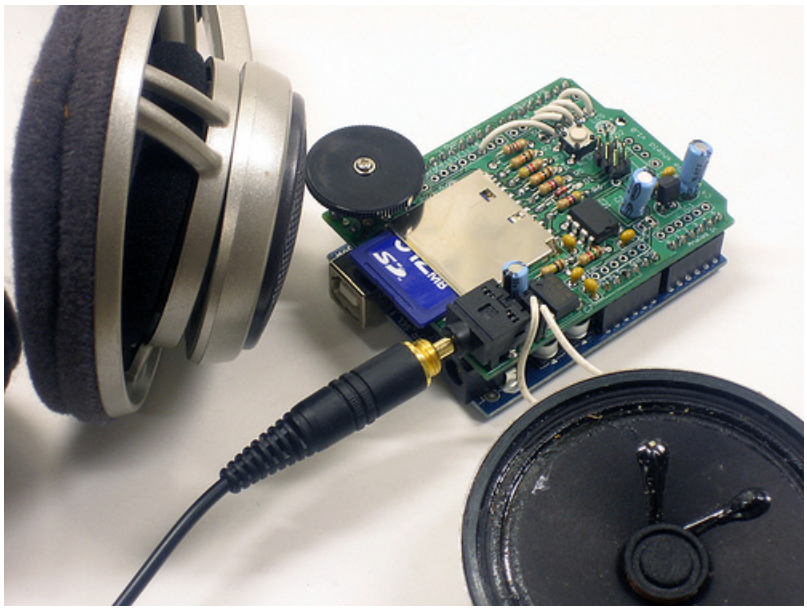
adafruit learning system

Wave Shield

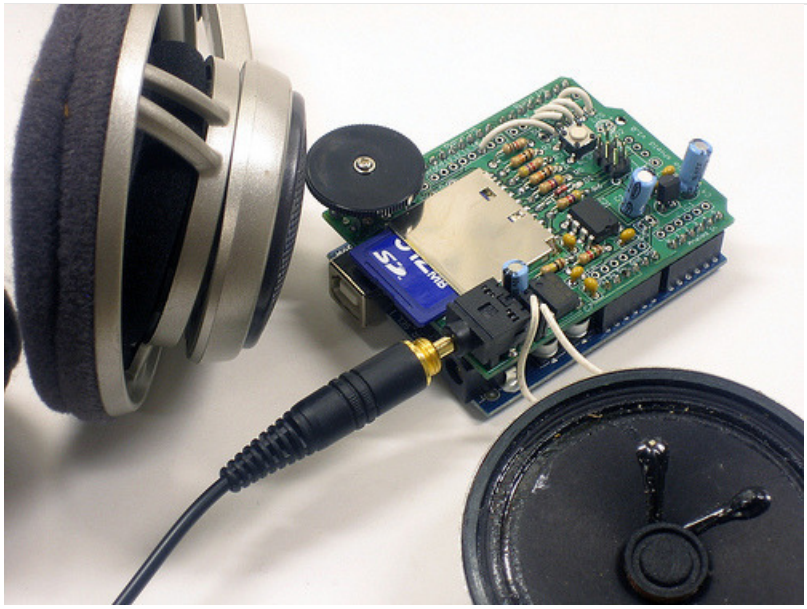
Created by lady ada



Last updated on 2018-08-22 03:34:24 PM UTC

Guide Contents

Overview



Adding quality audio to an electronic project is surprisingly difficult. People tend to end up either using low-quality ISD chips (https://adafru.it/d89) (you might get 8Khz sampling rate for 30seconds out of these, if you're lucky!) or mucking around with trying to control a CD or MP3 player. Although it's possible to generate audio direct from a microcontroller using a PWM output (https://adafru.it/d8a), the quality is often low and it's hard to fit a lot of music in an EEPROM chip. You can buy an embedded MP3 player board, but they're either expensive or difficult to use!

Here is a shield for Arduino that solves many of these problems. It can play up to 22KHz, 12bit uncompressed audio files of any length. It's low cost, available as an easy-to-make kit. It has an onboard DAC, filter and op-amp for high quality output. Audio files are read off of an SD/MMC card, which are available at nearly any store. Volume can be controlled with the onboard thumbwheel potentiometer.

Click on the play button to watch a demo of the wave shield playing assorting audio through a small speaker.

The shield comes with an Arduino library for easy use; simply drag uncompressed wave files onto the SD card and plug it in. Then use the library to play audio when buttons are pressed, or when a sensor goes off, or when serial data is received, etc. Audio is played asynchronously as an interrupt, so the Arduino can perform tasks while the audio is playing.

- Can play any uncompressed 22KHz, 12bit, mono Wave (.wav) files of any size. While it isnt CD quality, it is certainly good enough to play music, have spoken word, or audio effects.
- Output is mono, into L and R channels, standard 3.5mm headphone jack and a connection for a speaker that is switched on when the headphones are unplugged.
- Files are read off of FAT16 formatted SD/MMC card (https://adafru.it/c0k).
- Included library makes playing audio easy.

While the shield has been tested and works well, here are some points to keep in mind:

- The audio playback library uses 10K of flash - so if you want to use an NG arduino, you'll need to upgrade to an Atmega168 chip.
- About 600 bytes of SRAM are used to buffer the audio and keep track of file data, so RAM-heavy projects may not work well.
- The shield can't play MP3, WMA, Ogg or other compressed audio files. It can only play uncompressed PCM/WAV

files. Converting audio to WAV format (https://adafru.it/s8f) is very easy, and is often the default format for many audio programs.
- Files are stored as 8.3 name format, and can only be placed in the root directory. That means you can only have ~512 files (but they can be any size).

Ideas for what you can use it for...

- Make a portable audio player
- Use the AT&T text-to-speech site to make snippets of speech that you string together for a talking project, like..
- Talking temperature sensor
- Talking clock
- Interfaces for sight-impared people
- Doorbell that plays a cool tune
- Jukebox/music-box that plays a song when its opened, or a coin is inserted
- Security system that warns the intruder
- Audio looper for musical effects and performances
- Synthesizer with different sounds
- Really freaky halloween props that scream
- Display (like a point-of-sale box) that you can plug into to hear the message

FAQ

Can this shield play MP3 files? What about WMA, Ogg, AAC, etc?

No, compressed audio requires either a specialized chip (which is expensive) or a very powerful chip. The Arduino microcontroller can't uncompress MP3 on the fly and to keep the shield inexpensive, no mp3 decoder chip is included.

For recording and playback of other file formats, see the VS1053 breakout board.

What sort of audio can it play?

It can play uncompressed Wave files (.wav format). This is a standard format and pretty much every audio program can convert your music or audio into wave format. Make sure the sample rate is mono, 22KHz (or less) and 16-bit (or less). The user manual has instructions on how to convert and adjust files for optimum playback.

Can I play two files at once?

No, the waveshield can **only play one WAV file at a time**. There is no way to play two or more wave files at one time - the Arduino is not fast enough to mix audio.

Can it play Stereo?

No, the software libraries, and hardware DAC and amplfier do not support stereo sound. You could split the mono output into two speakers but they wont be 'true stereo.'

What does it sound like?

The best way to determine if the quality is good enough for your project is use Audacity and go thru the steps in the User Manual for converting MP3s (and other files) to 22KHz/16-bit format.

Can this shield record audio?

There is no hook-up for a microphone, so it will take a bit of hacking. But recording is possible with the WaveRP library by fat16lib.

Don't try this with an older Arduino (atmega168). You need the memory of an Atmega328 (such as an Arduino Uno).

What pins are used by the shield?

Pins **13, 12, 11** are always used by the SD card (they are the only pins that have a high speed SPI interface). Then there are 5 other pins used to talk to the DAC and SD card, but they can be set to connect to any arduino pin. However, by default, the library is configured to use pins **10** (for SD card) and pins **2, 3, 4** and **5** for the DAC. To chanage these pins requires modifying the library - the pins are referenced by their 'hardware' pin names (ie PORTD, etc) not by arduino pins.

That means pins **6, 7, 8, 9** and the 6 analog in pins (also known as digital i/o pins **14-20**) are available.

How come I cant use the PWM output on pin 9?

Timer 1 is used by the wave shield for timing, if you want to use a servo, you can use ServoTimer2 or a 'softservo' library.

What are LED1, LED2, R9 & R10 for? They're not in the instructions.

These are 'spots' for unincluded components, you can install 3mm LEDs into the slots and 1K resistors into the matching resistor slots. Theres a solder hole next to them so you can wire up the LEDs as indicators. There's no

software in the shield library to support them, they're just component locations.

What is CD, WP and SW?

CD is the Card Detect switch in the SD card holder, WP is the Write Protect detect switch in the SD card holder, SW is the switch in the potentiometer. See the schematics for how these are connected up. They are not used in any libraries, there is no example code for them. If you want to use them they are there but are not necessary.

Design

Overview

Here is an explaination of how the wave shield works. We'll go section by section. You'll want to refer to the schematic.

Voltage regulator

The easiest thing to understand is the 3.3V voltage regulator. This takes the 5V supply from the Arduino and converts it to a nice 3.3V supply. This is necessary because SD/MMC cards only work on 3.3V. If you give them 5V they'll burn out & die!

The voltage regulator used is the MCP1700-330, which can provide up to 250 mA of current. There are 4 capacitors associated with the regulator. **C1** and **C2** are the input capacitors; they stabilize the 5V input. **C3** and **C4** are the output capacitors, they stabilize the 3.3V output

There is a jumper that allows you to skip the regulator and use the 'built in' 3.3V supply from the Arduino. However, it is not suggested as that supply is not guaranteed to provide the current necessary.

SD/MMC card holder

SD/MMC cards are very popular, small, and inexpensive. The card holder is what allows you to remove and replace the card easily. They can be removed/replaced thousands of times. The top three 'pins' are **CD, WP** and **COMMON_SW**. CD stands for "card detect" this is a mechanical switch that closes when the card is inserted. WP stands for "write protect," this is a mechanical switch that closes when the card has the little side tab slid down to 'lock.' COMMON_SW is the common connection for the two switches. We simply connect this to ground. Thus CD and WP will be grounded when active.

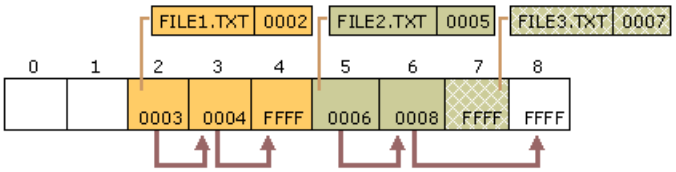
At the bottom are the power supplies. There are 2 mechanical ground connections and a logic ground. There is also the logic power connection, connected to the 3.3v regulator.

In the middle are the data connections. **DAT1** and **DAT2** are for advanced/high-speed SD card interfacing. We don't do this so they are left disconnected. **DATA_OUT** is the serial data out from the card, which is connected to the SPI port of the Arduino. **DATA_IN** is the input and **SCLK** is the clock input. Since they must be 3.3V and the Arduino usually sends 5V data, we use voltage dividers (**R2, R3, R4** and **R5**) to reduce the inputs down.

CS is the select line, used to tell the MMC that we want to send it data. This line is pulled low (to ground) when we want to send data to the card. That means we need to make sure when we dont have anything connected, the pin is pulled high to ~3.3V. We use **R6** as the pullup and zener diode **D1** to keep the voltage at 3.3V. **R1** allows the diode to bias properly when the Arduino pulls the pin high.

The microcontroller/Arduino

The library contains a bunch of specialized code. The first part is a 'FAT16' library, this is a set of functions that allow the chip to read the SD card, locate files and read their contents. The method it does this by is particularly detailed and you can read the SD/MMC (https://adafru.it/c0n) and FAT16 manuals (https://adafru.it/c0o) if you're interested.



Once it opens a file and is ready to read it, it looks through the first section of the file. If it's a Wave file, there will be all sorts of information stored in this header that will indicate the channels (mono/stereo/etc), bits-per-sample (8 to 32), sample rate (ie 16KHz) etc. You can read more about the header format here (https://adafru.it/c0p). Basically, the firmware verifies that it is mono channel, 16 or less bits-per-sample and 22KHz or less sample rate. Then it sets up the audio interrupt that will go off sample-rate times a second. For example, if it's a 22KHz audio sample, the interrupt will go off 22,000 times a second!

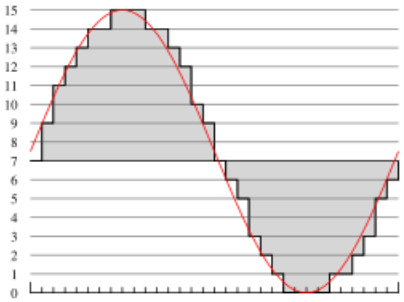


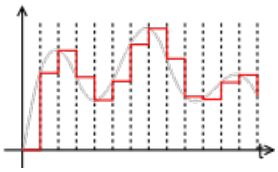
Image from wikipedia (https://adafru.it/c0o)

The audio is encoded in PCM format (https://adafru.it/c0q). This means "pulse Code Modulation". Lets say it's a 16bit, 22khz wave. The audio waveform is sliced up 22,000 times a second and a corresponding value (up to 16 bits - from 0 to 65,635) is read from the waveform, then that value is stored in the file. Each sample is a unique value. The file is not compressed. This means the files are very large but the quality is very very good.

The SD card can provide 512 bytes at a time. This is buffered inside the Arduino's RAM so that we have smooth playback. (Techinally, its a double-buffer which means we read 256 bytes and play 256 bytes, then swap.) The audio interrupt picks one sample at a time and sends the data to the DAC (digital/analog converter).

DAC

The DAC is a very simple device. When you send it data it will convert that digital information back into an analog signal!



You'll notice it actually doesn't get the orignal waveform perfect. The more bits of digital data, the higher quality of audio reproduction. CDs have 16-bits per sample. While it would have been nice to have a 16-bit DAC, the best option for this design was a 12-bit dac. (That's still quite good.)

The microcontroller/Arduino uses the DAC_CS (chip select), DAC_CLK (data clock), DAC_DI (data), and DAC_LATCH (convert the digital to analog) pins to send the sample data over. The DAC also has a Vref input, this is the reference voltage that it uses to define the maximum analog value it can generate. There is a very low low-pass filter connected to it (**C6** and **R8**) so that any digital noise (there is -a lot-) will not make it into the audio signal.

There is another low-pass filter connected to the output of the DAC (**R7** and **C8**). This is for filtering out the 'square wave' component you see in the recreated-audio wave. Even though the noise is only 1/4096'ths of the signal (about 1.2mV) it's still noise and these two components filter out anything above 11KHz. The reason the filter cut-off frequency is 11KHz and not 22KHz is that if you sample at 22KHz you will only be able to reproduce frequencies at half that rate, 11KHz. This is the Nyquist (https://adafru.it/vCL) theory. It is sneaky but true. If you try to sample 16KHz waveform at 22KHz it will actually sound much -lower-, it will play at 6KHz (it is 'mirrored' around 11KHz).

Analog output

Finally there is the volume control and output stage. The potentiometer acts as a simple volume control. It simply divides down the analog signal from 5Vpp down to as low as 0Vpp. The pot is 'audio' type which means that the voltage changes logarithmically, which our ears interpret as linearly.

The analog signal then goes into a high-output, rail-to-rail opamp. This op-amp can provide up to 100mA per channel. The two channels are hooked up in parallel for up to 200mA output (at 5V). This means it can provide 1/8 W into an 8ohm speaker (or 1/4 W into 4ohm speaker). This isn't enough for a boom-box but its good for headphones and small speakers. The output is filtered through a bypass capacitor **C9** which will keep any DC voltage from going to the speaker, which could damage it.

The headphone jack is stereo, which both mono channels connected in parallel. This gives the most power output. There are internal switches in the jack so that when the headphones are removed, the audio flows to the 'speaker connection' next to the jack.

Make it!

Steps

This is a very easy kit to make, just go through each of these steps to build the kit.

1. Tools and preparation (https://adafru.it/cmj)
2. Check the parts list (https://adafru.it/cWR)
3. Solder it (https://adafru.it/cWS)

Preparation

Prep

Learn how to solder with tons of tutorials (https://adafru.it/aTk)!

Don't forget to learn how to use your multimeter too (https://adafru.it/aOy)!

Tools

There are a few tools that are required for assembly. None of these tools are included. If you don't have them, now would be a good time to borrow or purchase them. They are very very handy whenever assembling/fixing/modifying electronic devices! I provide links to buy them, but of course, you should get them where ever is most convenient/inexpensive. Many of these parts are available in a place like Radio Shack or other (higher quality) DIY electronics stores.

Soldering iron

Any entry level 'all-in-one' soldering iron that you might find at your local hardware store should work. As with most things in life, you get what you pay for.

Upgrading to a higher end soldering iron setup, like theHakko FX-888 that we stock in our store (http://adafru.it/180), will make soldering fun and easy.

