* - Sweeps, white noise and amplitude modulation done entirely in software
*
* Hardware, purple PCM5102A breakout:
* - Teensy pin 36 -> BCK   (SAI1 TX bit clock)
* - Teensy pin 37 -> LCK   (SAI1 TX frame sync / LRCLK)
* - Teensy pin 7  -> DIN   (SAI1 TX data)
* - GND           -> SCK   (grounding SCK selects the DAC's internal PLL,
*                           so no master clock wire is needed)
* - 5V -> VIN, GND -> GND
* - Solder jumpers on the back: 1(FLT)=L, 2(DEMP)=L, 3(XSMT)=H, 4(FMT)=L.
*   XSMT=H unmutes the output; FMT=L selects I2S.
*
* Hardware, Adafruit PCM5102A module:
* - Teensy pin 36 -> BCK, pin 37 -> WSEL, pin 7 -> DIN
* - SCK (MCK pad) left unconnected; the board's default selects the
*   internal PLL. There are no solder jumpers to set.
* - 5V -> VIN, GND -> GND
*
* The output is about 6 Vpp (2.1 Vrms) at full scale and centered on ground.
* It is a line-level output for loads of 1 kilohm or more: a scope, the line
* input of a powered speaker, or a headphone amplifier. It cannot drive
* headphones directly. Do not wire it straight to a Teensy ADC pin - it
* swings below ground.
*
* Sketch outline:
* - The "Channel setup" section below is the only part you need to edit.
* - setup() seeds the noise generators, loads the starting frequencies, then
*   programs the audio PLL and SAI1 and enables the SAI1 interrupt.
* - The interrupt computes one left and one right sample per audio frame.
* - loop() steps any sweeps and prints a status line every two seconds.
*
* Things to try:
* - For a circle on an X/Y scope display, set both channels to SOURCE_FIXED
*   at 1000 Hz with equal amplitudes and modulationEnabled = false.
*   Set startPhaseDegrees to 0 on one channel and 90 on the other.
*   Connect the outputs to the scope's X and Y inputs, select X/Y mode,
*   and use equal volts-per-division settings on both axes.
* - Swap a sweep's start and stop frequencies and watch it fall instead.
* - Set modulationDepth to 1.0 and look at the envelope on a scope.
* - Halve a channel's amplitude and compare the spurs on a spectrum analyzer.
*
* Based on PCM5102 bring-up code by G. Kovacs and M. Flamer (2023), extended
* by Z. Magee and G. Kovacs (2026).
*
* set_audioClock() and the clock setup in config_i2s() are adapted from the
* PJRC Teensy Audio Library (output_i2s.cpp, utility/imxrt_hw.cpp):
*
* Copyright (c) 2014, Paul Stoffregen, paul@pjrc.com

```

```

*
* Development of this audio library was funded by PJRC.COM, LLC by sales
* of Teensy and Audio Adaptor boards. Please support PJRC's efforts to
* develop open source software by purchasing Teensy or other PJRC products.
*
* Permission is hereby granted, free of charge, to any person obtaining a
* copy of this software and associated documentation files (the
* "Software"), to deal in the Software without restriction, including
* without limitation the rights to use, copy, modify, merge, publish,
* distribute, sublicense, and/or sell copies of the Software, and to
* permit persons to whom the Software is furnished to do so, subject to
* the following conditions:
*
* The above copyright notice, development funding notice, and this
* permission notice shall be included in all copies or substantial
* portions of the Software.
*
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
* OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
* MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
* IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
* CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
* TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
* SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
*/

#include <math.h>
#include <Entropy.h>

#if !defined(ARDUINO_TEENSY41)
#error "This sketch needs a Teensy 4.1: pins 36 and 37 are SAI1 clock pins only on that board."
#endif

enum SourceType { SOURCE_FIXED, SOURCE_SWEEP, SOURCE_NOISE };

