EMBEDDED DEVELOPMENT
AUGUST 2017
ISSUE 325

circuitcellar.com

circuit cellar

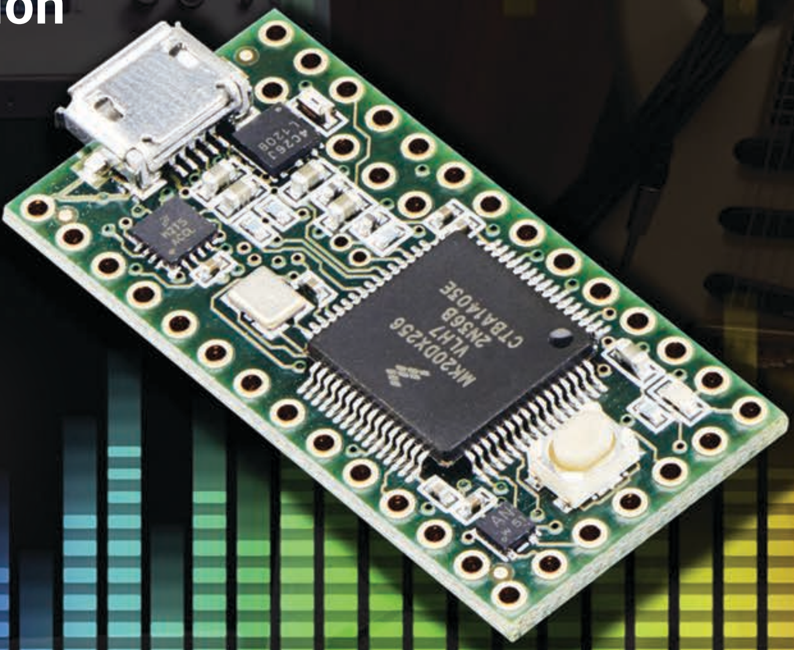
DIGITAL GUITAR AMPLIFER USES ARDUINO

Q&A: Krste Asanovic Talks RISC-V | Power Supplies for Guitar Amps | Design Example: Cypress PSoC | Firmware Upgrade on Microchip PIC32MZ | Reliability and Failure Prediction | PID Temp Control for a Meat Smoker | Inside Electronic Locks | How to Break Software DES Encryption | Fully Differential Amplifiers | More Consulting Tips | Wi-Fi to the Danger Zone

$9.00US $10.00CAN
08
0 74470 75349 0

Digital Guitar Amplifier/Effects Processor (Part 2)

Design and Construction



In the first part of this series, Brian introduced the Teensy 3.2 MCU module. Now he presents a digital guitar amplifier/effects unit that he built around two Teensy modules.

By Brian Millier

In the first part of this series, I introduced the PJRC Teensy family of Kinetis ARM-based modules. I emphasized how they are particularly well suited to audio applications due to the availability of a good audio library. In addition, they are supported by the Teensyduino add-in to the Arduino IDE. This month, I'll describe the digital guitar amplifier/effects unit that I built around two Teensy modules.

HARDWARE DESIGN

The guitar amplifier/effects unit design is about 60% software and 40% hardware. The analog part of the audio signal chain is made up of a simple, one-transistor input buffer and a 20-W output amplifier (using an automotive audio power amplifier IC). A Teensy 3.2 MCU module and a Teensy Audio Adapter module handle all the audio signal processing. **Photo 1** shows both the Teensy 3.2 and the Audio Adapter modules. Notice that the Teensy 3.2 module merely plugs in to the Audio Adapter module, eliminating all interconnecting wiring.

Some of the signal processing is provided by software running on the Teensy 3.2's MCU and the rest is done by the Digital Audio Processor (DAP) contained within the SGTL5000 CODEC device found on the Audio Adapter module itself. The front panel controls and TFT touch screen make up the rest of the circuitry.

Figure 1 is a block diagram of the audio part of the design. Note that the front panel controls and display are not included in this figure. **Figure 2** is a detailed schematic of the project and shows all the circuitry.

An electric guitar sounds best when fed into an amplifier with at least 100-kΩ input impedance, and 500 kΩ is preferred. The Line Input port of the SGTL5000 CODEC has an input impedance of only 29 kΩ, so an input buffer is needed. I used a single NPN transistor in an emitter-follower configuration for this purpose. This presents the necessary high-input impedance to the guitar. The fact that this buffer has only unity gain is not an issue, as the sensitivity of the SGTL5000's Line Input is more than sufficient to handle the guitar signals.