Do not use a "ColdHeat" soldering iron! They are not suitable for delicate electronics work and can damage the kit (see here (https://adafru.it/aOo)).

Click here to buy our entry level adjustable 30W 110V soldering iron. (http://adafru.it/180)

Click here to upgrade to a Genuine Hakko FX-888 adjustable temperature soldering iron. (http://adafru.it/303)

Solder

You will want rosin core, 60/40 solder. Good solder is a good thing. Bad solder leads to bridging and cold solder joints which can be tough to find.

Click here to buy a spool of leaded solder (recommended for beginners). (http://adafru.it/145)

Click here to buy a spool of lead-free solder. (http://adafru.it/734)

Multimeter

You will need a good quality basic multimeter that can measure voltage and continuity.

Click here to buy a basic multimeter. (http://adafru.it/71)

Click here to buy a top of the line multimeter. (http://adafru.it/308)

Click here to buy a pocket multimeter. (http://adafru.it/850)

Flush Diagonal Cutters

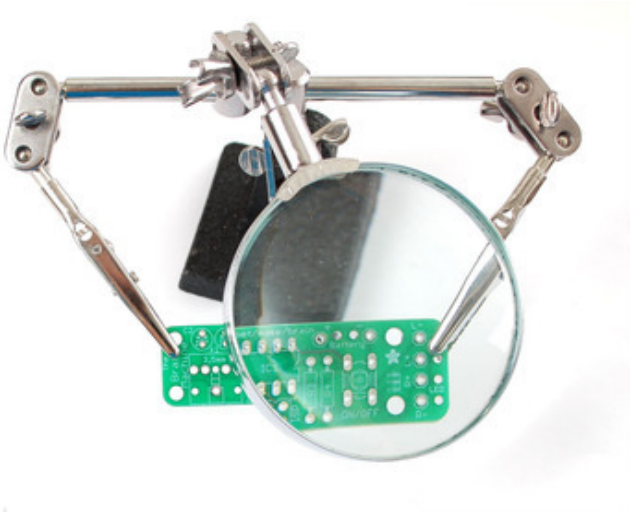
You will need flush diagonal cutters to trim the wires and leads off of components once you have soldered them in place.

Click here to buy our favorite cutters. (http://adafru.it/152)

Solder Sucker

Strangely enough, that's the technical term for this desoldering vacuum tool. Useful in cleaning up mistakes, every electrical engineer has one of these on their desk.

Click here to buy a one. (http://adafru.it/148)



Helping Third Hand With Magnifier

Not *absolutely* necessary but will make things go much much faster, and it will make soldering much easier.

Pick one up here. (http://adafru.it/291)

Parts

Parts list for version 1.1 only!

Check to make sure your kit comes with the following parts. Sometimes we make mistakes so double check everything and email support@adafruit.com if you need replacements!

Image	Name	Description	Part information	Qty
	IC1	3.3V linear voltage regulator, 250mA current	MCP1700-3302E/TO	1
	IC2	12-bit DAC	MCP4921	1
	IC3	High-current opamp	TS922IN or TS922AIN OR TLV2462	1
	IC4	Level shifter for SD card If you don't have this part you've probably got a v1.0kit. See the parts list below.	74xx125 (e.g. 74AHC125)	1
		SD/MMC card holder	Tyco 2041021-3	1
	TM1	10K or 50K Audio thumbwheel potentiometer. Includes pot, thumbwheel and tiny screw.	311-1204F-10K	1
	X1	Stereo headphone jack with switches.	STX-3100-5N	1
	R7	1/4W 5% 1.5K resistor Brown Green Red Gold	Generic	1
	R6	1/4W 5% 10K resistor Brown, Black, Orange, Gold	Generic	1
	R8	1/4W 5% 100K resistor Brown, Black, Yellow, Gold	Generic	1
	C8	0.01uF ceramic capacitor (103) May look deceptively like the 0.1uF ceramic capacitors! Lately has been shipped in an 'axial' (not 'radial' package. See instructions for details.	Mouser	1
	C2, C3, C5, C6,	0.1uF ceramic capacitor (104)	Generic	5

	C7	Looks deceptively like the 0.01uF ceramic capacitor!		
	C1, C4, C9	100uF / 6V or greater capacitor	Generic	3
	RESET	6mm tactile switch	B3F-1000	1
	ICSP	6-pin ICSP header	Generic	1
		36 pin male header (1x36)	Generic	1
	PCB	Circuit board with "v1.1" on it.	Adafruit Industries	1

Parts list for version 1.0 ONLY

Check to make sure your kit comes with the following parts.Sometimes we make mistakes so double check everything and email support@adafruit.com if you need replacements!

Image	Name	Description	Part Information	Qty
	IC1	3.3V linear voltage regulator, 250mA current	MCP1700-3302E/TO	1
	IC2	12-bit DAC	MCP4921	1
	IC3	High-current opmap	TS922IN or TS922AIN	1
		SD/MMC card holder	Tyco 2041021-3	1
	TM1	10K Audio thumbwheel potentiometer. Includes pot, thumbwheel and tiny screw.	311-1204F-10K	1
	X1	Stereo headphone jack with switches.	STX-3100-5N	1
	R1	1/4W 5% 1.0K resistor Brown, Black, Red, Gold If you don't have this part, you've probably got a v1.1 kit. See the parts list above.	Generic	1
	R7	1/4W 5% 1.5K resistor Brown, Green, Red, Gold	Generic	1
	R3, R5, R6	1/4W 5% 4.7K resistor Yellow, Purple, Red, Gold If you don't have this part, you've probably got a v1.1 kit. See the parts list above.	Generic	3

	R2, R4	1/4W 5% 10K resistor Brown, Black, Orange, Gold	Generic	2
	R8	1/4W 5% 100K resistor Brown, Black, Yellow, Gold	Generic	1
	C8	0.01uF ceramic capacitor (103) May look deceptively like the 0.1uF ceramic capacitors! Lately has been shipped in an 'axial' (not 'radial' package. See instructions for details.	Mouser	1
	C2, C3, C5, C6, C7	0.1uF ceramic capacitor (104) Looks deceptively like the 0.01uF ceramic capacitor!	Generic	5
	C1, C4, C9	100uF / 6V capacitor	Generic	3
	D1	3.6V Zener diode If you don't have this part, you've probably got a v1.1 kit. See the parts list above.	1N5227B	1
	RESET	6mm tactile switch	B3F-1000	1
	ICSP	6-pin ICSP header	Generic	1
		36 pin male header (1x36)	Generic	1
	PCB	Circuit board	Adafruit Industries	1

Solder

These are instructions for version 1.1 of the kit!

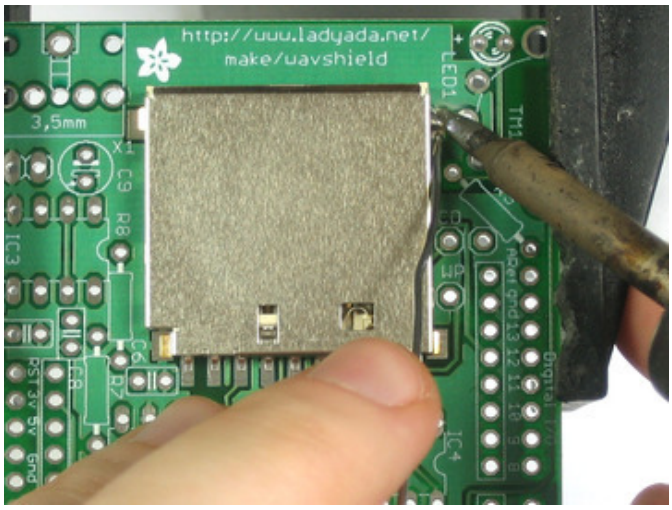
If you are confused because your kit doesn't have a 74HAC125 in it, you probably want to read the v1.0 instructions (https://adafru.it/c0u).

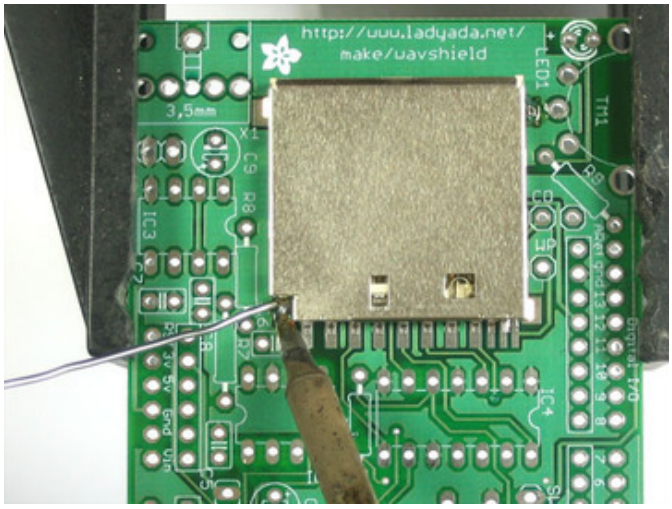
Make it

Get ready by placing the PCB in a vise.

We're going to the SD card first. While surface mount parts are a little tougher than thru-hole, this piece has pin spacing of 0.1" so it is quite easy. Doing it first also gives us lots of working room.

The holder should 'snap' perfectly into place thanks to two bumps on the bottom.

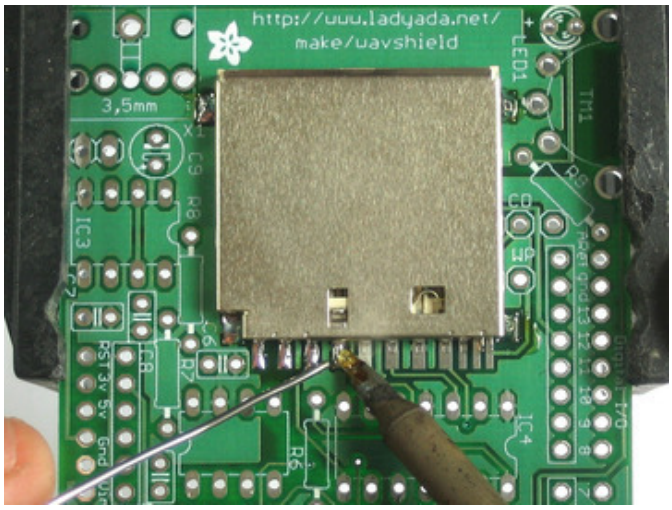


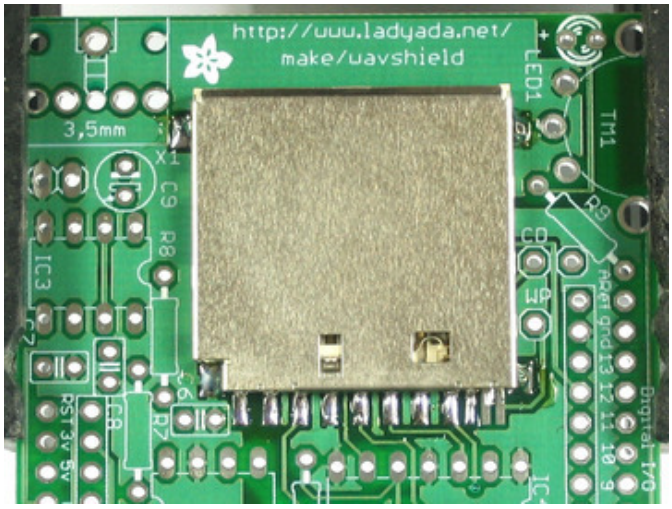


We'll start with the four side tabs that are used to mechanically secure the card holder in place.

Heat up the metal tab and the pad (the silver square beneath it) for 3 seconds with a hot soldering iron, then poke just a bit of solder in.

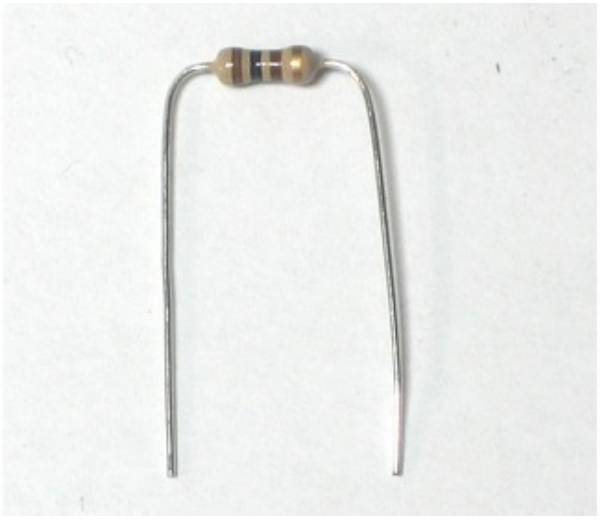
Do this for all three corners. Once this is done you should not be able to lift the card holder





Now go thru and solder the 8 leftmost pins that stick out from the holder. The three rightmost pins are thinner and closer together so they are tougher to solder. Luckily they are not used and you simply skip them (although the photo shows them done).

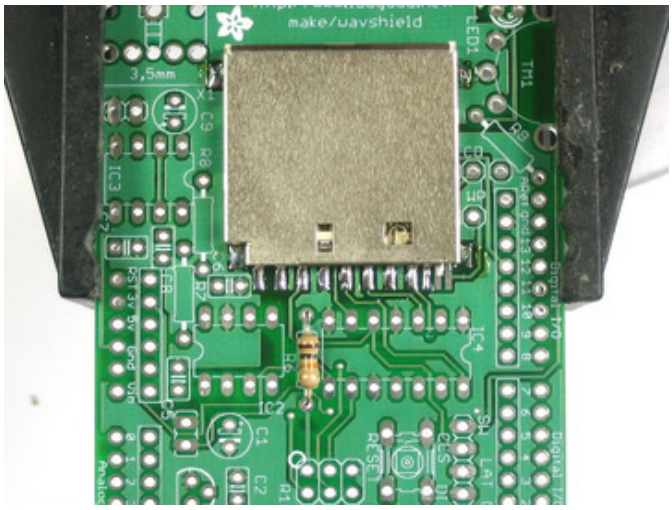
Check that you have no solder bridges - the pins should not be soldered to the metal body of the holder or to each other.

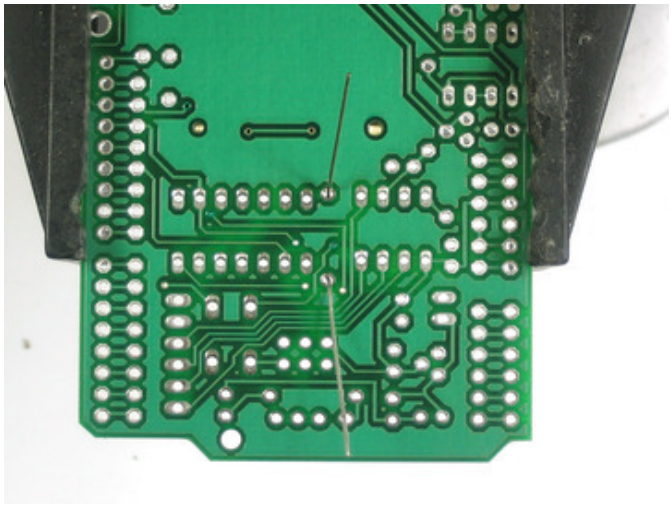


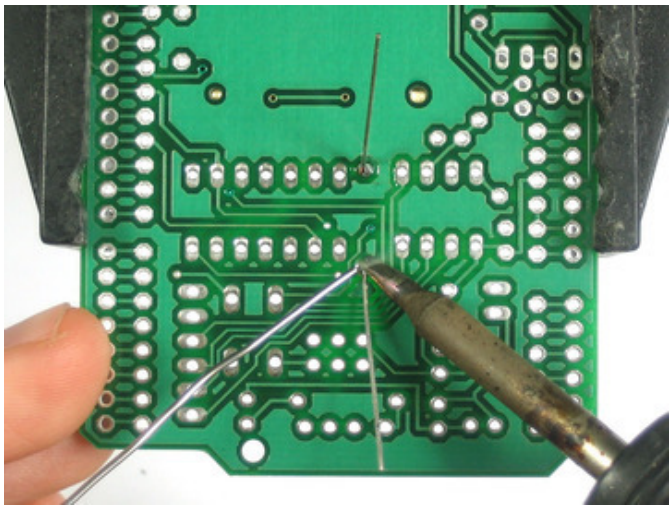
Next, we will solder all of the many resistors. The 10K resistor **R6** is first.