// ChannelConfig groups the settings for one output channel.
// rightChannel and leftChannel below each supply a set of these values.
// Edit those values to choose the signals; this declaration defines
// the settings available to both channels.
struct ChannelConfig {
  SourceType sourceType;
  // Output level. 1.0 is full scale, about 6 Vpp. A negative value inverts
  // the waveform.
  double amplitude;
  // SOURCE_FIXED only.
  double frequencyHz;
  double startPhaseDegrees;

```

```

// SOURCE_SWEEP only. The frequency starts at sweepStartFrequencyHz and
// moves toward sweepStopFrequencyHz by sweepStepFrequencyHz. Each frequency
// is held for at least sweepHoldMilliseconds. If start is above stop the
// sweep falls. After the stop frequency has played, the sweep begins again
// from start.
double    sweepStartFrequencyHz;
double    sweepStopFrequencyHz;
double    sweepStepFrequencyHz;
uint32_t  sweepHoldMilliseconds;
// Amplitude modulation, for SOURCE_FIXED and SOURCE_NOISE only.
// modulationDepth ranges from 0.0 (no modulation) to 1.0 (100 %).
// amplitude sets the maximum envelope level. The carrier amplitude is
// amplitude / (1.0 + modulationDepth), so increasing depth reduces the
// carrier level. At 100 % depth the envelope reaches zero but does not
// reverse the carrier's polarity.
bool      modulationEnabled;
double    modulationFrequencyHz;
double    modulationDepth;
};

// ChannelState holds the working values that change as a channel runs.
// setup() initializes these values from that channel's configuration.
// The interrupt advances the waveform phase or noise sequence;
// loop() updates the sweep frequency and timing.
// Each channel has its own state. No user settings are entered here.
struct ChannelState {
    volatile uint32_t phaseAccumulator;    // carrier phase, 2^32 counts per turn
    volatile uint32_t phaseIncrement;      // carrier phase step per sample
    volatile uint32_t modulationPhaseAccumulator; // phase of the modulating sine
    volatile uint32_t modulationPhaseIncrement; // its phase step per sample
    volatile uint32_t noiseState;          // last random number drawn
    volatile float    amplitude;           // output scale initialized from
ChannelConfig
    double    sweepFrequencyHz;            // frequency the sweep is playing now
    uint32_t  sweepHoldStartedMs;          // millis() when that frequency began
};

// ---- Channel setup -----
// Edit rightChannel and leftChannel independently, then compile and upload.
// constexpr makes these settings fixed for that build.
//
// For each channel:
// - Choose sourceType and amplitude. An amplitude of 0.0 silences the channel.
// - SOURCE_FIXED: set frequencyHz and startPhaseDegrees.
// - SOURCE_SWEEP: set the start and stop frequencies, a positive frequency
//   step, and the hold time. Start above stop gives a falling sweep.
//   Set modulationEnabled to false.
// - SOURCE_NOISE: no frequency or starting phase is needed.

```

```

// - For AM on a fixed sine or noise, set modulationEnabled to true,
//   then choose modulationFrequencyHz and modulationDepth.
//
// Leave unused fields in place; their values are ignored.
// Invalid combinations stop compilation with an explanatory message.

// Example: for an unmodulated 1 kHz sine, set sourceType to SOURCE_FIXED,
// frequencyHz to 1000.0, startPhaseDegrees to 0.0, amplitude to 0.5,
// and modulationEnabled to false. Leave the sweep settings unchanged.