The SGTL5000's Line Input pin goes to an internal amplifier. The gain of which can be set in software using an audio library call. This amplifier isn't shown in the block diagram. My touchscreen user interface provides an input level metering screen, where you adjust this amplifier's gain properly (i.e., having sufficient gain but without overloading the ADC).

This amplified signal is fed to the 16-bit sigma-delta ADC contained in the SGTL5000. There it is converted to a digital datastream in the I²S format. The I²S datastream is fed to the Teensy ARM MCU via its I²S input port.

The Teensy Audio library receives this 16-bit, 44.1-kHz datastream. This process is done in the background, using a DMA controller and interrupt driven routines. The Teensy 3.2 MCU's mainline program configures the various library routines to perform different sound effects, which are then performed in the background. The foreground program looks after the front-panel interface. We'll look at those software-generated effects later.

After all the software signal processing is done by the Teensy 3.2's "C" program, the 16-bit/44-kHz digital datastream is sent out the Teensy 3.2's I²S output port. From there, it reenters the SGTL5000 CODEC on the audio adapter board. It then goes to the CODEC's DAP. The DAP can perform the following functions. Automatic volume control can be set up to act like a compressor, which is a common guitar effect. Surround sound is basically a stereo enhancement effect, particularly effective with headphones. It is not useful with the monophonic guitar signal. I didn't use bass enhancement. Instead, I choose to use the parametric EQ, which is much more versatile. Tone control can be one of the following: simple bass/treble control, five-band graphic EQ, or seven-band parametric EQ. I used the parametric EQ (but I only configured as five bands, as that was all that was necessary in this application).

After the DAP signal processing, the digital datastream is fed to a 16-bit DAC in the CODEC. The analog output signal from the DAC is fed into an analog amplifier contained within the CODEC device. This amplifier has variable gain, which can be set by an audio library routine. However, the resolution of this "volume control" is only 19 steps, which is quite rough. Rather than attempt to use this variable gain amplifier as the guitar amplifier's main volume control, I chose instead to use a fixed gain here and fed the LINE OUTPUT signal into a conventional log-taper potentiometer for overall volume control. (This volume control, as well as all other front-panel controls, are not shown in Figure 1.)

After going through the volume control, the signal enters a power amplifier. This is a TDA7269A integrated circuit in a Multiwatt 11 package. This STMicroelectronics device is actually a 14 W + 14 W stereo amplifier that I have hooked up in a single-channel "Bridge-Tied Load" (BTL) configuration. I chose this device because it is inexpensive, requires a minimum of external passive components, and needs only a single power source. The full power amplifier circuit is shown in Figure 2.

The TDA7269A can drive either 4- or 8-Ω loudspeakers. I chose a 4-Ω, 8" loudspeaker,

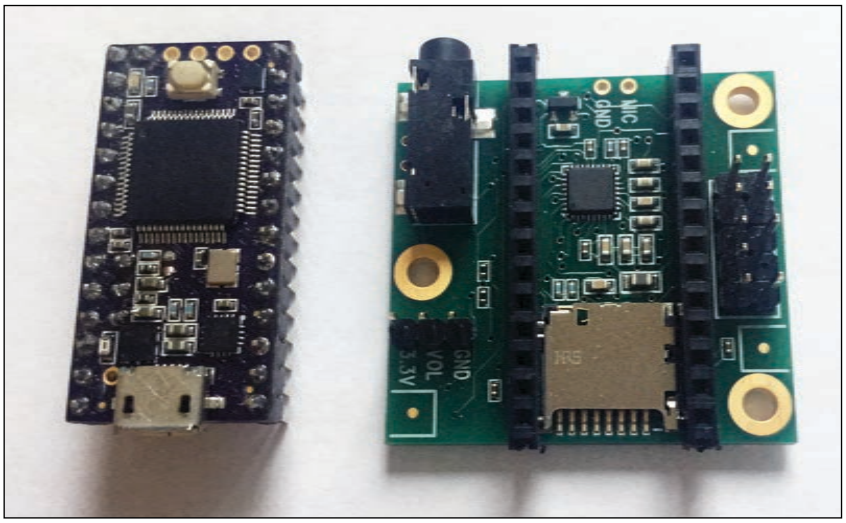


PHOTO 1
The Teensy 3.2 module is the one on the left, the Audio Adapter is on the right. The Teensy 3.2 MCU module plugs into the Audio adapter.