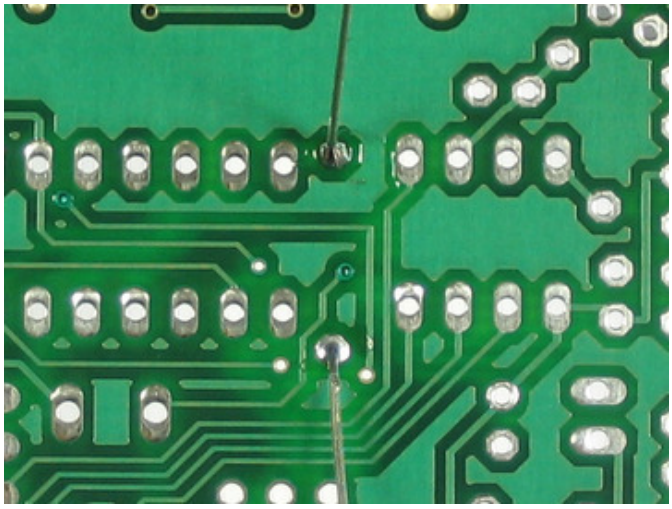
Form them into staples (as shown left with a 100 ohm resistor), then place them so they sit flat against the PCB, in the correct locations. Resistors don't have *polarity* so they can go in 'either way' and work fine!

Once placed, bend the leads out so the resistors don't fall out.

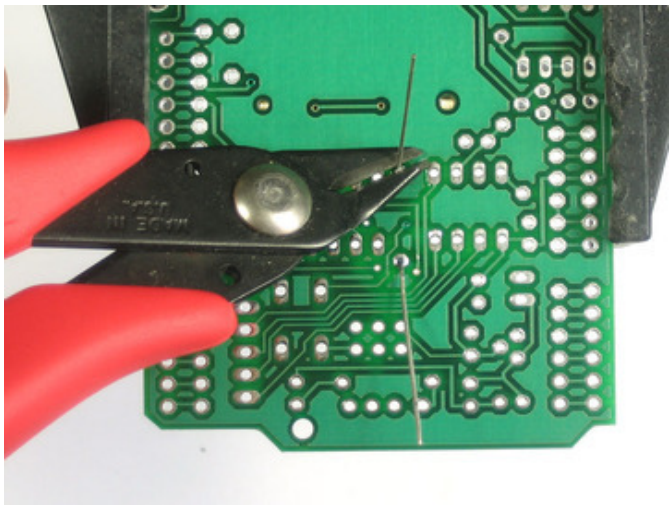




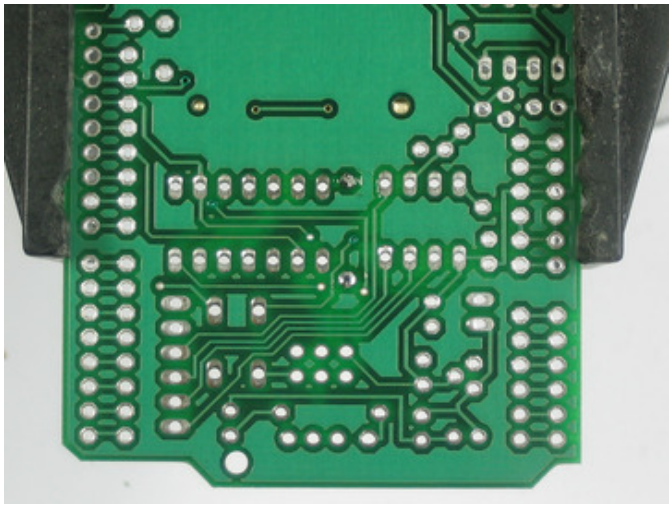


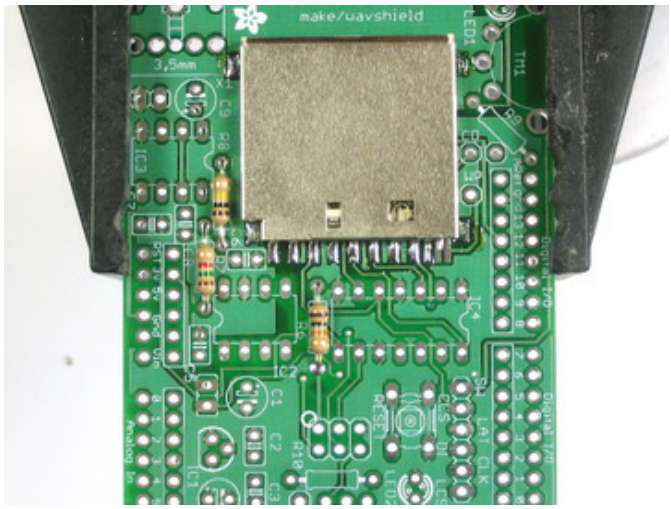


Solder the leads to the pads (metal ring) by heating both with the side-tip of the iron for 3 seconds and then poking in a bit of solder.



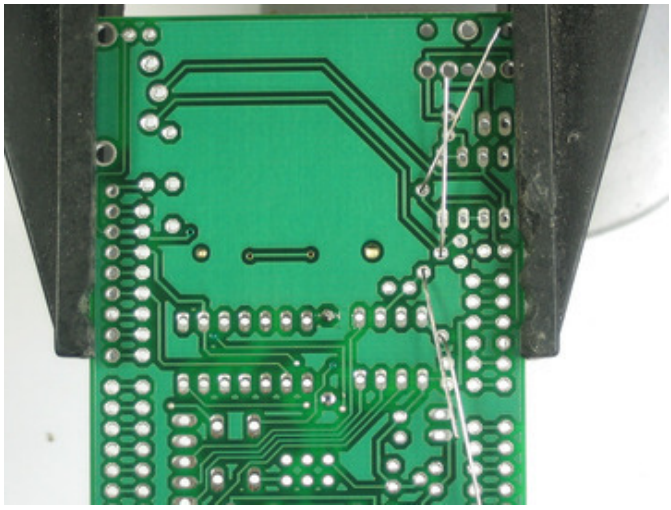
Use your diagonal cutters to clip the leads off just above the solder joint.

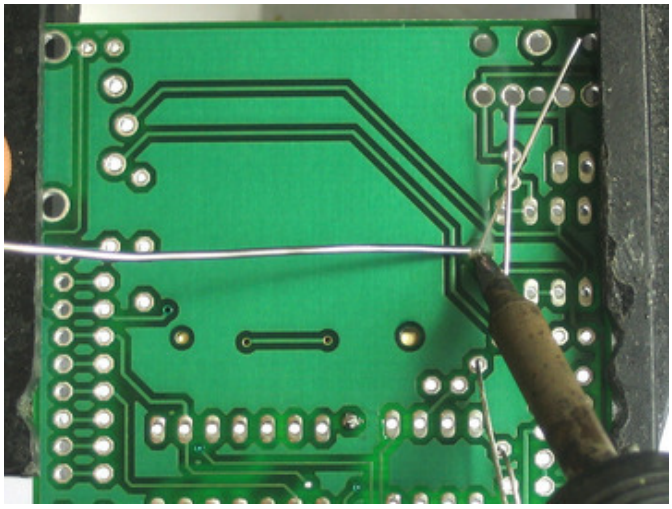


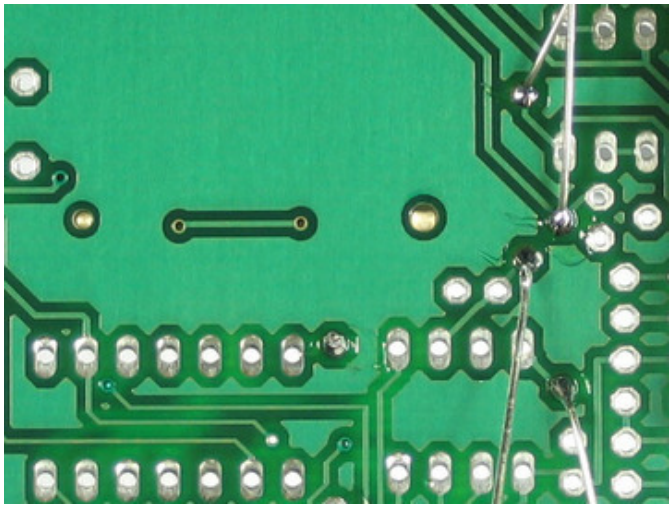


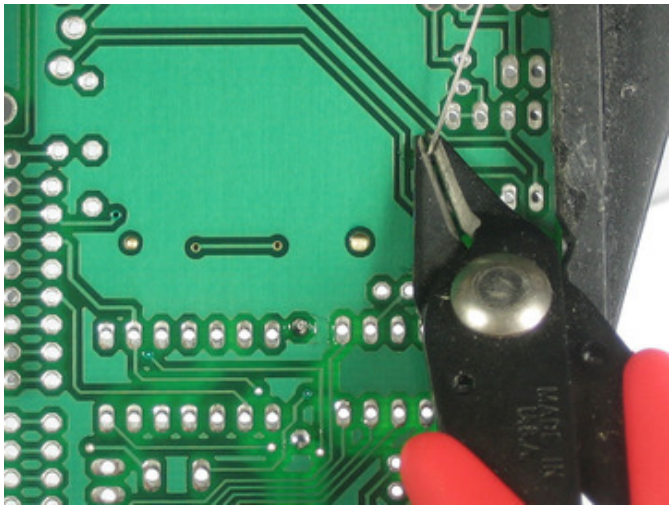
Finish up the resistors by placing **R8** (100Kohm), and **R7** (1.5K)

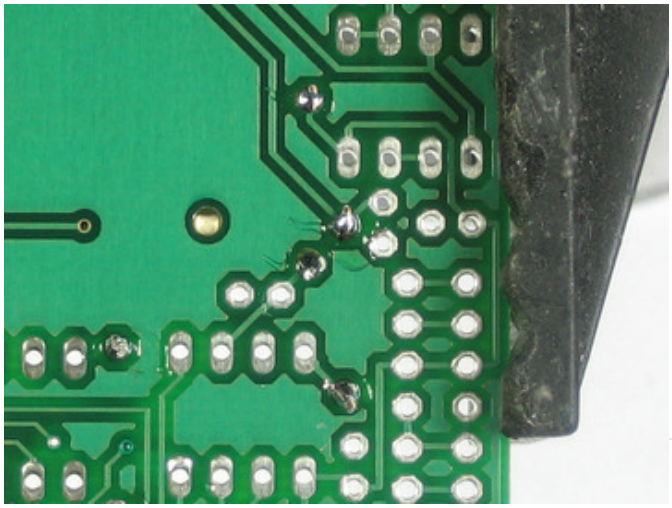
Solder the components.







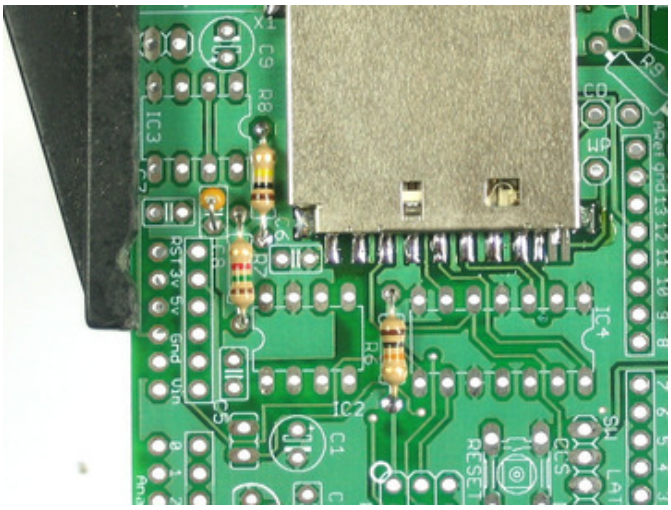




Next is the 0.01uF ceramic capacitor **C8**. The tricky part here is that in older kits there are many 0.1uF ceramic capacitors in the kit that look identical to the 0.01uF!

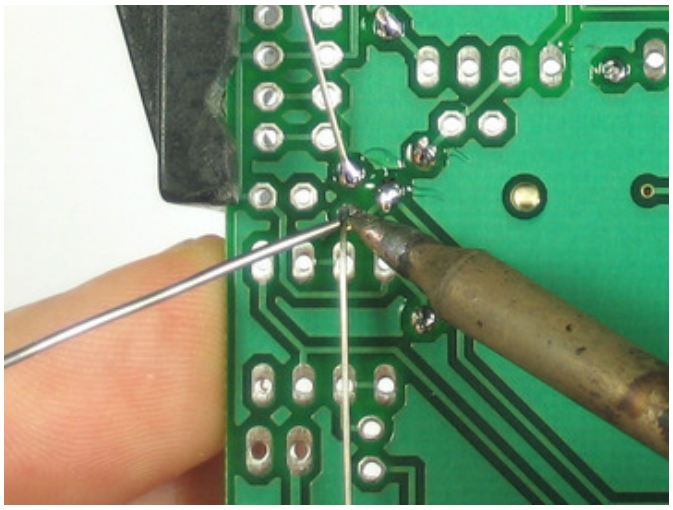
The way to tell the difference is look for the **103** printed on it. If it says **104** then it's a 0.1uF. Make sure it says **103**! This capacitor forms the output low-pass filter for the audio so its important to have the right value.

Lately I have been shipping kits with *axial* (long-ways) package, not *radial* (side-ways) package. These are longer (see left) and are easy to bend over for soldering. This way there is less confusion. Either way, try to spot the **103** marking.

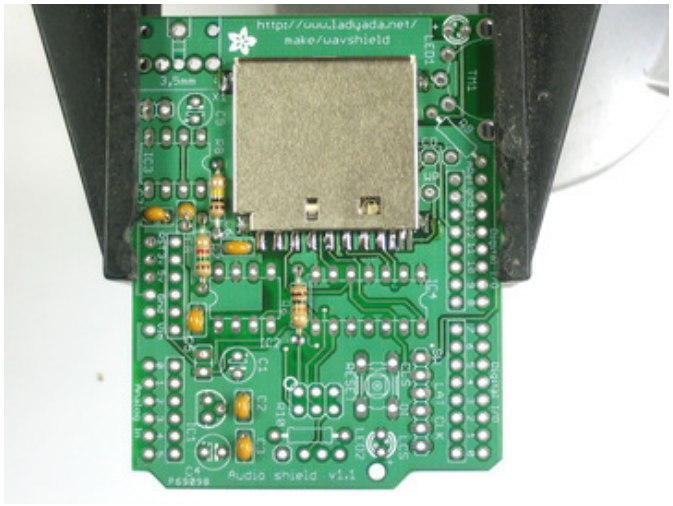


Place the capacitor right next to R7.

Ceramic capacitors are non-polarized and can go in 'either way.'



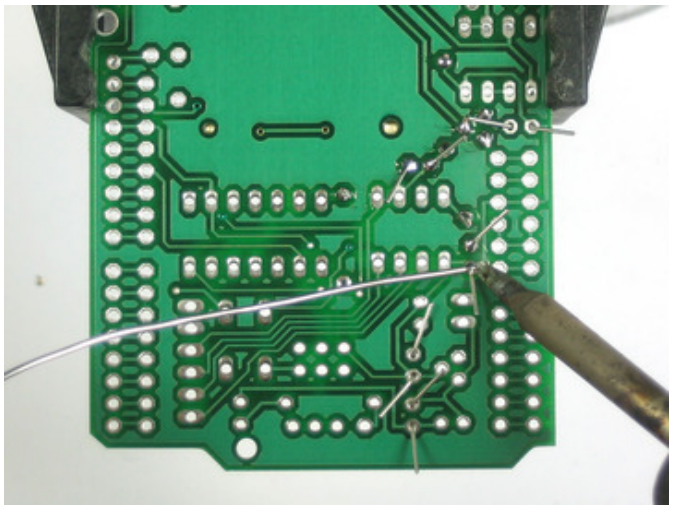
Solder and clip the small capacitor leads.



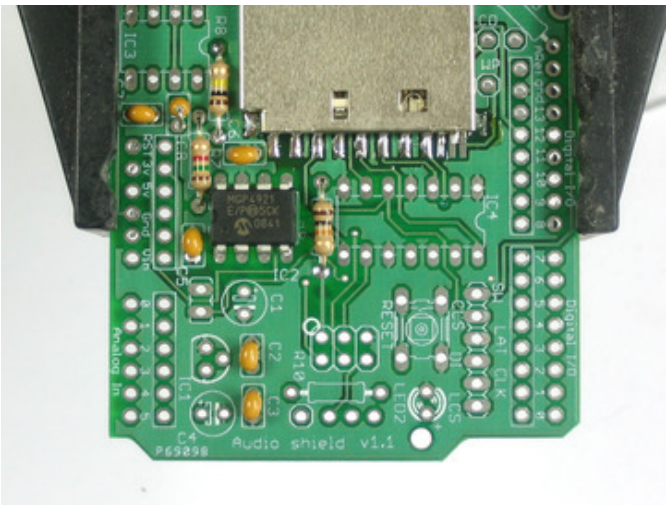
Once you're sure you have **C8** correct, you can place the remaining 0.1uF ceramic capacitors **C2, C3, C5, C6** and **C7**.

Ceramic capacitors are non-polarized and can go in 'either way.'