constexpr ChannelConfig rightChannel = {
    // SOURCE_FIXED, SOURCE_SWEEP or SOURCE_NOISE.
    .sourceType          = SOURCE_FIXED,
    // Used by every source. 1.0 is full scale; negative inverts.
    .amplitude           = 1.0,
    // Used by SOURCE_FIXED only.
    .frequencyHz         = 10000.0,
    .startPhaseDegrees   = 0.0,
    // Used by SOURCE_SWEEP only. Start above stop gives a falling sweep.
    .sweepStartFrequencyHz = 200.0,
    .sweepStopFrequencyHz = 6000.0,
    .sweepStepFrequencyHz = 10.0,
    .sweepHoldMilliseconds = 50,
    // Used by SOURCE_FIXED and SOURCE_NOISE. Not allowed with SOURCE_SWEEP.
    // The frequency and depth are ignored while modulationEnabled is false.
    .modulationEnabled    = true,
    .modulationFrequencyHz = 1000.0,
    .modulationDepth      = 0.5,
};

constexpr ChannelConfig leftChannel = {
    // SOURCE_FIXED, SOURCE_SWEEP or SOURCE_NOISE.
    .sourceType          = SOURCE_SWEEP,
    // Used by every source. 1.0 is full scale; negative inverts.
    .amplitude           = 0.5,
    // Used by SOURCE_FIXED only.
    .frequencyHz         = 2000.1,
    .startPhaseDegrees   = 0.0,
    // Used by SOURCE_SWEEP only. Start above stop gives a falling sweep.
    .sweepStartFrequencyHz = 60000.0,
    .sweepStopFrequencyHz = 20.0,
    .sweepStepFrequencyHz = 100.0,
    .sweepHoldMilliseconds = 50,
    // Used by SOURCE_FIXED and SOURCE_NOISE. Not allowed with SOURCE_SWEEP.
    // The frequency and depth are ignored while modulationEnabled is false.
    .modulationEnabled    = false,
    .modulationFrequencyHz = 100.0,
    .modulationDepth      = 1.0,
};

```

```

};

// Samples per second on each channel. The SAI1 interrupt fires once per
// sample pair, so raising this raises interrupt load one-for-one.
static const int sampleRateHz = 384000;

// ---- Limits -----

// Highest frequency the generator is specified for.
constexpr double maximumFrequencyHz = 60000.0;

// AM puts a sideband on each side of the carrier, at carrier +/- modulation
// frequency. Holding the modulation to 20 % of the carrier keeps the
// sidebands close to it, and the upper sideband must also stay under
// maximumFrequencyHz. Noise has no carrier frequency, so this sketch holds
// noise modulation to 20 % of maximumFrequencyHz, 12 kHz. These limits apply
// to the sines only; they do not narrow the bandwidth of the noise itself.
constexpr double maximumModulationRatio = 0.20;

constexpr bool frequencyInRange(double frequencyHz) {
    return frequencyHz >= 0.0 && frequencyHz <= maximumFrequencyHz;
}

#define CHECK_CHANNEL(config) \
    static_assert(config.amplitude >= -1.0 && config.amplitude <= 1.0, \
        #config ": amplitude must be within -1.0 .. 1.0"); \
    static_assert(config.sourceType != SOURCE_FIXED \
        || frequencyInRange(config.frequencyHz), \
        #config ": frequencyHz must be within 0 .. 60 kHz"); \
    static_assert(config.sourceType != SOURCE_SWEEP \
        || (frequencyInRange(config.sweepStartFrequencyHz) \
            && frequencyInRange(config.sweepStopFrequencyHz)), \
        #config ": sweep start and stop must be within 0 .. 60 kHz"); \
    static_assert(config.sourceType != SOURCE_SWEEP \
        || (config.sweepStepFrequencyHz > 0.0 \
            && config.sweepHoldMilliseconds > 0), \
        #config ": sweep step size and step time must be above zero"); \
    static_assert(!(config.modulationEnabled && config.sourceType == SOURCE_SWEEP), \
        #config ": modulation is not available on a sweep"); \
    static_assert(!config.modulationEnabled \
        || (config.modulationDepth >= 0.0 && config.modulationDepth <= 1.0), \
        #config ": modulationDepth must be within 0.0 .. 1.0"); \
    static_assert(!(config.modulationEnabled && config.sourceType == SOURCE_FIXED) \
        || (config.modulationFrequencyHz >= 0.0 \
            && config.modulationFrequencyHz \
                <= maximumModulationRatio * config.frequencyHz), \
        #config ": modulationFrequencyHz must be at most 20 % of \
frequencyHz"); \

```