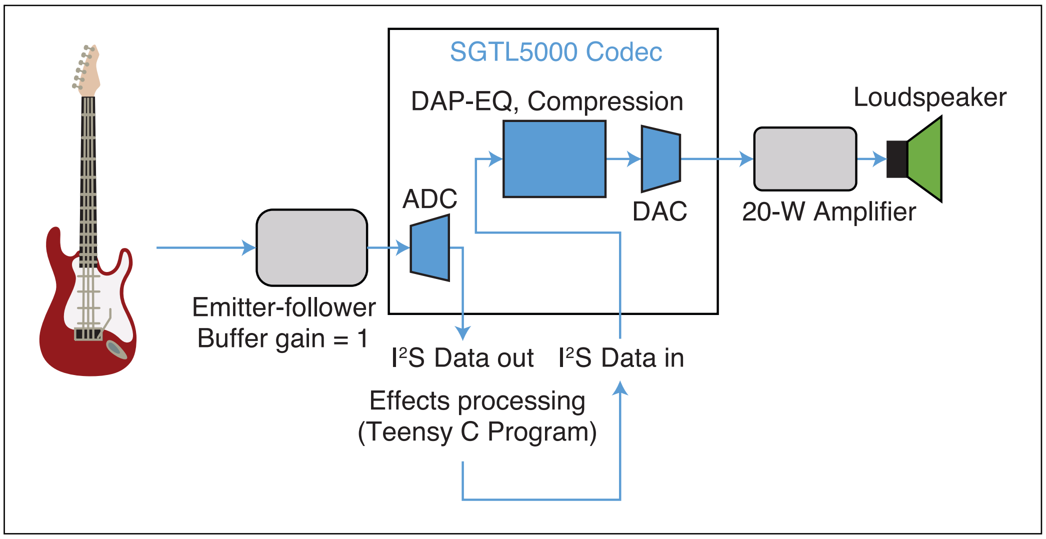


FIGURE 1
This is a block diagram of the audio signal flow through the guitar amplifier.

as the TDA7269A will produce more power at this load resistance (when running from the project's 20-V power supply). The 8" loudspeaker I chose was an automotive "woofer" that was very well-made and much less expensive than loudspeakers that are specifically marketed for guitar amplifiers. Its treble response was a bit lacking, so I added a small 4" speaker to supplement it.

SOFTWARE-GENERATED EFFECTS

In the last section, I mentioned that I was handling the compression and five-band parametric EQ functions using the SGTL5000's internal DAP function block. These features would require a fair amount of processor time to achieve, if they were being done by the Teensy 3.2's ARM MCU, so it was a good idea to "off-load" these functions to the DAP.

The rest of the signal processing is done by the Teensy's ARM MCU, utilizing various audio library routines. In the first part of the article series, I listed all the audio-processing functions contained in this library. Now, I'll cover in more detail the specific ones that I used.

To begin with, let's look at the Audio Designer browser-based application that comes as a part of Teensyduino. This is basically a web application that can be run "on-line" using your web browser. However, it can also installed by the Teensy installer to