Note carefully where C5 goes, it doesnt go below C1 but rather next to the 1.5K resistor

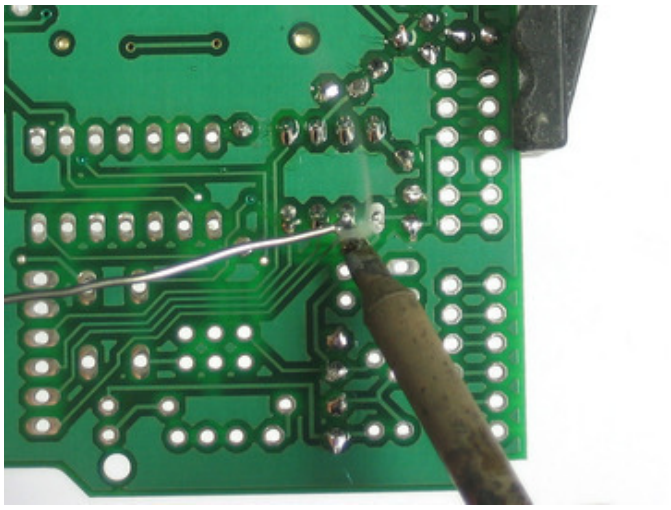


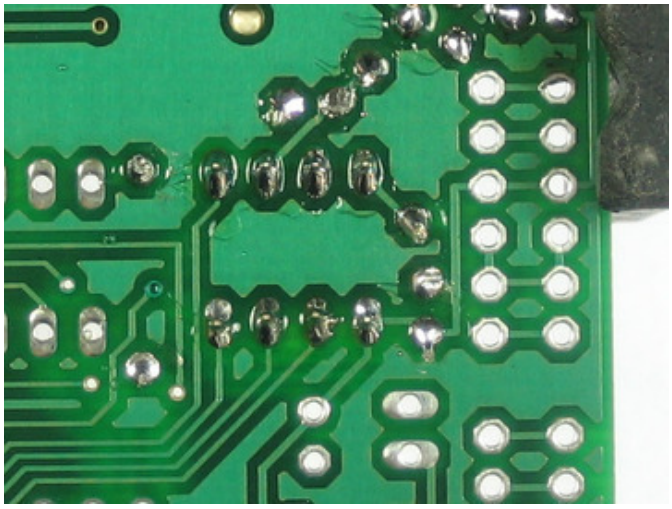
Solder and clip the capacitors.



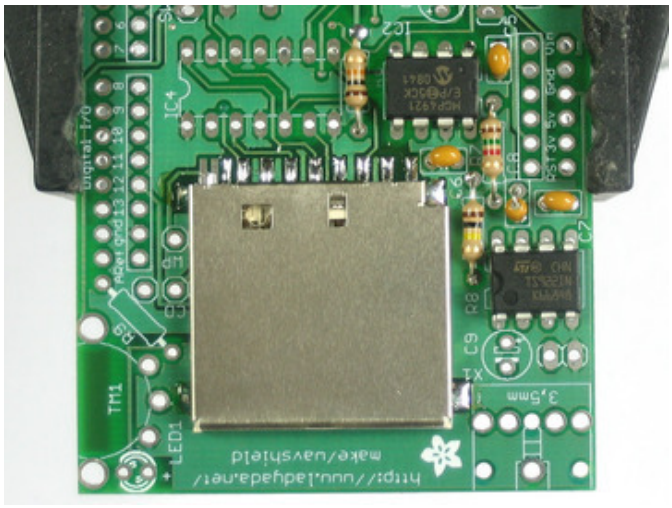
Next is the DAC (digital-analog converter) **IC2**. This is what turns the data into music. Make sure you pick the DAC to solder in here, it says MCP4921 on it and has a stylized M.

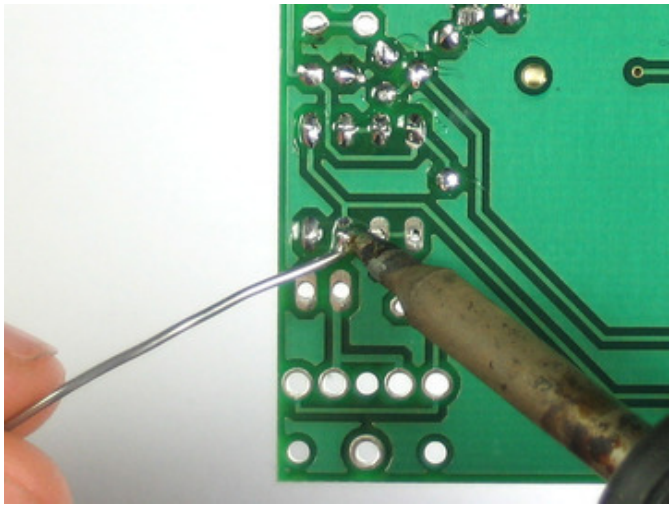
The chip has a notch in one end and that notch must line up with the notch in the silkscreen. In this photo, thats on the left.





Flip over the board and solder in each pin of the chip. The pins are already quite short so you dont have to clip them.



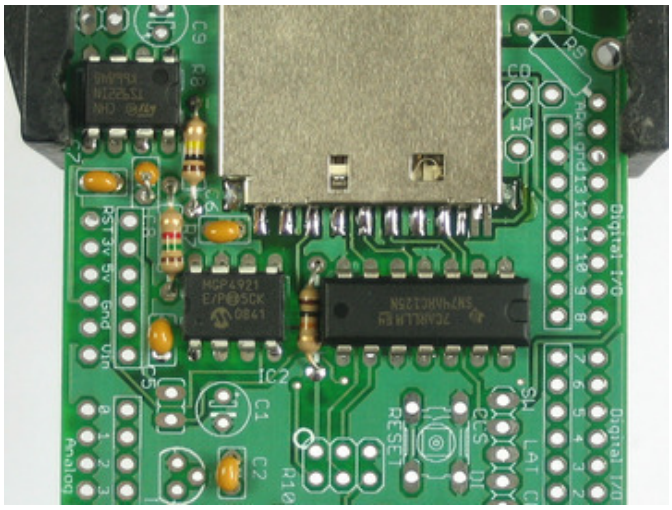


Next is the operational-amplifier (op-amp) **IC3**. It is used to buffer and amplify the output, so that it can drive a small speaker or headphones.

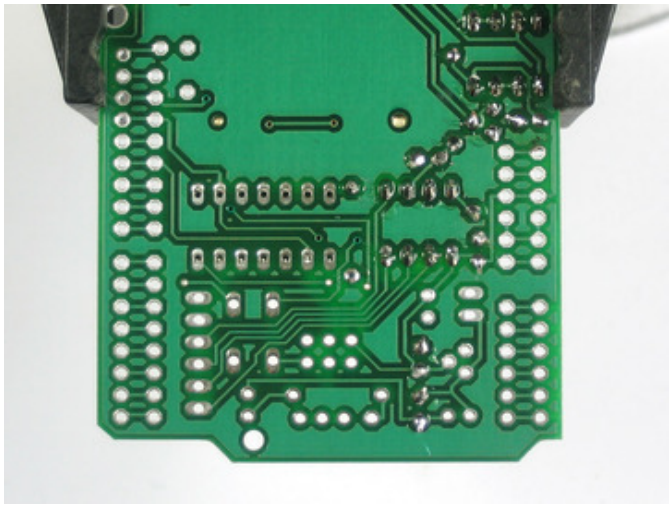
This chip may be labeled TS922 **or** TLV2462

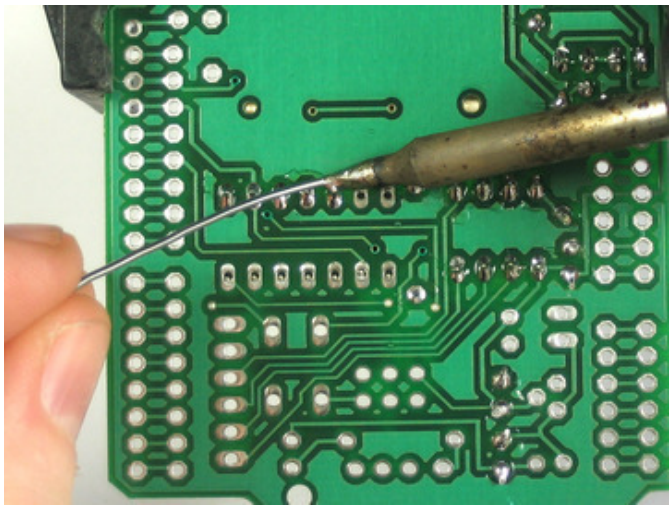
This is a similar-looking chip to the DAC. Again, check that the notch matches the silkscreen notch. In this photo, thats to the left.

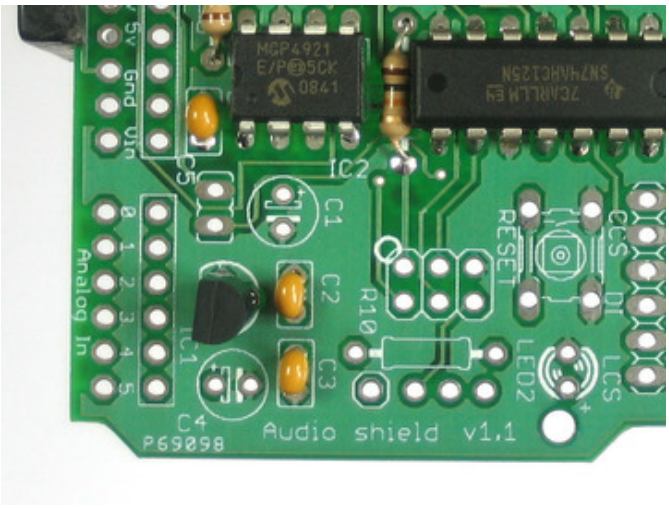
Solder it in, just like you did with the DAC.



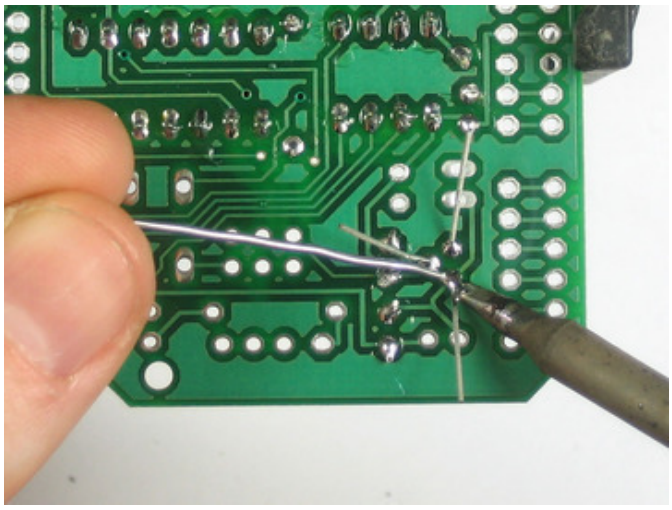
Next is **IC4**, the buffer to talk to the SD card. Match up the notch just like you did with the smaller chips.

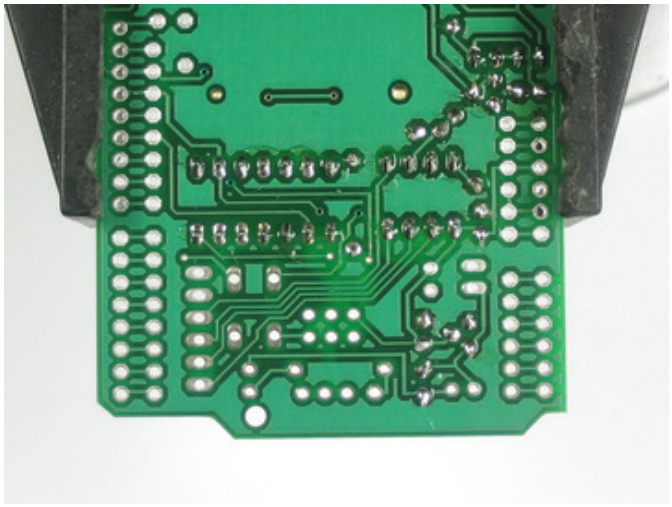




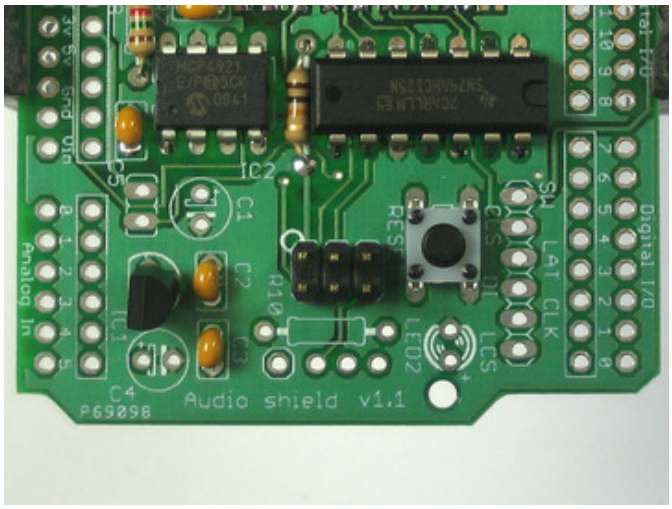


Next is the 3.3V regulator **IC1** that provides a nice powersupply to run the SD card. The regulator comes in a semi-circular package, so make sure it matches up with the silkscreened image.



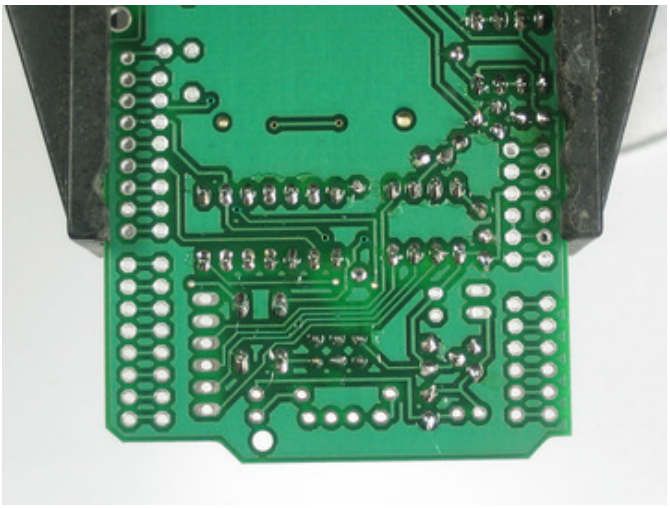


Turn the board over and solder/clip the three leads.

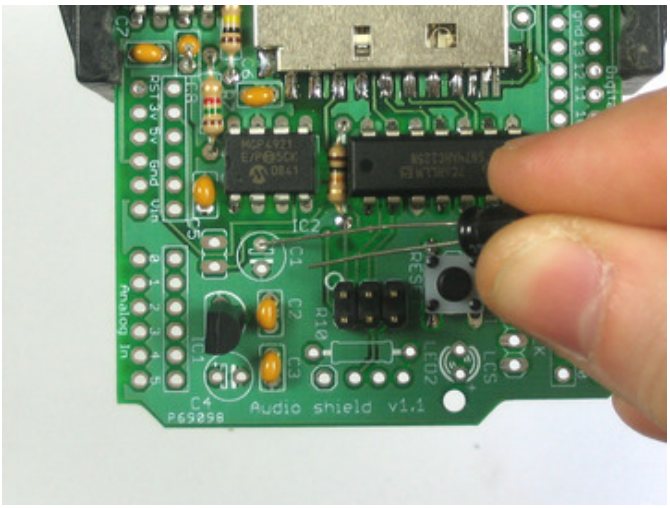


Next is the reset button and the ICSP header. These let you reset the Arduino manually, and reprogram it directly with a AVR programmer.

The button will snap in, its symmetric so it goes in 'either way'. The header is also symmetric, make sure the long end sticks up.