```

    static_assert(!(config.modulationEnabled && config.sourceType == SOURCE_FIXED) \
        || config.frequencyHz + config.modulationFrequencyHz \
            <= maximumFrequencyHz, \
        #config ": frequencyHz + modulationFrequencyHz must stay under 60 \
kHz"); \
    static_assert(!(config.modulationEnabled && config.sourceType == SOURCE_NOISE) \
        || (config.modulationFrequencyHz >= 0.0 \
            && config.modulationFrequencyHz \
                <= maximumModulationRatio * maximumFrequencyHz), \
        #config ": noise modulationFrequencyHz must be at most 12 kHz")

CHECK_CHANNEL(rightChannel);
CHECK_CHANNEL(leftChannel);

// ---- Generator state -----

static ChannelState rightState;
static ChannelState leftState;

static const double countsPerTurn      = 4294967296.0; // 2^32
static const double radiansPerPhaseCount = 2.0 * M_PI / countsPerTurn;

// I2S data words are signed 32-bit two's complement; the int32_t cast in the
// interrupt produces exactly that.
static const double fullScale = 2147483647.0;

// Worst-case time spent in the interrupt, in CPU clock cycles, and the number
// of times the transmit FIFO ran empty. Both are printed by loop(). The
// interrupt has one sample period (2.6 us at 384 kHz) to finish. Two fixed
// sines with AM on both is the heaviest setup. If the interrupt is made
// slower than the sample period, loop() never gets to run and the status
// lines stop.
static volatile uint32_t longestInterruptCycles = 0;
static volatile uint32_t fifoErrorCount = 0;

static uint32_t incrementForFrequency(double frequencyHz) {
    return (uint32_t)((frequencyHz / (double)sampleRateHz) * countsPerTurn + 0.5);
}

// Returns the next sample for one channel, from -1.0 to +1.0 at full
// amplitude. Because each ChannelConfig is a compile-time constant, the
// compiler keeps only the branches that channel uses.
static inline double nextSample(const ChannelConfig &config, ChannelState &state) {
    double sample;

    if (config.sourceType == SOURCE_NOISE) {
        // xorshift32: three shift-and-XOR steps turn one 32-bit number into the
        // next, visiting every non-zero value before repeating. Read as a signed

```

```

    // number, the result is white noise spread evenly over full scale.
    uint32_t randomBits = state.noiseState;
    randomBits ^= randomBits << 13;
    randomBits ^= randomBits >> 17;
    randomBits ^= randomBits << 5;
    state.noiseState = randomBits;
    sample = (double)(int32_t)randomBits / 2147483648.0;
} else {
    sample = sin(state.phaseAccumulator * radiansPerPhaseCount);
    state.phaseAccumulator += state.phaseIncrement;
}

if (config.modulationEnabled) {
    // envelope = 1 + depth x modulating sine, which swings between
    // (1 - depth) and (1 + depth). Dividing by (1 + depth) keeps the peaks at
    // the amplitude setting, so 100 % modulation cannot overflow the DAC.
    // Carrier, modulating sine and amplitude are all signed floating-point
    // numbers, so every product carries the correct sign. With depth limited
    // to 1.0 the envelope never goes below zero.
    double modulatingSine = sin(state.modulationPhaseAccumulator *
radiansPerPhaseCount);
    state.modulationPhaseAccumulator += state.modulationPhaseIncrement;
    sample *= (1.0 + config.modulationDepth * modulatingSine)
        / (1.0 + config.modulationDepth);
}

return state.amplitude * sample;
}

// Runs every time the SAI1 TX FIFO drains to its watermark – once per audio
// frame in steady state. Pushes one left and one right sample.
static void sai1_isr(void) {
    uint32_t startCycles = ARM_DWT_CYCCNT;