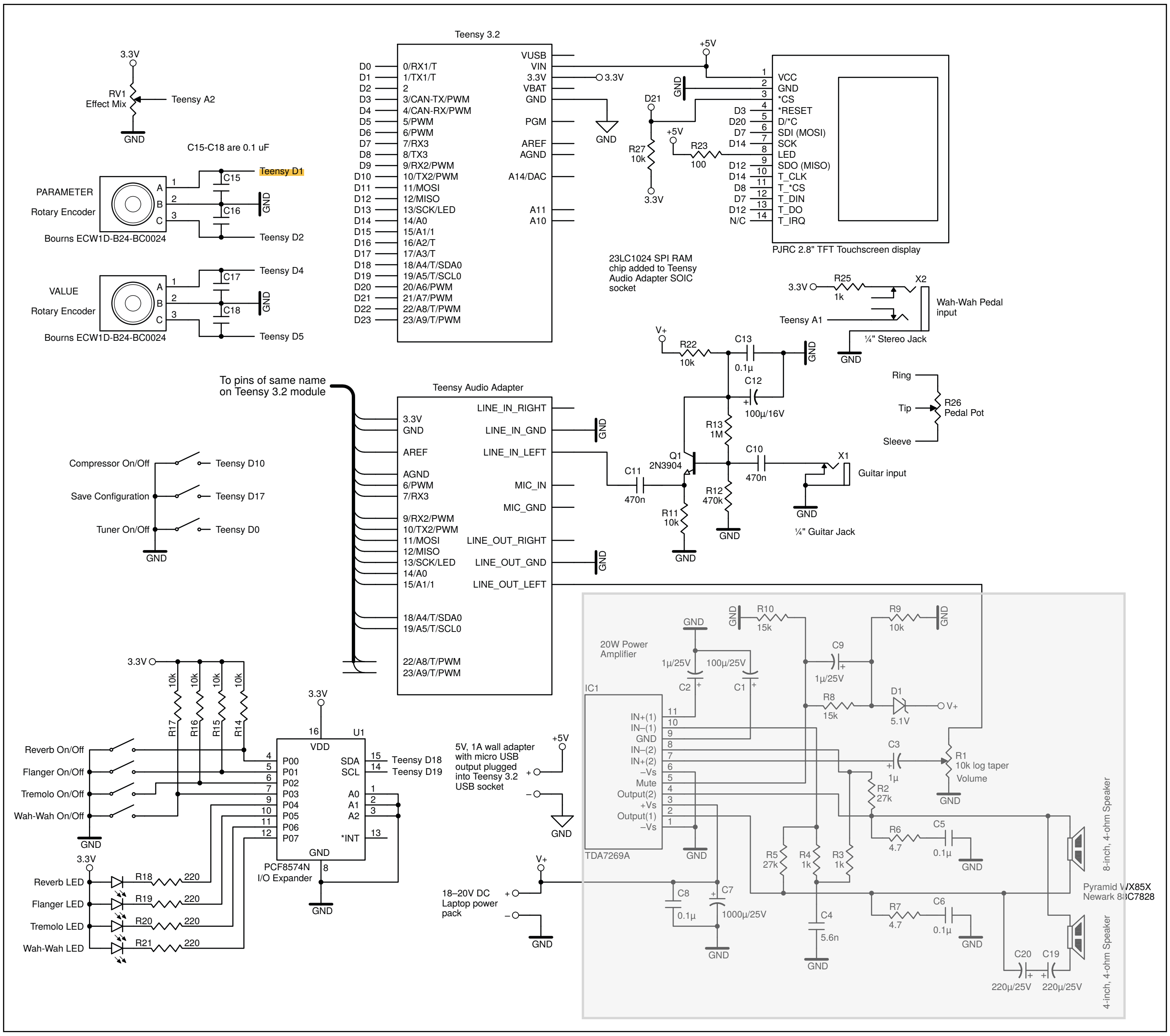


FIGURE 2
This is the complete schematic diagram of the guitar amplifier.

your hard drive, so it can be run "off-line" as well. It can be found in the following folder: C:\Program Files (x86)\Arduino\hardware\teensy\avr\libraries\Audio\gui. And it has a filename of "index.html."

I've included a link to a video tutorial covering this application, which I suggest you watch before use. **Figure 3** shows the audio processing blocks that I used for the effects. This figure is just a screen capture of the workspace window of the Audio Designer program after I composed the design for this project. This program doesn't allow you to annotate your workspace with user-specific labels. I added those manually later, for clarity. I'll start at the top and explain the functions of the relevant blocks.

`Peak1` is a library routine that constantly monitors the input datastream and returns the peak value of the signal (since the last time it was called). It is used to drive a graphic bar meter indicating input level, on the Input Level page of the TFT touchscreen display.

`Notefreq1` is a library routine that monitors the input datastream and returns the frequency of any signals it finds. This is used to drive a tuning meter on the guitar tuner page of the TFT touchscreen display.

`Filter1` and `DC2` library routines are used to implement the wah-wah effect. (I'm assuming that the reader/guitarist will know what these various effects sound like.) `Filter1` is a state-variable (Chamberlin) filter with a corner frequency that is varied by the `DC2` value. The `DC2` value is set by the mainline program, in response to the value of a pot contained in a pedal (such as found in a real wah-wah pedal). The pot voltage is constantly read by one of the MCU's ADC inputs, and that value is used to set the value of `DC2`.

`Flange1` is a library effect used to produce flanging. It consists of a software delay line, which contains numerous "taps" that are mixed with the original signal to provide a "comb-filter." This provides the "swooshing" (moving) notch-filter effect that is known as flanging.