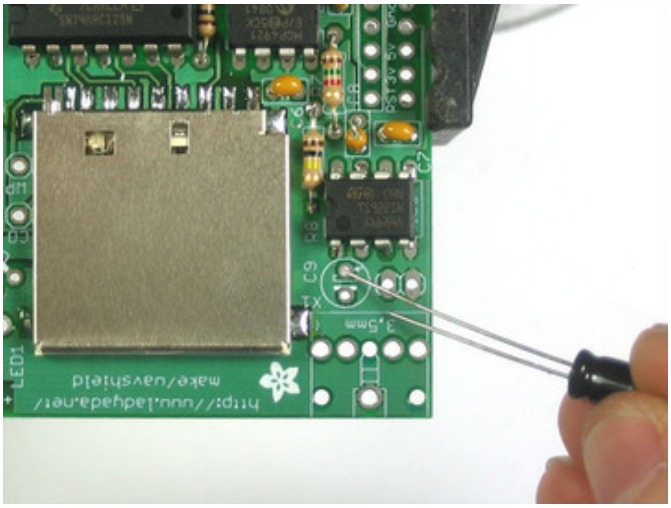


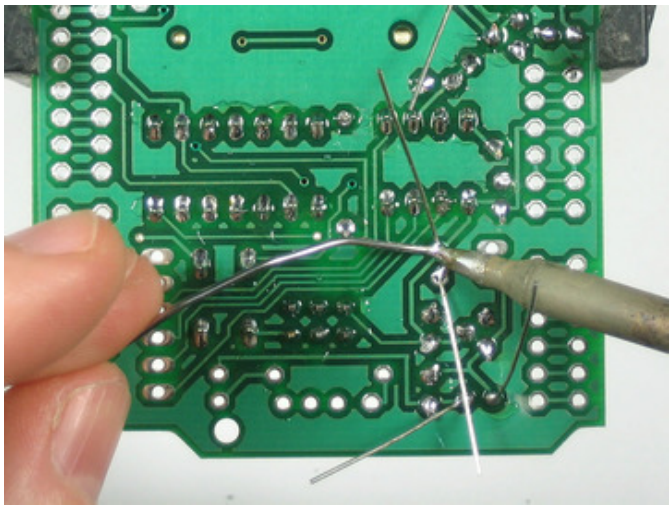
Solder in both components. Their leads are pretty short so you dont need to clip them.



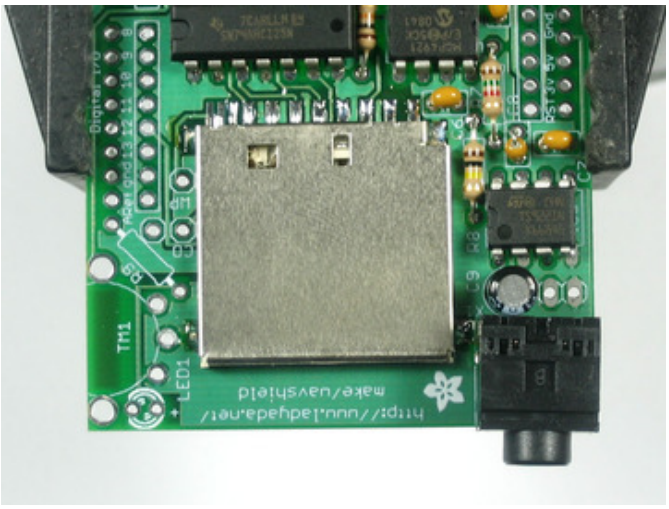
Next are the three electrolytic capacitors **C1 C4** and **C9**.

Electrolytic capacitors are polarized so make sure they go in the right way! The long lead is the positive lead, make sure that goes into the hole marked with a + as shown here.

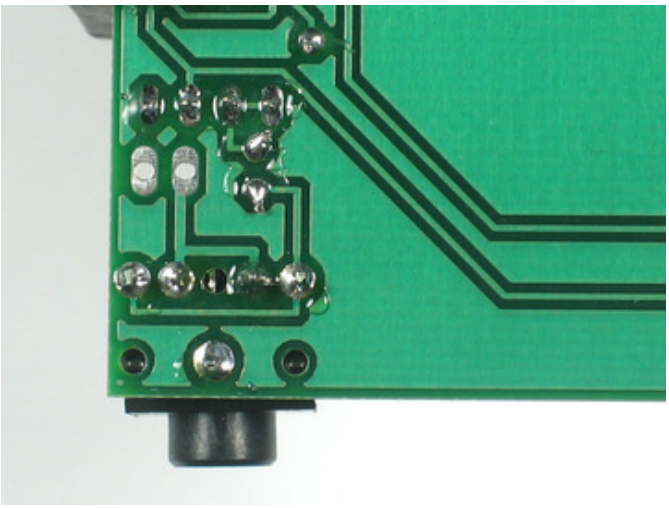




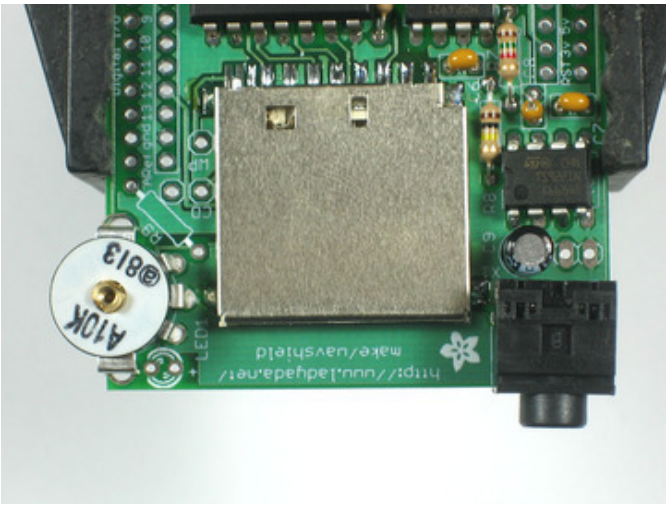
Solder them in.



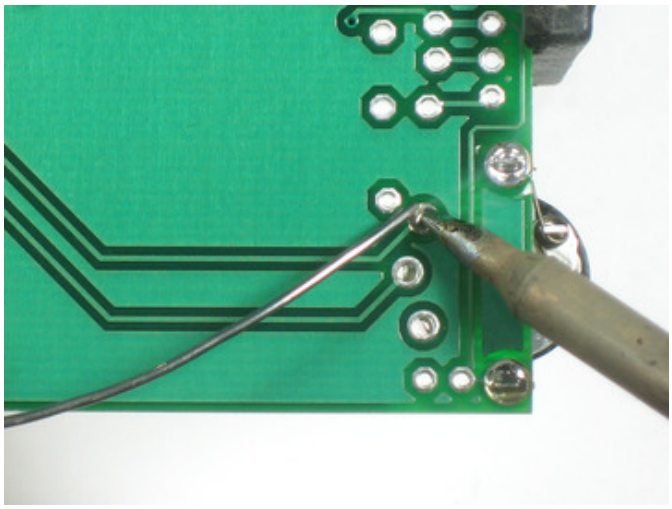
Next is the headphone jack. It snaps into place right at the edge of the PCB.



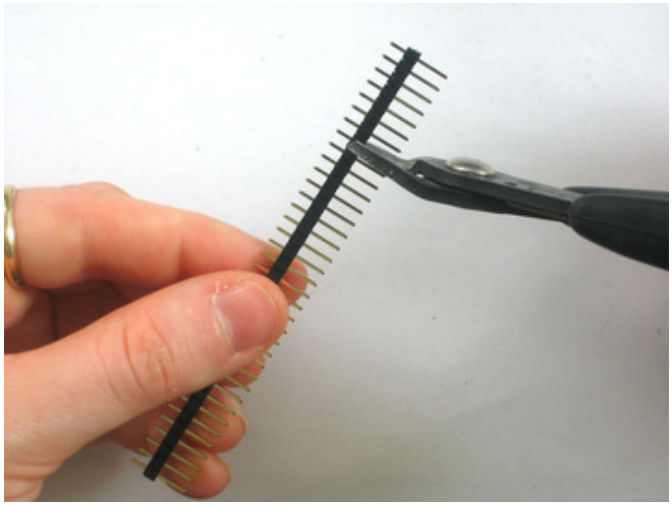
Solder the jack in place. You'll want to clip the legs a little if you can, so that it will sit better on the Arduino.

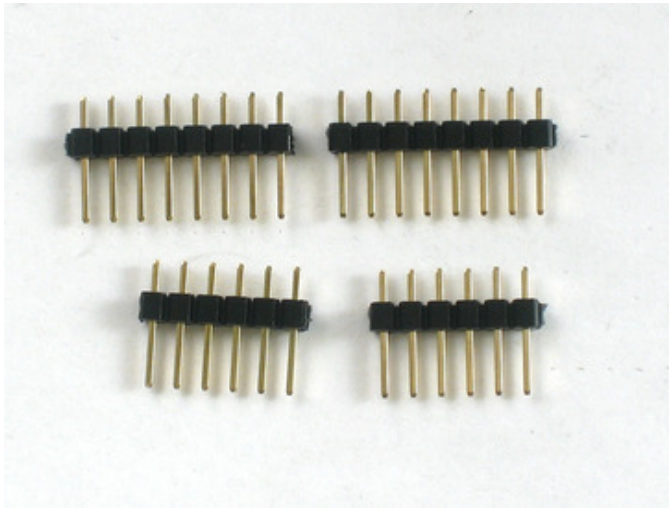


Next is the volume potentiometer **TM1**. This is an audio-type 10K pot. It will slip into place pretty easily.

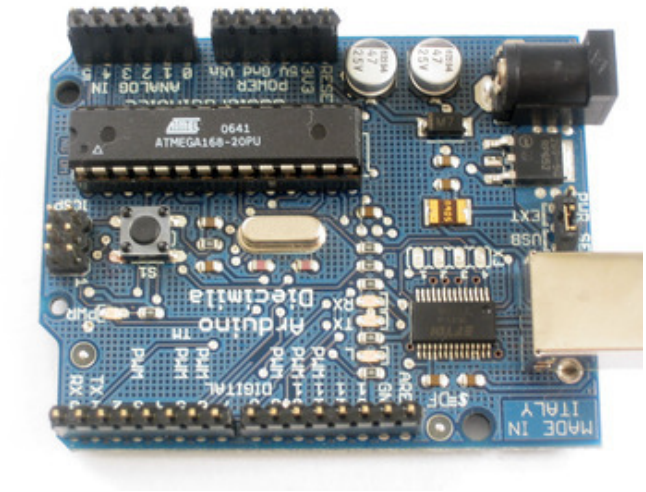


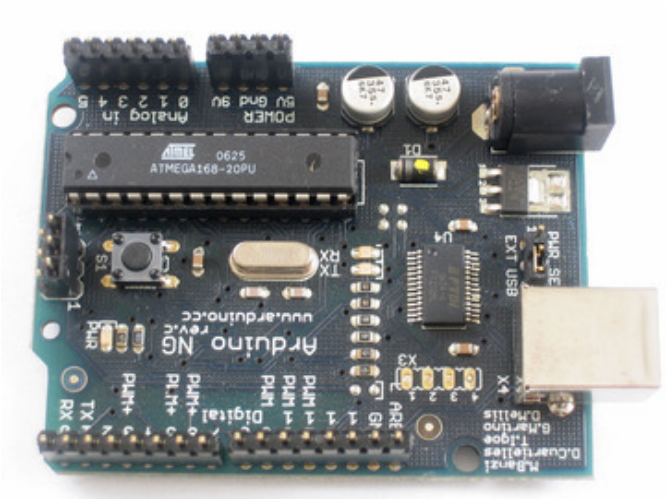
Solder all 5 pins of the potentiometer. Use plenty of solder so that it has a lot of mechanical strength.





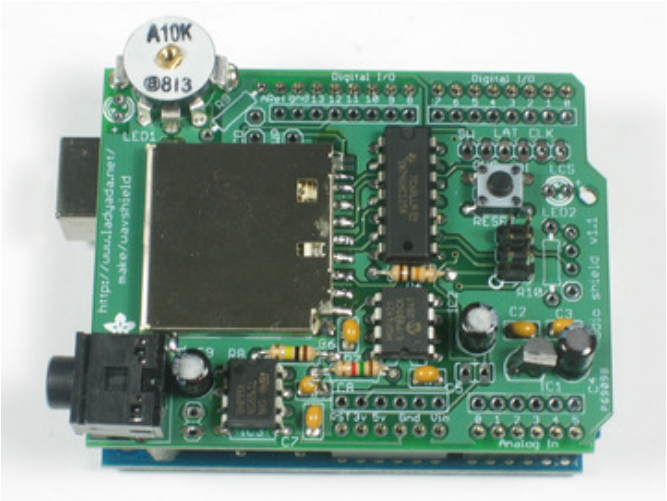
Next, break the 36-pin header strip into smaller sections so that the shield can be placed on the Arduino. You can use pliers or diagonal cutters. Clip off 2-6pin and 2-8pin pieces.



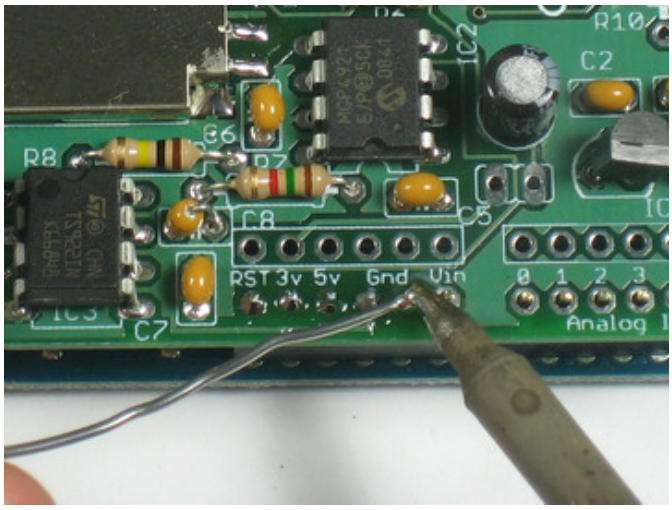


If you're using a Diecimila, Duemilanove, Uno or later Arduino, place the 6 and 8 pin headers into the female sockes.

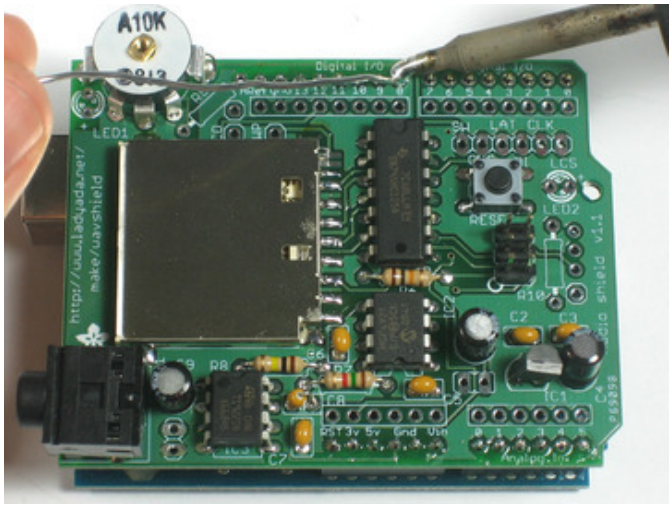
If you have an NG Arduino, you can place a 3-pin female header (not included) as shown, which will let you use the reset button.

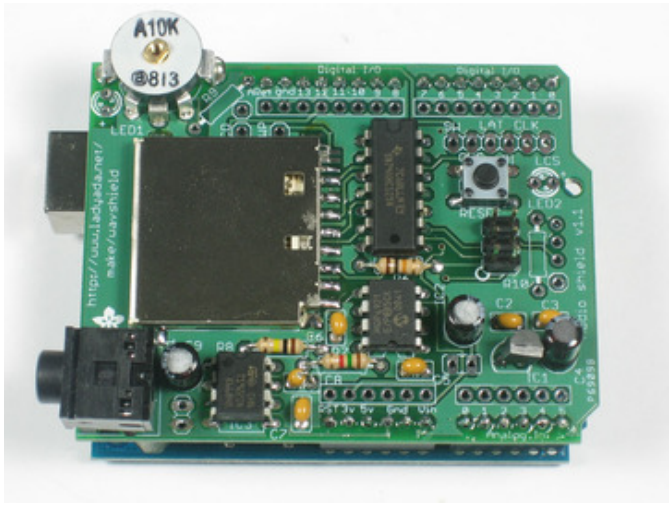


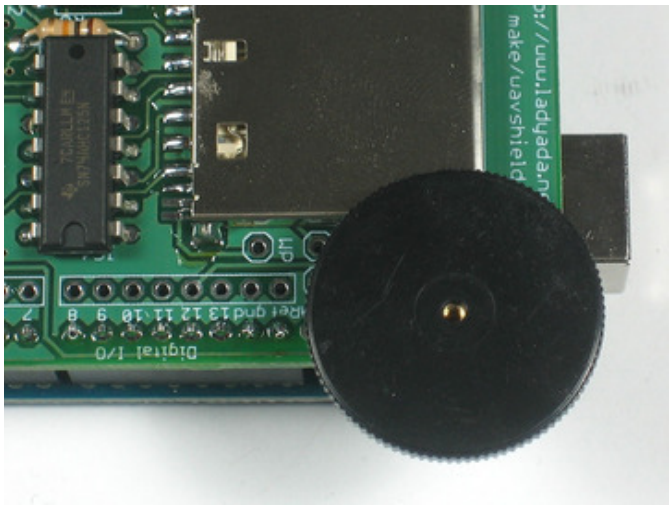
Place the shield PCB onto the arduino so that all the holes match up with the header.