    // An I2S frame carries the left word first, then the right word.
    I2S1_TDR0 = (int32_t)(fullScale * nextSample(leftChannel, leftState));
    I2S1_TDR0 = (int32_t)(fullScale * nextSample(rightChannel, rightState));

    // FEF is set by the SAI when the FIFO ran empty; writing the bit back
    // clears it.
    if (I2S1_TCSR & I2S_TCSR_FEF) {
        fifoErrorCount++;
        I2S1_TCSR |= I2S_TCSR_FEF;
    }

    uint32_t cycles = ARM_DWT_CYCCNT - startCycles;
    if (cycles > longestInterruptCycles) longestInterruptCycles = cycles;
}

```

```

// Loads one channel's starting values. seed must not be zero: xorshift32
// stays at zero forever once it gets there.
static void startChannel(const ChannelConfig &config, ChannelState &state, uint32_t
seed) {
    state.amplitude = config.amplitude;
    state.noiseState = seed ? seed : 1;
    state.phaseAccumulator = 0;
    state.phaseIncrement = 0;
    state.modulationPhaseAccumulator = 0;
    state.modulationPhaseIncrement = 0;

    if (config.sourceType == SOURCE_FIXED) {
        state.phaseAccumulator = (uint32_t)(int64_t)(config.startPhaseDegrees / 360.0 *
countsPerTurn);
        state.phaseIncrement = incrementForFrequency(config.frequencyHz);
    } else if (config.sourceType == SOURCE_SWEEP) {
        state.sweepFrequencyHz = config.sweepStartFrequencyHz;
        state.sweepHoldStartedMs = millis();
        state.phaseIncrement = incrementForFrequency(state.sweepFrequencyHz);
    }

    if (config.modulationEnabled) {
        state.modulationPhaseIncrement =
incrementForFrequency(config.modulationFrequencyHz);
    }
}

// Moves a sweeping channel to its next frequency once the current one has
// played for sweepHoldMilliseconds. The hold time restarts when the frequency
// actually changes, so a late call never shortens the next hold. The last
// frequency step is made smaller if needed so the stop frequency itself is
// always played.
static void updateSweep(const ChannelConfig &config, ChannelState &state) {
    if (config.sourceType != SOURCE_SWEEP) return;
    if (millis() - state.sweepHoldStartedMs < config.sweepHoldMilliseconds) return;
    state.sweepHoldStartedMs = millis();

    const bool rising = config.sweepStopFrequencyHz >= config.sweepStartFrequencyHz;
    double frequencyHz = state.sweepFrequencyHz;

    if (frequencyHz == config.sweepStopFrequencyHz) {
        frequencyHz = config.sweepStartFrequencyHz;
    } else if (rising) {
        frequencyHz += config.sweepStepFrequencyHz;
        if (frequencyHz > config.sweepStopFrequencyHz) frequencyHz =
config.sweepStopFrequencyHz;
    } else {

```

```

        frequencyHz -= config.sweepStepFrequencyHz;
        if (frequencyHz < config.sweepStopFrequencyHz) frequencyHz =
config.sweepStopFrequencyHz;
    }

    state.sweepFrequencyHz = frequencyHz;
    state.phaseIncrement = incrementForFrequency(frequencyHz);
}

static void printChannel(const char *name, const ChannelConfig &config, const
ChannelState &state) {
    Serial.printf(" %s: ", name);
    if (config.sourceType == SOURCE_FIXED) {
        Serial.printf("fixed %.3f Hz", config.frequencyHz);
    } else if (config.sourceType == SOURCE_SWEEP) {
        Serial.printf("sweep %.1f -> %.1f Hz, now %.1f Hz", config.sweepStartFrequencyHz,
            config.sweepStopFrequencyHz, state.sweepFrequencyHz);
    } else {
        Serial.printf("noise");
    }
    Serial.printf(", amplitude %.3f", (double)state.amplitude);
    if (config.modulationEnabled) {
        Serial.printf(", AM %.1f Hz at %.0f %%", config.modulationFrequencyHz,
            config.modulationDepth * 100.0);
    }
    Serial.println();
}