`Sine1`, `dc1`, `mixer2`, and `multiply1` implement the Tremolo effect. Basically, `sine1` is a low-frequency sine wave (2 to 6 Hz) that is added to a fixed DC offset (provided by DC1). This offset sine wave is fed into one input of `multiply1`. The other is fed with the guitar signal. This multiplication results in an amplitude modulation of the guitar signal at a slow rate, or tremolo. The program allows for the adjustment of both frequency and modulation depth of the tremolo effect.

`DelayExt1`, `mixer3,4` provide the reverb effect. `DelayExt1` is a software delay line that can hold about 1.5 s of audio signal. The delay line has up to eight taps, of which I use four. Some of these delayed signals are sent to `mixer3` (at various gain levels) and fed back into the delay line. This feedback extends the 1.5 s into a longer reverb decay time, making for a more realistic effect. There is really not enough SRAM in the Teensy 3.2's MCU to handle reverb realistically. The `Ext1` refers to external RAM memory. In this case, it is a 23LC1024 SPI RAM chip (128 KB) that is mounted on the Audio Adapter module (in the blank, SOIC footprint provided). Note that there is also a delay library function that works identically, but instead uses internal MCU SRAM and is limited to about 425 ms (assuming your program is not using most of

ABOUT THE AUTHOR

Brian Millier runs Computer Interface Consultants. He was an instrumentation engineer in the Department of Chemistry at Dalhousie University (Halifax, NS, Canada) for 29 years.

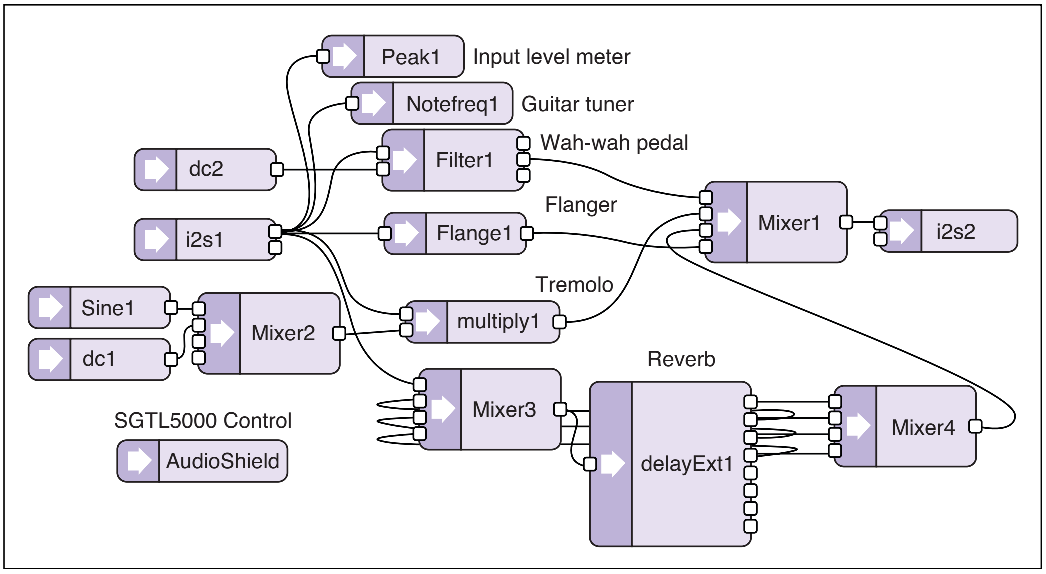


FIGURE 3
This is the diagram of the audio library routines, running on the Teensy MCU, which perform the digital signal processing for effects, etc.