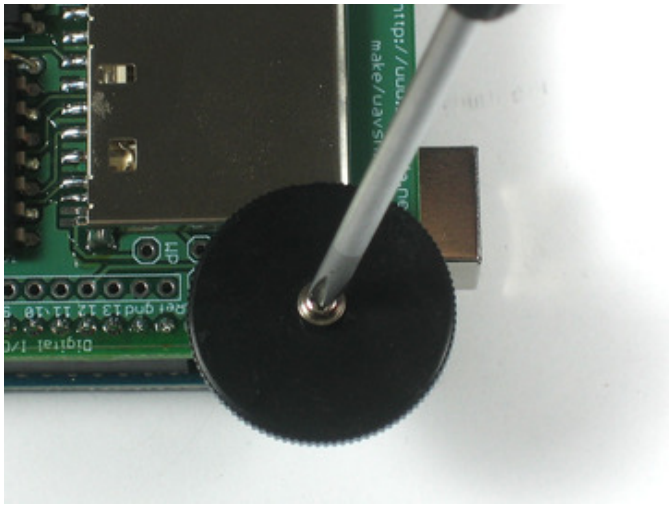
Solder in each and every pin of header.

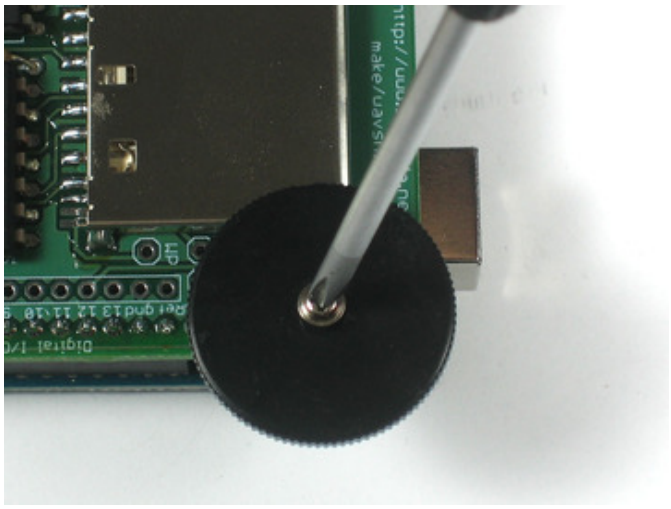


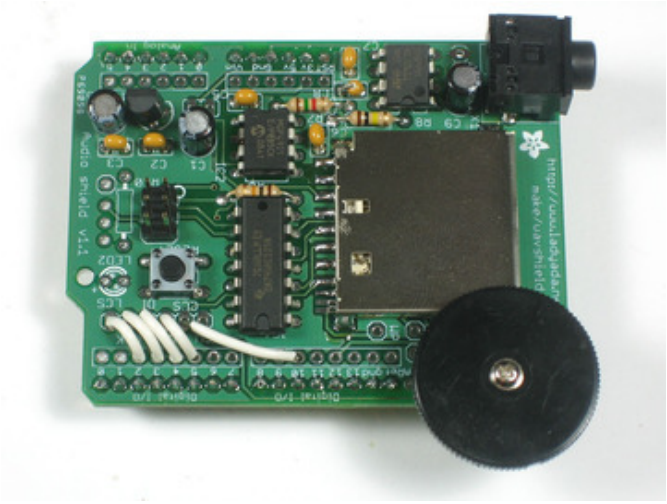




Next you can install the thumbwheel. Use a #0 screwdriver. Align the thumbwheel so it 'grabs' the potentiometer, then gently screw it in place.



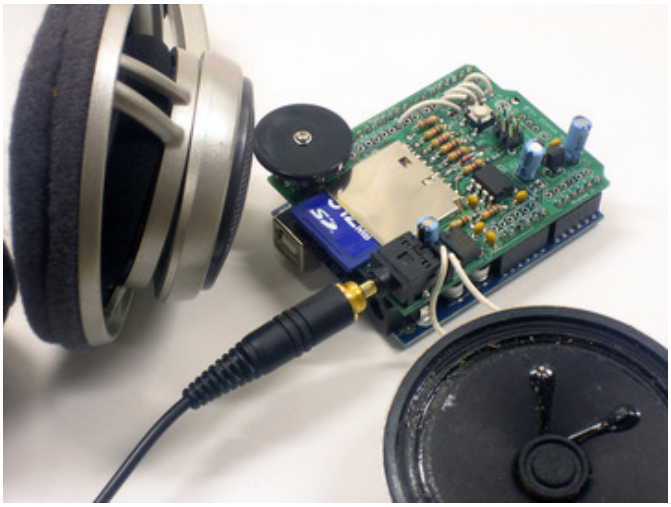




Pins 13, 12 and 11 are used to talk to the SD card and cant be changed. The rest of the pins, however, are more flexible. Still, for all the examples on the site we'll be using this wiring, so it is suggested to just go with this.

2 -> LCS
3 -> CLK
4 -> DI
5 -> LAT
10 -> CCS

You can use any sort of wire. Solder the jumper wires in place.



Hooray you are done! Now onto the user manual...

Use it!

How to use it

Once you've got your wave shield assembled and tested you can now customize it as desired, See the next 6 steps for instructions on

1. Format an SD/MMC card
2. Convert audio files so they are suitable for playing
3. Check out documentation for WaveHC library
4. Check out walkthroughs for a digital audio player and 6-button audio player
5. Try out some of the example sketches.

SD Card

Introduction

The wave shield uses SD/MMC cards. They are extraordinarily popular, sometimes even available in grocery stores! They are used in MP3 players, cameras, audio recorders, etc. You can use any card that can store 32 MB ore more. A 4 gigabyte card can hold 25 hours of uncompressed audio for the shield, and costs $12 (http://adafru.it/102).

The shield kit doesn't come with an SD card but we carry one in the shop that is guaranteed to work (http://adafru.it/102). Pretty much any SD card should work but be aware that some cheap cards are 'fakes' and can cause headaches.

You'll also need a way to read and write from the SD card. Sometimes you can use your camera and MP3 player - when its plugged in you will be able to see it as a disk. Or you may need an SD card reader.

These are very common, available in any computer store. The shield **doesn't** have the ability to display the SD card as a 'hard disk' like some MP3 players or games, the Arduino does not have the hardware for that, so you will need an

external reader!

To use the SD card interface library you'll need a '328 Arduino. If you have an NG or '168 chipped Arduino we suggest upgrading to a '328 (https://adafru.it/aIH). It's easy and inexpensive and you'll be very happy with the 2x RAM and Flash. All UNO's have Atmega328 chips.

Formatting under Windows/Mac

If you bought an SD card, chances are it's already pre-formatted with a FAT filesystem. However you may have problems with how the factory formats the card, or if it's an old cart it needs to be reformatted. The Arduino SD library we use supports both **FAT16** and **FAT32** filesystems. If you have a very small SD card, say 8-32 Megabytes you might find it is formatted **FAT12** which isn't supported. You'll have to reformat these card. Either way, it's always a good idea to format the card before using, even if it's new! Note that formatting will erase the card so save anything you want first.

We strongly recommend you use the official SD card formatter utility - written by the SD association it solves many problems that come with bad formatting!

https://adafru.it/c73

https://adafru.it/c73

Download it and run it on your computer, there's also a manual linked from that page for use.

Convert files

Intro

Small microcontroller audio project like CircuitPython or the Adafruit Wave Shield are designed to play a very specific type of audio. If your music sample is in MP3 format, or 44KHz wav, you'll want to convert it to the right format. This way you will get the best sounding audio, and makes your code easy!

This guide will show how to convert your music of FX into **PCM 16-bit Mono Wave files at 22KHz samplerate**

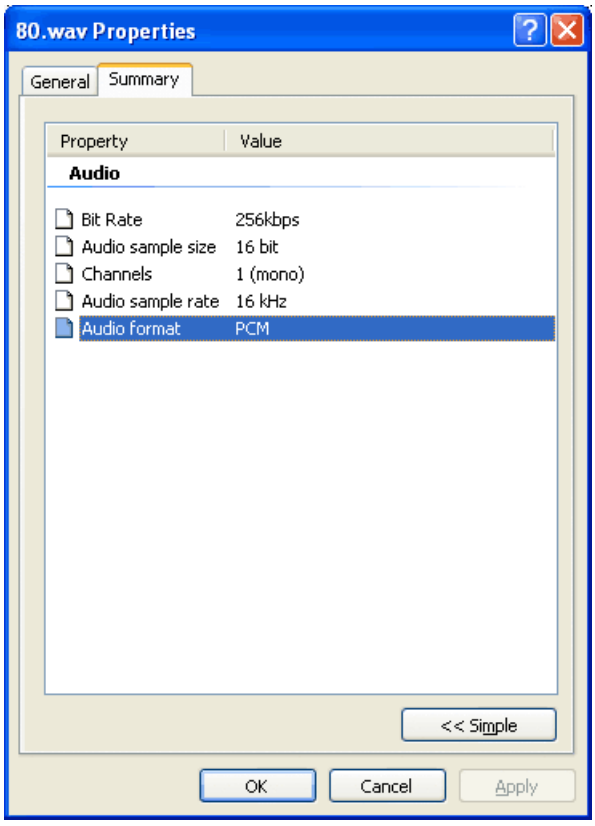
If you are using a Wave Shield, note that Arduino does not support "long filenames" - so if you have a file called, say, "My favorite song.wav" it will show up as "MY FAV~1.WAV" in the shield, which can be confusing. To make your brain hurt a little less, rename your files to 8.3 all-caps format so for example, "My favorite song.wav" -> "MYFAVSNG.WAV"

CircuitPython is fine with long file names, but still, we recommend less than 32 characters long names

Check the file

In Windows 7, if you have a wave file already, you should check to see if it's already in a proper format. That way you will save yourself some time! In Windows, right-click on the file, and select **Properties** then click on the **Summary** tab.

Note: This doesn't work under WIndows 10 unfortunately. You'll probably need to put the file into one of the conversion programs and just change it to ensure it works well.



This file is 16KHz, 16-bit, mono PCM. Since thats below the maximum (22KHz, 16-bit, mono PCM) you are good to go. No need to convert the file.

Ok, let's say the file is an MP3 or 44KHz or stereo wave file. Or that you don't know what the file is. We will need to convert it down.

The next two sections of this guide shows two ways in which to convert files. There are also other programs on various platforms that perform the same type of conversion but we don't have a step-by-step for such programs.

Convert Sound Files in Audacity

Small microcontroller audio projects are designed to play very specific types of audio files. If your music sample is in MP3 format, or 44KHz .wav or if you have grabbed a sample from a source (or the Internet) and do not know how it was encoded, you'll want to convert it to the right format.

This way you will get the best sounding audio, and it'll make using audio with your code painless!

This page will show how to convert your sound file(s) into **PCM 16-bit Mono WAV files at 22KHz sample rate**, which is usually best for the current crop of microcontrollers which take WAV files and play them on a speaker.

Currently, we are recommending two ways to convert the files: via use of the program Audacity or via iTunes. Other software may do similar conversions, just remember the parameters above when you do the conversion.

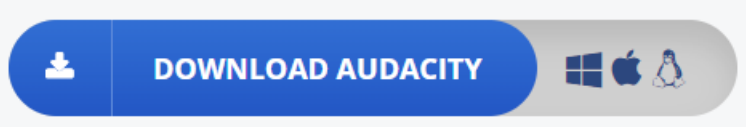
Audacity

The open-source software program Audacity is available for PC/Mac/Linux and is very easy to use to check files and convert the files if necessary.

Audacity can also be used to trim audio files to just a small clip you might want as a phrase or sound effect. The Audacity online manual (https://adafru.it/Bwy) has information on doing the trimming and more.

Download Audacity

Go to https://www.audacityteam.org/ (https://adafru.it/Bof) and click the **Download** button:

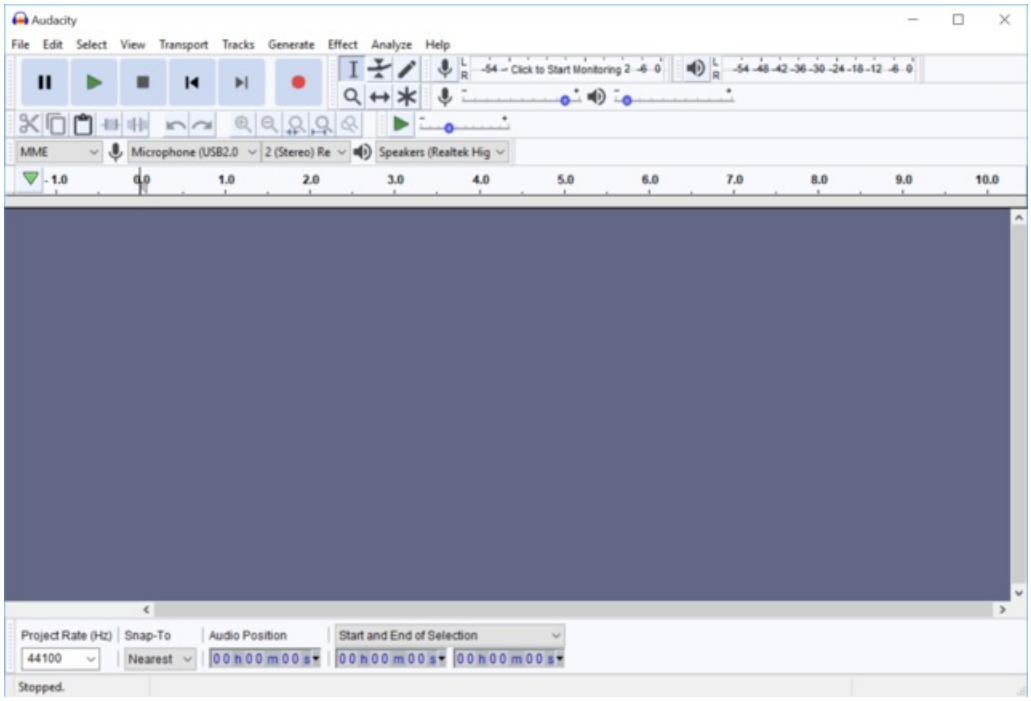


Select your operating system and download. For Windows, use the Windows Installer.

Run the downloaded installer program and then start Audacity.

(On macOS, double-click on the downloaded .dmg file, then drag the Audacity.app file to your Applications folder, then double-click the Audacity.app to launch.)

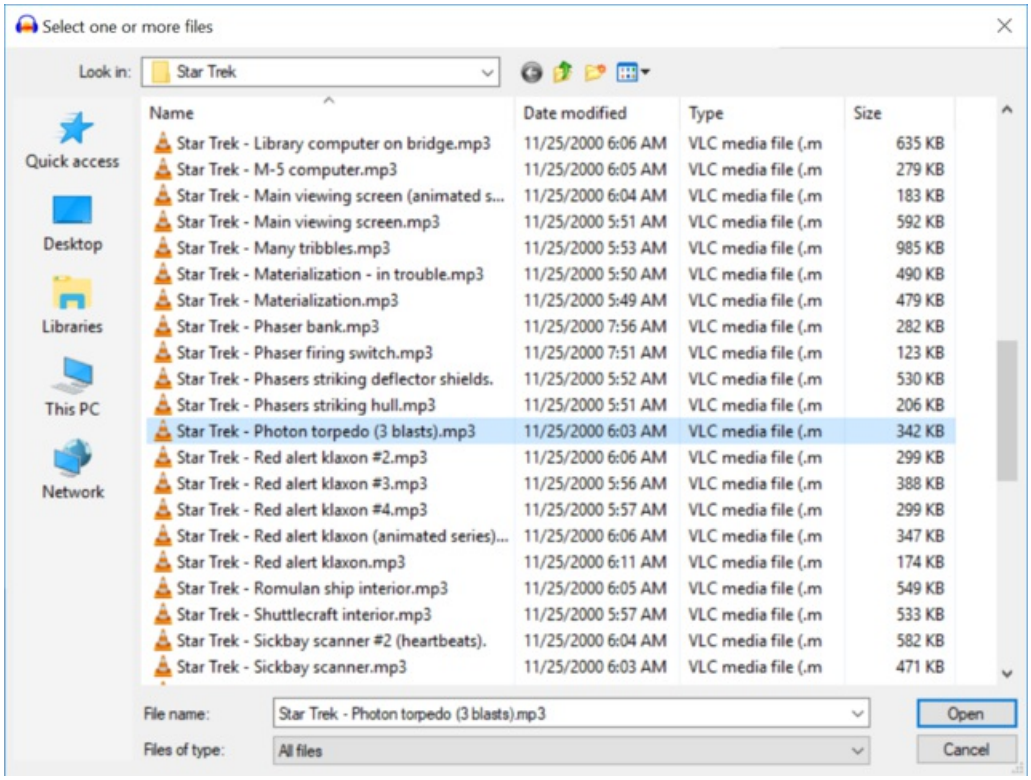
You should see a program window similar to the one below:



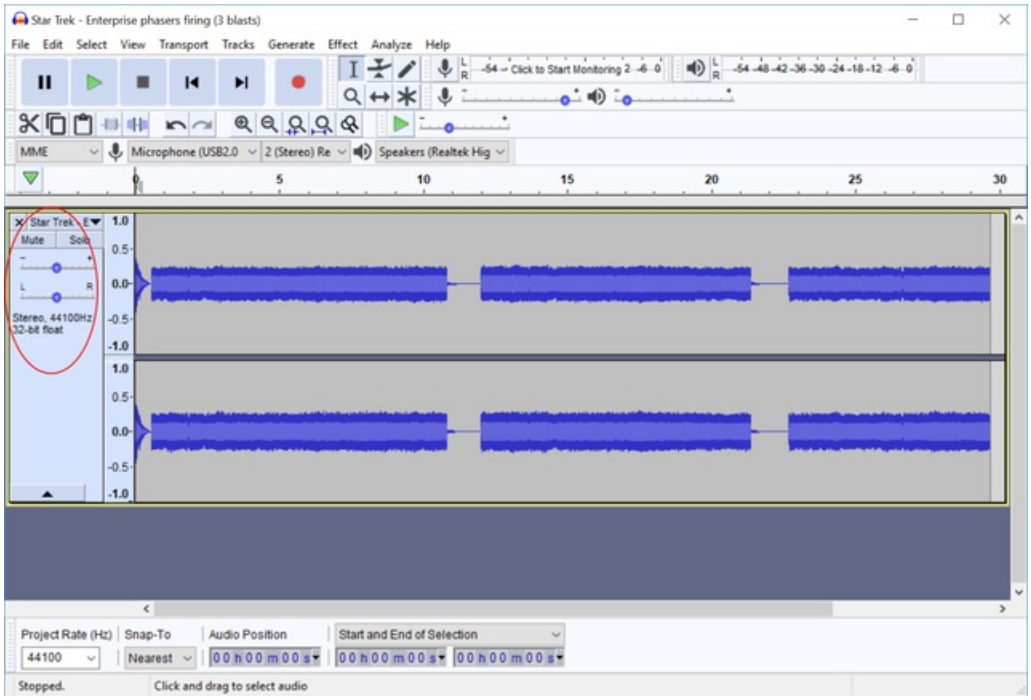
Step by Step Conversion

Step 1: Open the File

Select **File -> Open** to open your sound file. Audacity can open a wide range of files including MP3 and many obscure formats besides WAV (which is great!).