// Sets PLL4 (the audio PLL) to 24 MHz x (nfact + nmult/ndiv). A PLL that is
// already enabled is left untouched.
static void set_audioClock(int nfact, int32_t nmult, uint32_t ndiv) {
    if (CCM_ANALOG_PLL_AUDIO & CCM_ANALOG_PLL_AUDIO_ENABLE) return;

    CCM_ANALOG_PLL_AUDIO = CCM_ANALOG_PLL_AUDIO_BYPASS | CCM_ANALOG_PLL_AUDIO_ENABLE
        | CCM_ANALOG_PLL_AUDIO_POST_DIV_SELECT(2)
        | CCM_ANALOG_PLL_AUDIO_DIV_SELECT(nfact);
    CCM_ANALOG_PLL_AUDIO_NUM = nmult & CCM_ANALOG_PLL_AUDIO_NUM_MASK;
    CCM_ANALOG_PLL_AUDIO_DENOM = ndiv & CCM_ANALOG_PLL_AUDIO_DENOM_MASK;

    CCM_ANALOG_PLL_AUDIO &= ~CCM_ANALOG_PLL_AUDIO_POWERDOWN;
    while (!(CCM_ANALOG_PLL_AUDIO & CCM_ANALOG_PLL_AUDIO_LOCK)) {}

    CCM_ANALOG_MISC2 &= ~(CCM_ANALOG_MISC2_DIV_MSB | CCM_ANALOG_MISC2_DIV_LSB);

    CCM_ANALOG_PLL_AUDIO &= ~CCM_ANALOG_PLL_AUDIO_BYPASS;
}

// Clocks SAI1 from PLL4 and configures it as I2S master: two 32-bit words

```

```

// per frame, MSB first, data one bit clock after each frame-sync edge.
static void config_i2s(int fs) {
    CCM_CCGR5 |= CCM_CCGR5_SAI1(CCM_CCGR_ON);

    // PLL4 must run between 648 MHz (24 MHz x 27) and 1296 MHz (x 54); the
    // n1/n2 dividers scale the target so the multiplier C lands in range.
    int n1 = 4;
    int n2 = 1 + (24000000 * 27) / (fs * 256 * n1);
    double C = ((double)fs * 256 * n1 * n2) / 24000000;
    int c0 = C;
    int c2 = 10000;
    int c1 = C * c2 - (c0 * c2);
    set_audioClock(c0, c1, c2);

    CCM_CSCMR1 = (CCM_CSCMR1 & ~(CCM_CSCMR1_SAI1_CLK_SEL_MASK))
        | CCM_CSCMR1_SAI1_CLK_SEL(2); // 2 = PLL4
    CCM_CS1CDR = (CCM_CS1CDR & ~(CCM_CS1CDR_SAI1_CLK_PRED_MASK |
CCM_CS1CDR_SAI1_CLK_PODF_MASK))
        | CCM_CS1CDR_SAI1_CLK_PRED(n1 - 1)
        | CCM_CS1CDR_SAI1_CLK_PODF(n2 - 1);

    // MCLK1, selected as the bit-clock source by MSEL below, is the SAI1
    // clock root. The MCLK pad itself is never muxed out; the DAC recovers
    // its clock from BCK.
    IOMUXC_GPR_GPR1 = (IOMUXC_GPR_GPR1 & ~(IOMUXC_GPR_GPR1_SAI1_MCLK1_SEL_MASK))
        | (IOMUXC_GPR_GPR1_SAI1_MCLK1_DIR |
IOMUXC_GPR_GPR1_SAI1_MCLK1_SEL(0));