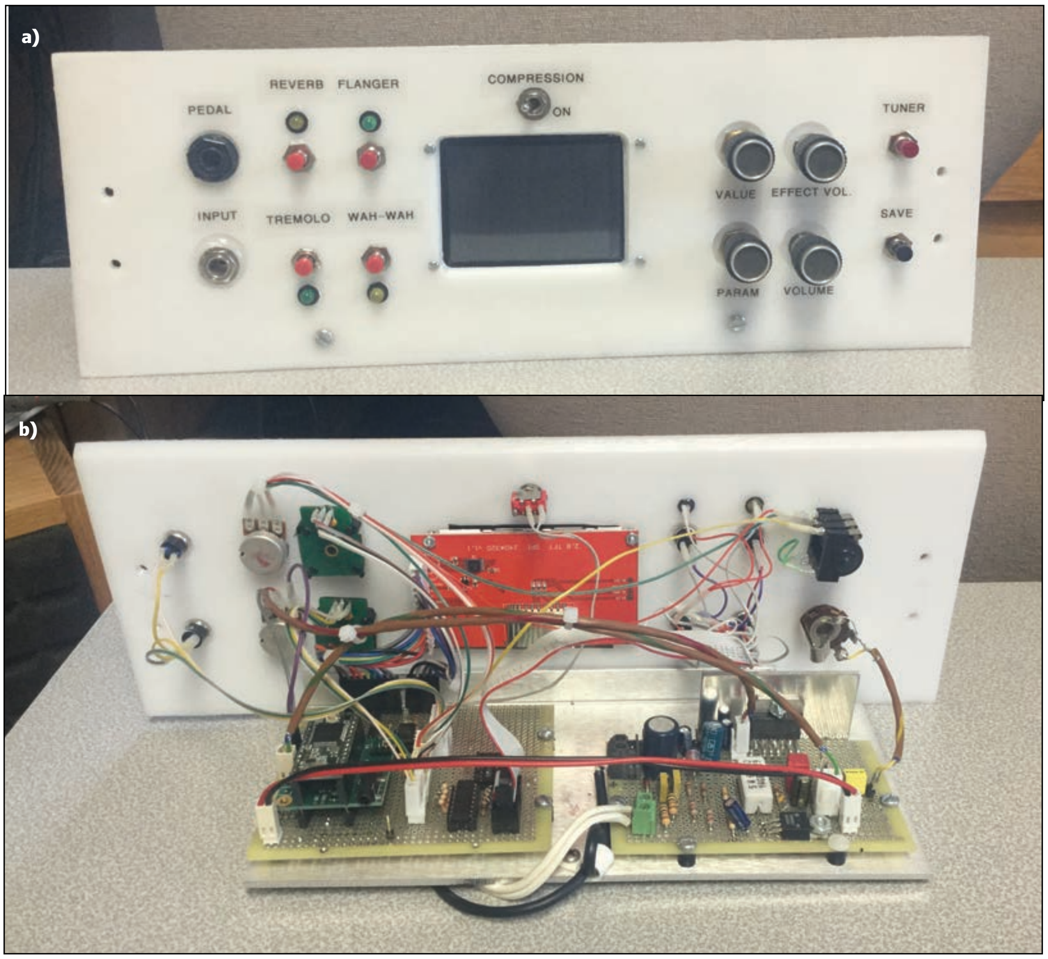


PHOTO 2
a—This is a close-up of the front panel. b—This shows the circuitry. The left board contains the two Teensy modules and the PCF8574N I/O expander. The right board contains the preamplifier and power amplifier.

the internal SRAM for other purposes).

`Mixer1` is a four-input mixer that combines the datastreams from the various effects. Each input has its own gain setting. These gain values are set by the user in a touchscreen graphics page dedicated to that effect.

`AudioShield` is the library routine that controls the configuration/setting of the audio CODEC you're using. (The four supported CODECs were mentioned in Part 1.) In this project, the Audio Adapter contains the SGTL5000 CODEC. The `AudioShield` routine sends out the necessary commands via the I^{2}C bus to the CODEC. You must place this block in your workspace if you are using any of the CODEC devices in your project or nothing will work!

FRONT-PANEL CONTROLS & DISPLAY

Because the project contains so many effects and adjustable parameters, designing an efficient front panel was a bit of a challenge. I didn't have room to mount a large number of pots, nor did I want to buy a lot of them. So, I settled on a design in which the parameters that might be routinely changed would have a dedicated front panel pot or switch. All the other parameters would be adjusted via a graphic user interface implemented on a small color thin-film-transistor (TFT) touchscreen display. **Photo 2** shows the front panel and the circuitry behind the panel.

The Teensyduino add-on to the Arduino IDE contains libraries that support several small TFT displays that are currently available. I've used a few different 3.2" to 4.3" TFT touchscreen displays in earlier projects. They were a bit expensive for this project and weren't directly supported by Teensyduino, so I instead chose a readily-available 2.8" TFT touchscreen based upon the ILI9341 controller chip. There is a Teensyduino library for this display, and they are a very reasonable choice for a project like this, at only $15.

These displays have good resolution (320 × 240) and are quite bright. However, due to their small size, I did not think it was practical to use the touchscreen for actually adjusting the parameters. To do so would have limited the number of parameter sliders visible on each screen to just a few to make them large enough for my fingers to be able to adjust easily. I decided that I would use the touchscreen just for moving between the various display pages. A tap on the left side would switch to the previous menu page. A tap on the right would advance to the next.