Step 2: Check the File Properties



See the red circled text. That shows that the WAV file I chose is a 44.1 KHz, 32 bit stereo file which is above our microcontroller specifications noted (22 KHz, 16 bit, mono). We'll need to tell Audacity what parameters we want to convert the file.

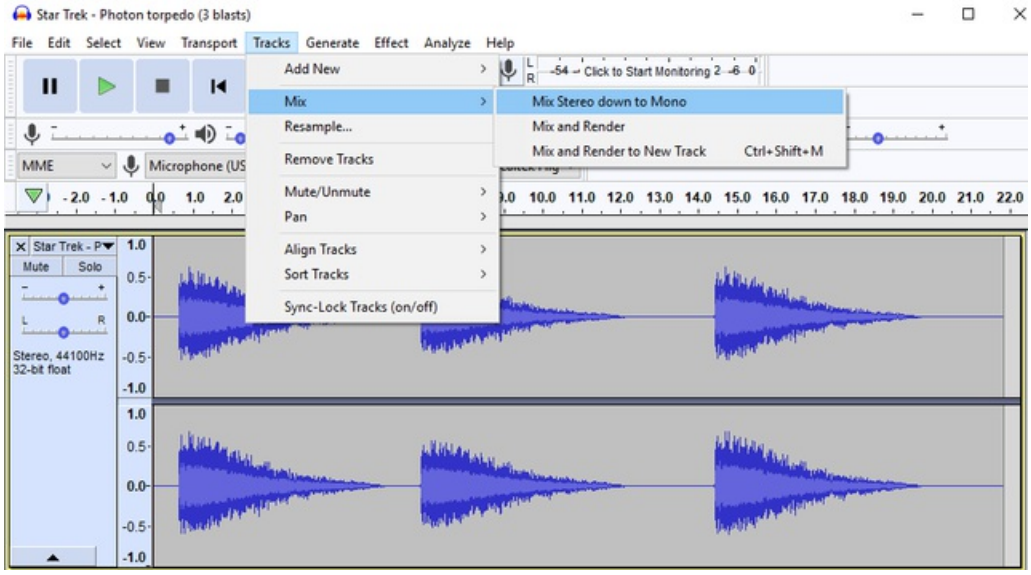
Step 3: Split and Mix Stereo Sound (if necessary)

If Audacity reports that you have a stereo file, you will have two blue waveform files. If you only have one waveform and Audacity says you have a mono file, skip this step.

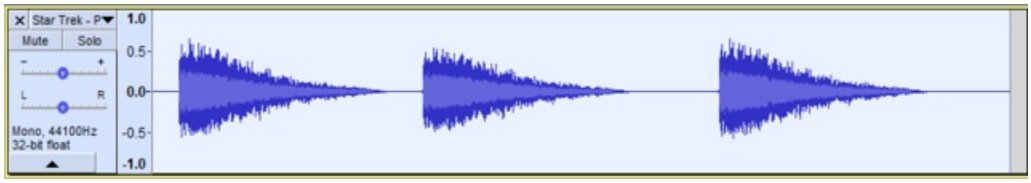
Combining a stereo track into a mono track will produce the original best on a mono speaker.

In Audacity 2.2, this is easier than in previous versions, it's one step instead of several:

Click on the menu item Tracks -> Mix -> Mix Stereo Down to Mono



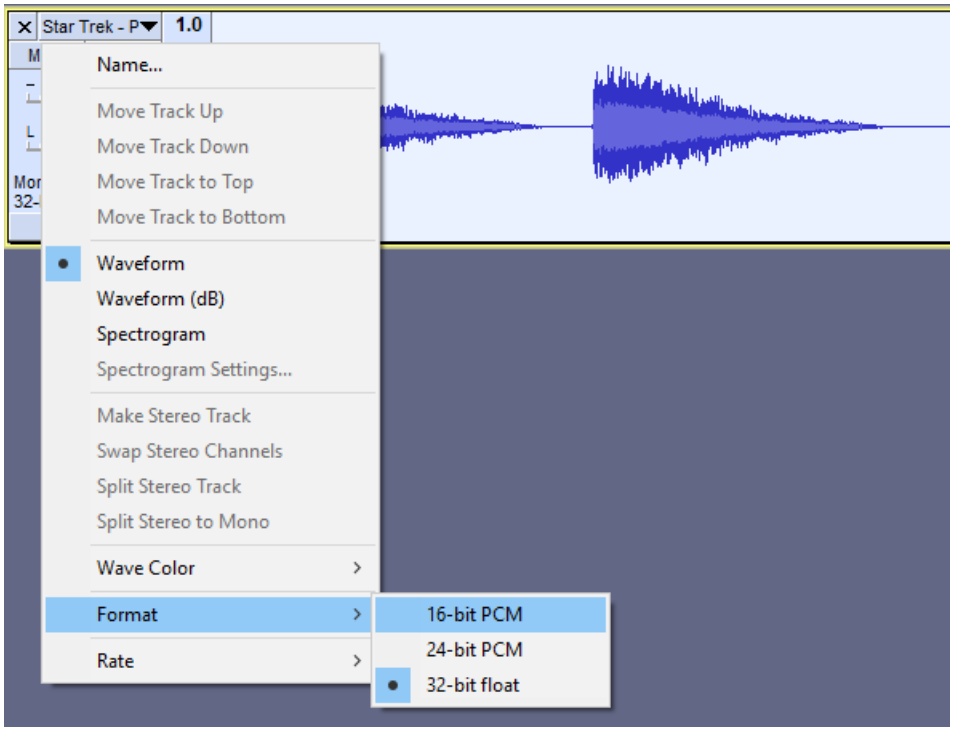
Your file should now have one waveform and the text on the left says **Mono**. This is your combined track.



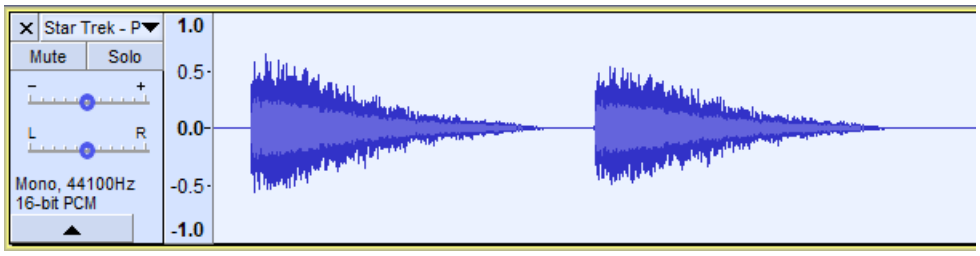
Step 4: Convert File to 16-bit Audio

If your audio rate is higher than 16-bit, you will want to downconvert it. The sample in the pictures above shows it is 32-bit.

Click on the track title and select **Set Sample Format -> 16-bit**



Your file bitrate should now show 16-bit PCM in the properties to the left.

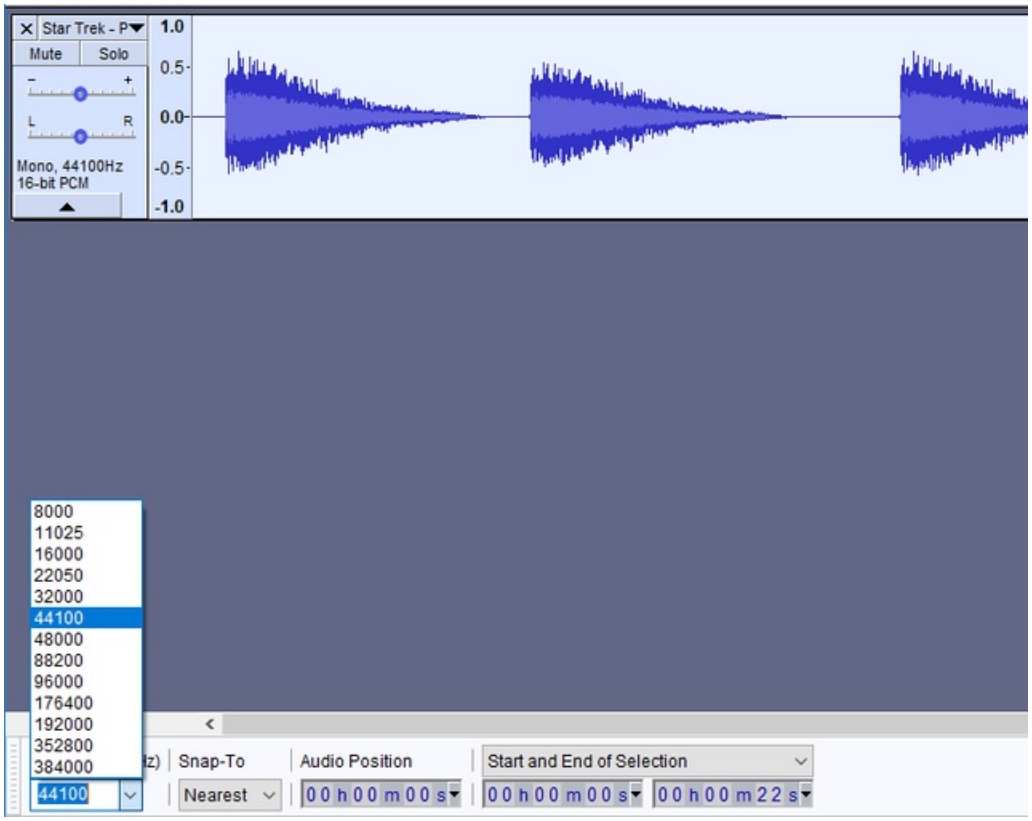


Step 5: Convert to 22 KHz or Less

Finally, make sure the audio file will be saved as 22 kHz or less. If the the track label says 44100 Hz or some number higher than 22050 you will want to convert it.

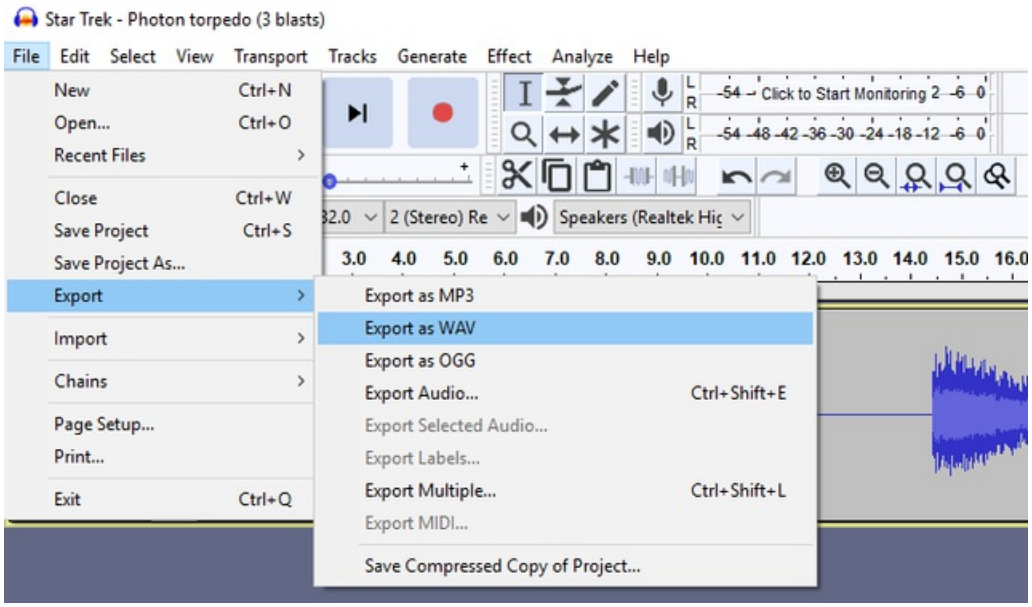
At the bottom of the window there is a little button named **Project Rate (Hz)** - Make sure this is 22 kHz or less.

The lower the number the smaller the number of samples and the smaller your file will be when saved. BUT, you lose sound quality with lower sampling rates. Be careful about your choices. Upconverting a low bitrate (say 8 kHz to 16 to 22 kHz) will not make your file sound better.

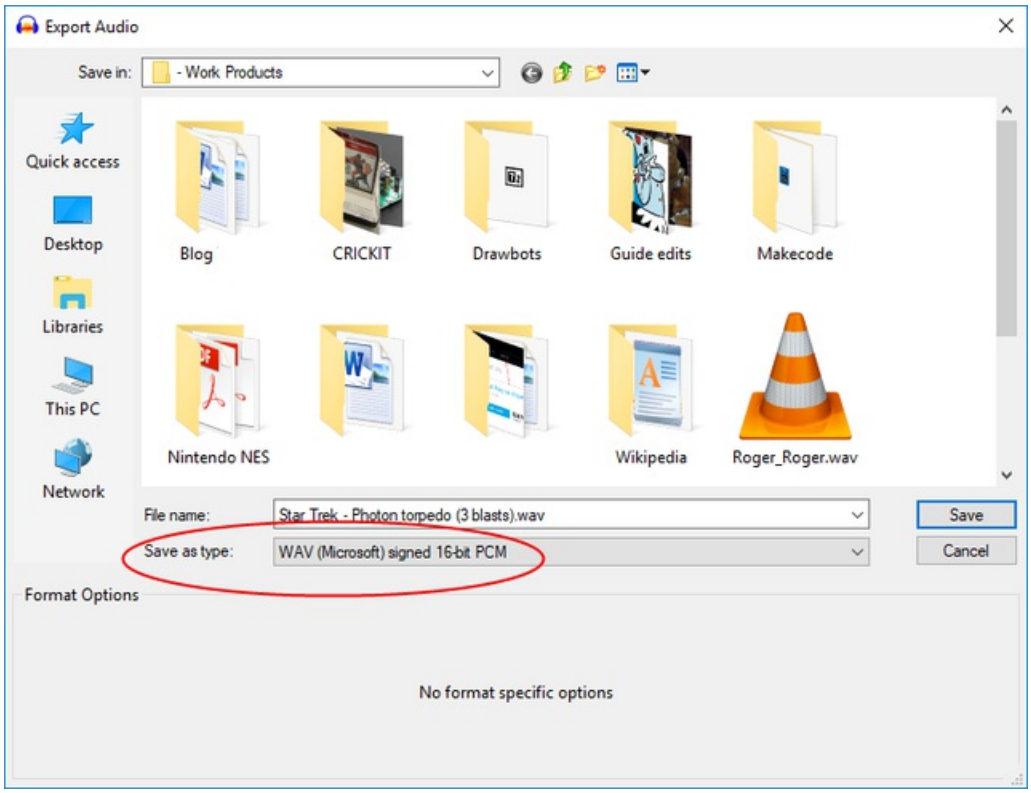


Step 6: Export

The file you have now is ready to be written to disk. Select **File -> Export -> Export as WAV**



This will bring up the save dialog. Be sure to select **WAV (Microsoft) signed 16-bit PCM** as the file type.