    // These SAI1 TX pins are free on Basecamp; on Ascent the same pins belong
    // to the parallel sampler.
    CORE_PIN36_CONFIG = 3; // pad B1_02 -> SAI1_TX_BCLK (BCK)
    CORE_PIN37_CONFIG = 3; // pad B1_03 -> SAI1_TX_SYNC (LCK)
    CORE_PIN7_CONFIG = 3; // pad B1_01 -> SAI1_TX_DATA0 (DIN)

    // Transmitter as asynchronous master: it generates BCK and LRCLK itself.
    // The FIFO watermark of 8 fires the refill interrupt with room to spare.
    I2S1_TMR = 0;
    I2S1_TCR1 = I2S_TCR1_RFW(8);
    I2S1_TCR2 = I2S_TCR2_SYNC(0) | I2S_TCR2_BCP
        | I2S_TCR2_BCD | I2S_TCR2_DIV(1) | I2S_TCR2_MSEL(1);
    I2S1_TCR3 = I2S_TCR3_TCE;
    I2S1_TCR4 = I2S_TCR4_FRSZ(2 - 1) | I2S_TCR4_SYWD(32 - 1) | I2S_TCR4_MF
        | I2S_TCR4_FSD | I2S_TCR4_FSE | I2S_TCR4_FSP | I2S_TCR4_FCONT;
    I2S1_TCR5 = I2S_TCR5_WNW(32 - 1) | I2S_TCR5_W0W(32 - 1) | I2S_TCR5_FBT(32 - 1);

    // Prime the FIFO with silence so the transmitter has data the moment it
    // starts.
    for (int i = 0; i < 32; i++) {

```

```

    I2S1_TDR0 = 0;
}

attachInterruptVector(IRQ_SAI1, sai1_isr);
NVIC_ENABLE_IRQ(IRQ_SAI1);
I2S1_TCSR = I2S_TCSR_TE | I2S_TCSR_BCE | I2S_TCSR_FRIE;
}

void setup() {
    Serial.begin(115200);

    // Seed each channel from the hardware random number generator. Each
    // channel then generates pseudorandom samples without waiting for more
    // hardware entropy, which arrives far too slowly for one number every
    // 2.6 us.
    Entropy.Initialize();
    uint32_t rightSeed = Entropy.random();
    uint32_t leftSeed = Entropy.random();

    startChannel(rightChannel, rightState, rightSeed);
    startChannel(leftChannel, leftState, leftSeed);
    config_i2s(sampleRateHz);
}