The actual parameter adjustments would be performed using two rotary encoders: a `value` encoder to adjust the parameter value itself and a `param` encoder, which would cycle through the various parameters shown on any given display page. So, you would highlight the parameter you wanted to change with this encoder, and then use the value encoder to change it.

I have a Save button on the front panel. Whenever this is pressed, the current values of all parameters are saved to the MCU's internal EEPROM. At power-up, these parameters are reloaded automatically.

Each effect has its own On/Off pushbutton switch and an LED to indicate whether it is active or not. There is a dedicated potentiometer for the main volume control, as well as a separate Effects volume control. While each effect has its own gain adjustment (on the touchscreen display's menu page for that effect), the Effects volume pot acts as an overall mix control for all the effects. Last, there are dedicated switches to activate the compressor and to invoke the guitar tuner.

With all these switches and LEDs, there weren't enough free I/O pins on the Teensy 3.2 to handle them. This isn't surprising: it is only a 28-pin module and it is already connected to the Audio Adapter module, the TFT display screen, and a 23LC1024 SRAM device. Therefore, I added an NXP PCF8574N I^{2}C 8-bit expander chip, which handles the four effects pushbuttons and LEDs. Note that the PCF8574 comes in a couple of versions. Make sure you get the "N" version, as its I^{2}C address matches the I^{2}C address constant found in the program code (other versions use different addresses).

POWER SUPPLY

I decided to use the readily available 18-to-20-V power packs used by laptop computers, as a power supply for this project. Most everyone has one of these from an old laptop lying around surplus. These "bricks" are ideally suited to the TDA7269A's voltage requirements, and they can supply 3 to 5 A, which is more current than needed.

Besides running the power amplifier, we must also supply the Teensy 3.2 module with 5 V. The Teensy 3.2 module itself has an on-board 3.3-V regulator, which supplies both the Audio Adapter module and the TFT display screen. The TFT display's backlight is supplied by the same 5-V supply that the Teensy 3.2 uses.

During the development phase of this project, I ran the power amplifier from a 19-V laptop power pack. The preamplifier ran on 12 V derived from the 19-V supply. The Teensy module was powered by the USB cable that connected to my computer. The USB port was needed, during the development phase, for programming and serial debugging. The Audio Adapter module and TFT display were getting their power from the Teensy 3.2, as I mentioned earlier.

Once development was done and the USB port was no longer being used for power/programming/debug, I intended to use a small switching regulator module to provide the necessary 5 V from the 19-V power supply. While this worked, I found that using a common power supply for both the digital (Teensy 3.2/Audio Adapter module/TFT) and the analog sections (preamplifier/power amplifier) resulted in ground-loop issues. This presented as a low-level buzzing noise that was loud enough to annoy me. I am familiar with audio circuitry, and tried everything I knew of to eliminate this ground loop noise. In the end, I found the only solution was to provide a separate 5-V power supply for the Teensy 3.2 (and also the Audio adapter and TFT display to which it supplies 3.3-V power). To keep the power supply design inexpensive, I used a common 5-V, 1-A adapter module, with a micro-USB plug that plugs into the Teensy 3.2 module directly. These adapters are very inexpensive as they are commonly used as cell phone chargers.

Photo 3 shows the finished amplifier. The guitar input and wah-wah pedal jacks are at

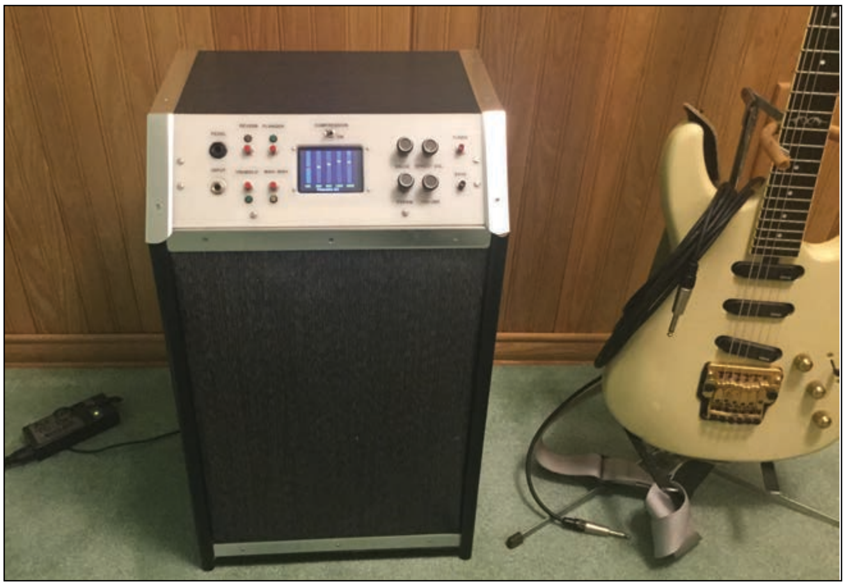


PHOTO 3
This is the finished project. There are quite a few controls and switches even though most of the Effects parameter adjustments are done on the touchscreen using various pages of the GUI.