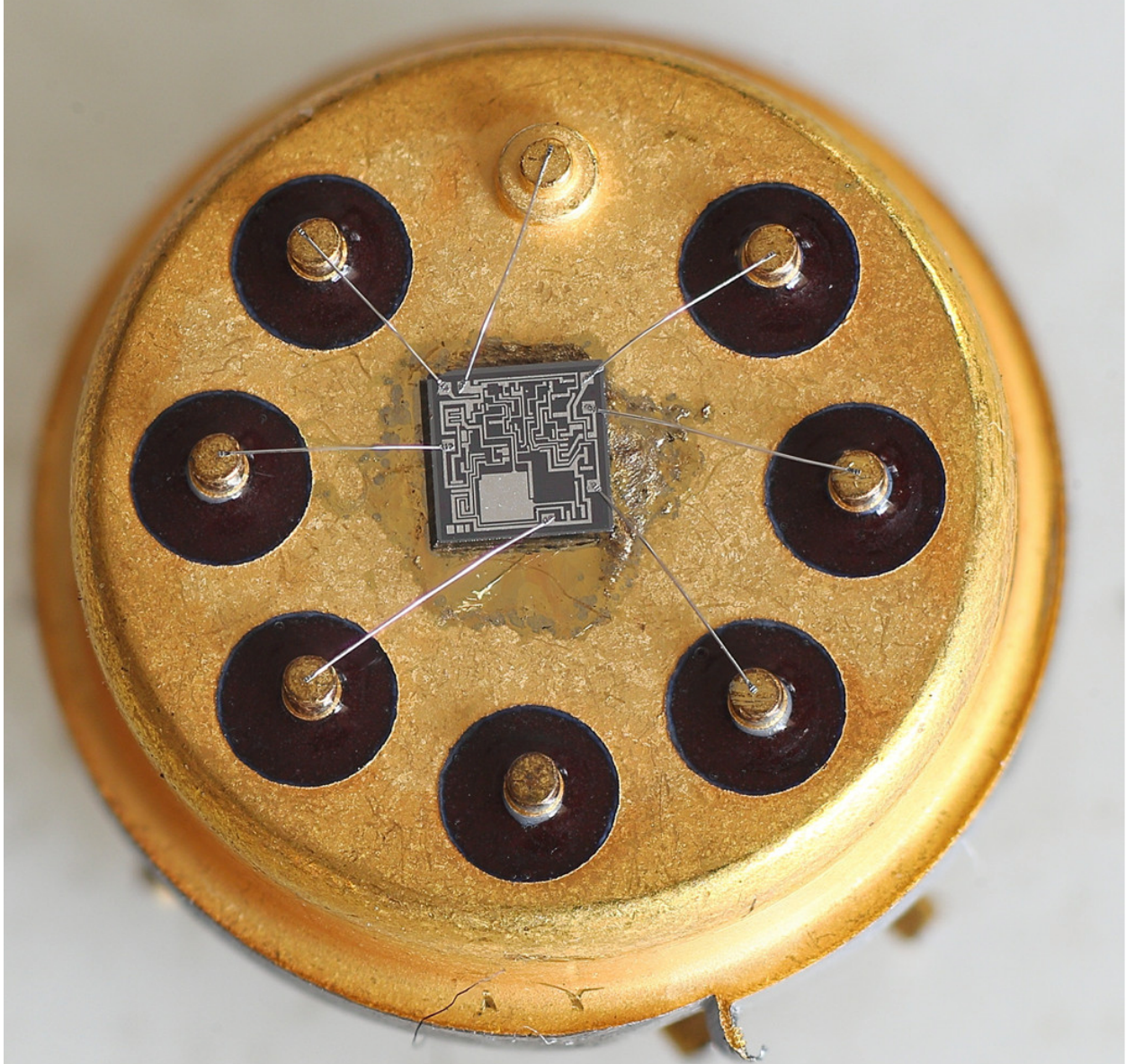
void loop() {
    updateSweep(rightChannel, rightState);
    updateSweep(leftChannel, leftState);

    // Status line, so a silent DAC is distinguishable from a board that never
    // booted or never took the flash.
    static uint32_t lastPrintMs = 0;
    if (millis() - lastPrintMs >= 2000) {
        lastPrintMs = millis();
        double longestInterruptMicroseconds = longestInterruptCycles * 1.0e6 /
F_CPU_ACTUAL;
        Serial.printf("PCM5102A DDS at %d samples/s, up %lu s\n", sampleRateHz, millis() /
1000);
        printChannel("right", rightChannel, rightState);
        printChannel("left ", leftChannel, leftState);
        Serial.printf("  longest interrupt %.2f us of %.2f us available, FIFO errors
%lu\n",
                        longestInterruptMicroseconds, 1.0e6 / sampleRateHz,
                        (unsigned long)fifoErrorCount);
    }
}

```

Appendix 4: The Legendary 741 Opamp

(Just for fun – this chip was released during the Apollo Program!)



https://commons.wikimedia.org/wiki/File:Fairchild_uA741_opamp_6920.jpg

([Creative Commons Attribution-Share Alike 4.0 International](#) license.)