the far left. The effects switches/LEDs are next over. You can see the TFT display in the middle, showing the parametric EQ screen. The rotary encoders and the main/effects volume controls are next and the Save to EEPROM and guitar tuner push-buttons are at the extreme right.

SOFTWARE SNAGS

In this section, I'd like to detail some of the complications I ran into while using the Teensyduino Audio library, and explain what I did to get past them. I need to stress that you must handle the issues that I mention, or the (supplied) source code, which you compile to run this project, won't work!

A few of the routines in the Teensyduino Audio library didn't work properly, or at all. I corrected the library code for those routines that were needed for this project. While I reported them on the proper github site, while I was writing this article some of them had yet to be corrected.

I also had to modify a few library routines to perform functions that I needed, but weren't implemented in the original code. In one case, I had to change a few variables typed as Private into Public, to allow my code to modify them. Doing this, for example, allowed me to configure some SGTL5000 CODEC features that I needed, but weren't otherwise accessible.

To implement the guitar tuner feature, I made use of the `notefreq` function in the Audio library. As written, this function works great, but there are two complications. This routine alone uses 70% of the MCU's processing power. That does not leave much processing power for everything else the MCU must handle. Once the routine is initialized, it can't be turned off. So, even if you could live without any of the software-based effects running at the same time as the tuner, you can't recover that processing power by shutting the tuner off when it isn't being used. I made changes to this library routine to reduce its processor usage to 40% and to allow it to be shut off when not needed.

If you want to use the source code that I wrote for this project, you can download it from the *Circuit Cellar* FTP site. However, you still have to compile it from within the Arduino IDE to get the hex code needed to upload to the Teensy 3.2. The compile process will use the "official" audio library files that you received when you installed the Teensyduino add-on to the Arduino IDE, as I described in the first part of this series Some of the corrections mentioned above may be missing from the "official" library files, if they have not been updated by PJRC. Certainly, the changes I made to a few of the files, to customize them for this project, will not be present in the official library files.

For this reason, I am including the library files that I made changes to, in the ZIP file that also contains the Arduino .INO file for this project. Also included is a text file (changelog.txt) which explains what these changes are, and at what line(s) within the library file they are found.

You should replace the "official" library files with the ones I supply. The folder that these files generally reside in (unless you customize your version of Teensyduino) is: \Program Files(x86)\Arduino\hardware\teensy\avr\libraries\Audio. Although this path contains a folder named AVR, this is a byproduct of the fact that early Teensy boards used AVR MCUs. Subsequent Teensy boards that are ARM-based (such as the Teensy 3.2 used in this project) still place their libraries within this AVR folder hierarchy. To replace these files, which are within the program files(x86) folder of the PC, you must copy these files using Administrator access privileges.

LOW-COST AUDIO PROCESSING

I have been working with microcontrollers for 25 years both at work and for hobby projects. But, this is the first time I have done a project that performed audio processing in software, and it was quite intriguing to see the project come together. While you may not be interested in duplicating my project exactly, I hope you can gain some useful tips regarding the low-cost audio processing capabilities of the NXP Kinetis ARM processors found on the Teensy line of MCU modules.

circuitcellar.com/ccmaterials

RESOURCE

PJRC, "Teensy Audio Designer Tutorial & Workshop," www.pjrc.com/teensy/td_libs_Audio.html.

STMicroelectronics, "TDA7269A: 14W+14W Stereo Amplifier with Mute & St-By," 2003.

Texas Instruments, "PCF8574 Remote 8-Bit I/O Expander I²C Bus," SCPS068J, 2015.

SOURCES

Teensy 3.2 MCU Module
PJRC | www.pjrc.com

Kinetis K20 Microcontrollers
NXP Semiconductors | www.nxp.com

TDA7269A Stereo amplifier
STMicroelectronics | www.st.com

PCF8574N
Texas Instruments | www.ti.com