Now you have your audio file! You can put it on an SD Card for a music player or on an Express flash drive for playback or whatever!

Convert Sound Files in iTunes

You can do the conversion easily with iTunes (available for Mac/Windows). If you have your music in iTunes already, this will be super fast to convert multiple files!

You can download iTunes from the Apple website (https://adafru.it/dYx).

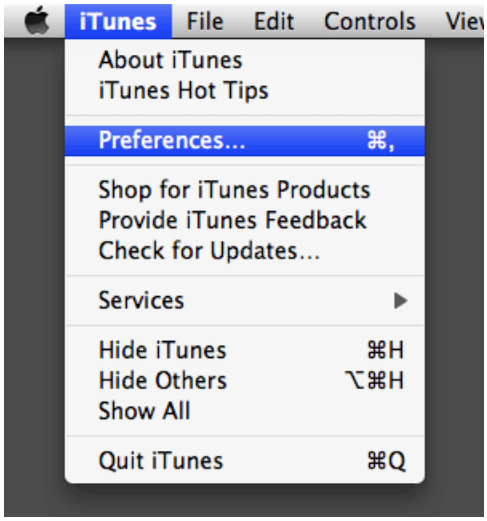
If you have Windows 10 or 10 S (for education), you can download a native Windows 10 version from the Microsoft App Store (https://adafru.it/BvM). For Windows 10 S this is the only version that may work. If you are having trouble downloading a Windows 10 version not from the Microsoft store, you might consider downloading this version (https://adafru.it/BvN) from Apple, then upgrading.

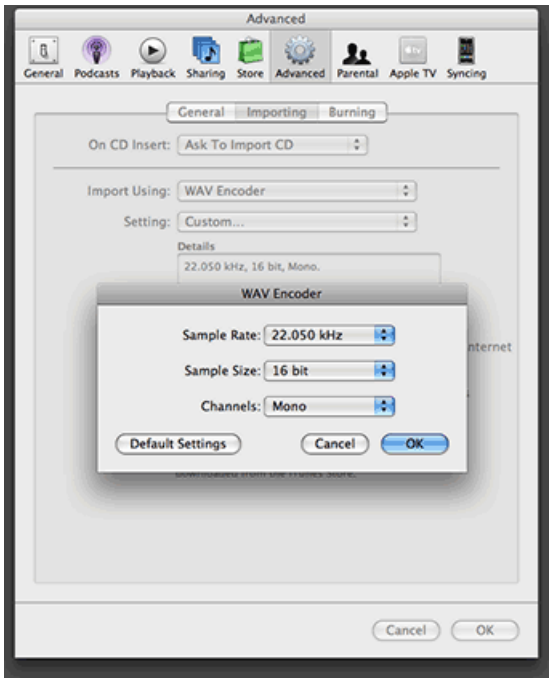
Once you install and run the program, you'll have to set the preferences first, but you only have to do it once.

Step 1: Set Preferences

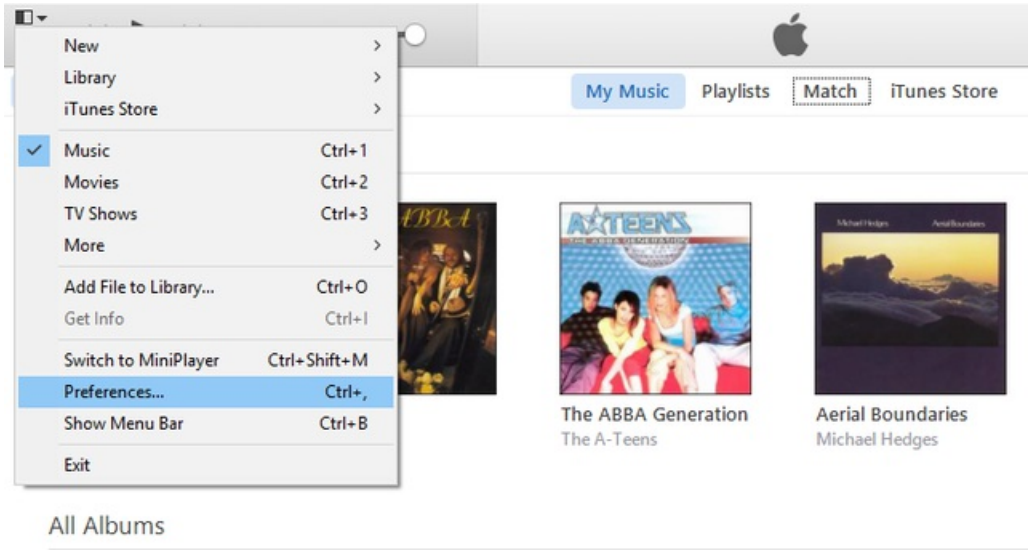
The settings depend on what version of iTunes you are running:

- For older versions, Go to the **Advanced -> Importing** tab. Make sure it is set to 22KHz (or less), 16-bit (or less) and Mono channels. Click OK.

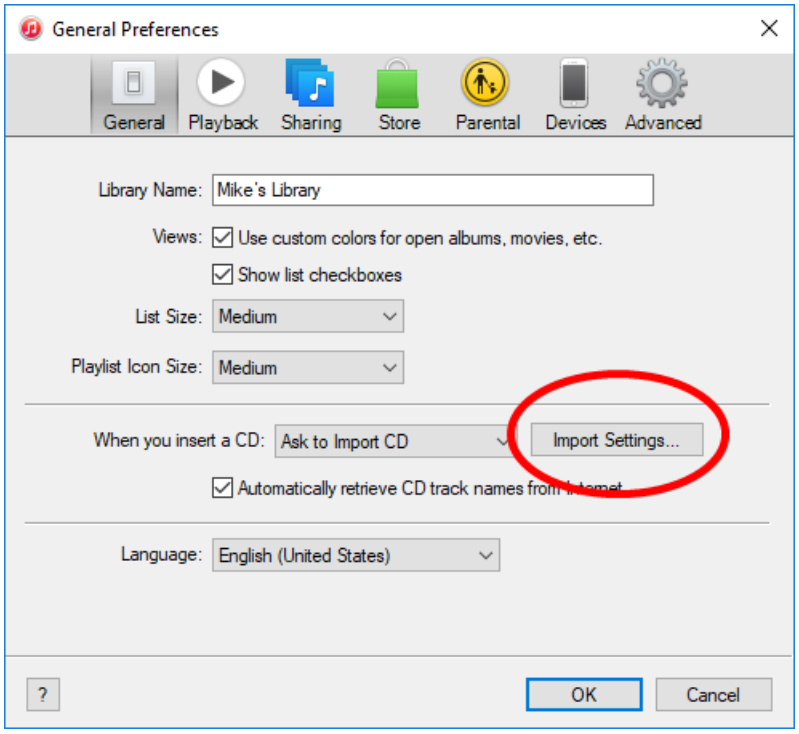




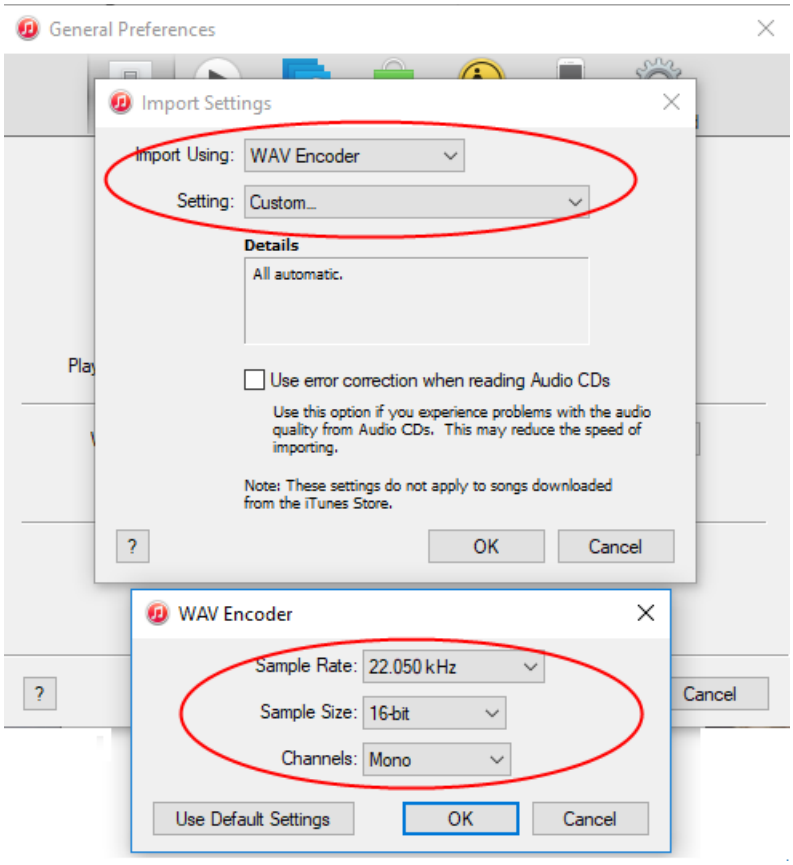
- For newer versions like 12.1 and later, go to the tiny square in the upper left hand corner, left click with your mouse, and then click **Preferences**



Then click the **Import Setting**s button under **General**:

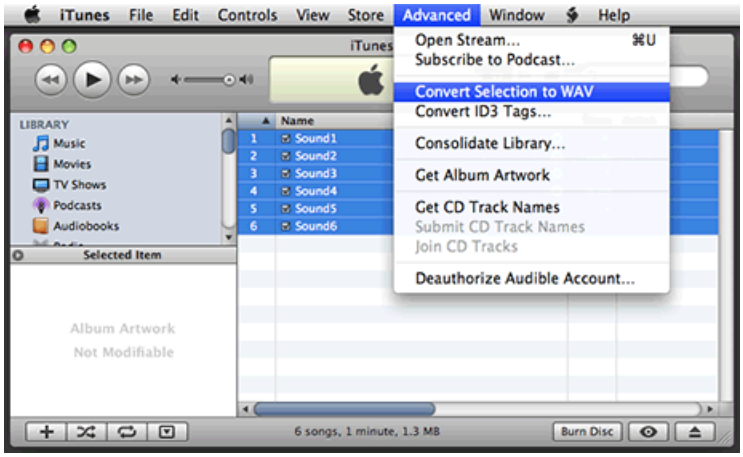


Then click **Import Using: WAV Encoder**, **Setting Custom**. In the new popup window, select **Sample Rate: 22,050 kHz, Sample Size: 16-bit, Channels: Mono**.

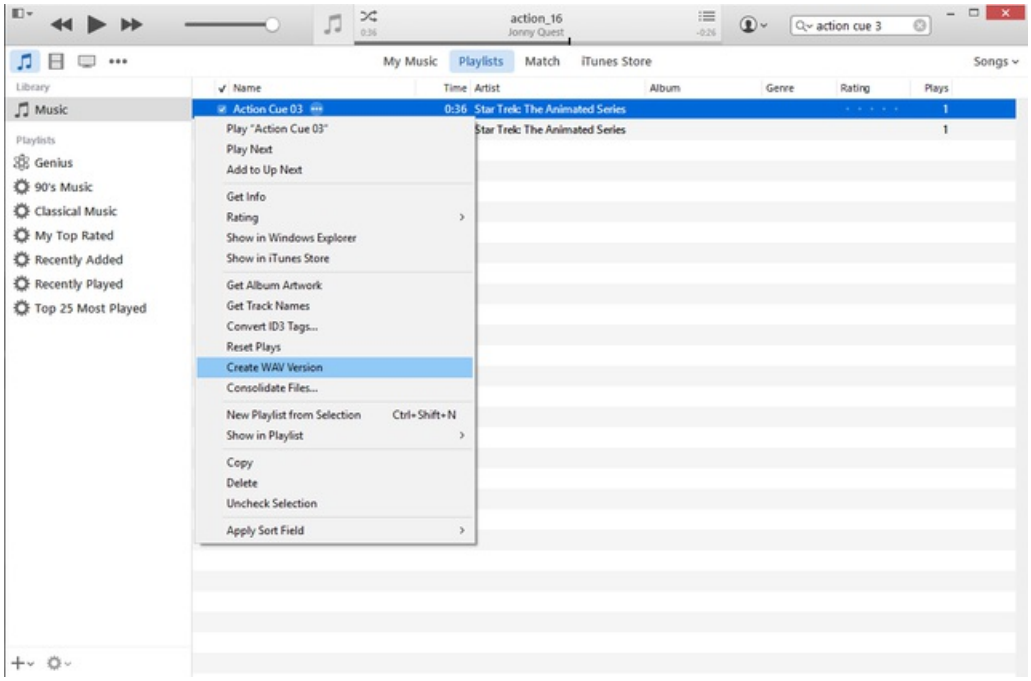


Step 2: Convert

Older versions: Search for your file. then select **Convert Selection to WAV** from the menu.

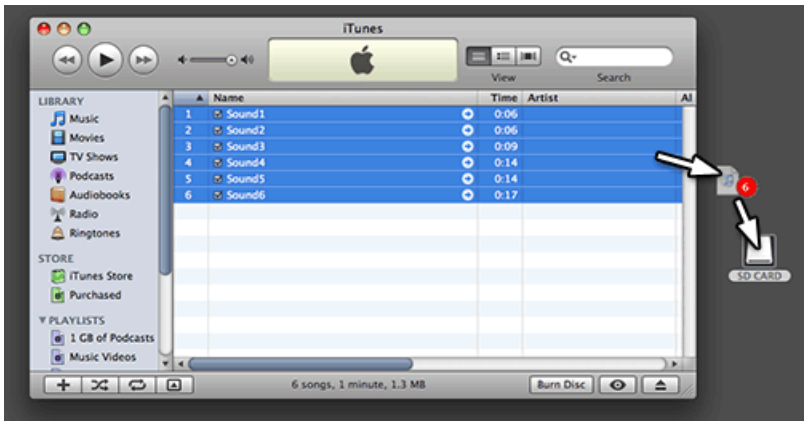


For newer versions without a menu, right click the file you want and select **Create WAV Version**



Step 3: Copy Your Files

Simply drag the wav copy of the sound files onto an SD card, your hard drive, or desktop.


iTunes
View
Search
LIBRARY
Music
Movies
TV Shows
Podcasts
Audiobooks
Radio
Ringtones
STORE
iTunes Store
Purchased
PLAYLISTS
1 GB of Podcasts
Music Videos
Name
Time
Artist
1 Sound1 0:06
2 Sound2 0:06
3 Sound3 0:09
4 Sound4 0:14
5 Sound5 0:14
6 Sound6 0:17
6 songs, 1 minute, 1.3 MB
Burn Disc
SD CARD

waveHC Library

Get more RAM & Flash!

This library uses a lot of RAM, if you are using an older '168 or '8 Arduino, you mustupgrade to an ATmega328 (https://adafru.it/c0B). The shield was designed with the expectation that this upgrade would be available.

A tour of dap_hc.pde

This is a tutorial of the waveHC library (https://adafru.it/kAa) by going through dap_hc.pde

It's detailed and a little daunting. But stick with it since much of the code is going to be duplicated from this sketch to others!

Make sure you install the library by downloading it from the link above and sticking WaveHC folder in the **libraries** folder. The zip also contains **dap_hc.pde**

Note: *The WaveHC library repository has an extra level of folders in it. To install the WaveHC library, you must first remove it from the top-level folder that you downloaded.*

In case you need the sketch we're referring to here, it's at the bottom of the page.

Initialize the card

The hard work of playing music is all done right on the Arduino. This lets us skip having an MP3 decoder or other dedicated chip. That means more control and flexibility, but more firmware! Lets take a tour through the canonical sketch "dapHC.pde" this is a Digital Audio Player (dap) sketch using the Wave HC librarry. We used to use the Adafruit AF_Wave library but Mr. Fat16 did a fantastic job rewriting our code and making it faster, smaller and better. So we're going to use his library in this tutorial!

Download the dapHC.pde sketch and read along! The first thing we need are some objects. The tough part is talking to the card. All cards are manufactured and formatted a little different. And the formatting has many layers - the card format - the partition format and the filesystem formatting. At the beginning of the sketch we have the #include to get the library header files in and an object for storing information about the card **card**, partition volume **vol** and filesystem **root**. We also have a directory buffer, for information on any folder/directories and an object for storing information about a single wave file **wave**.

These are all pretty much manditory unless perhaps you dont want directory traversal in which case you can probably skip **dirBuf**.

```
#include "WaveUtil.h"
#include "WaveHC.h"


SdReader card;    // This object holds the information for the card
FatVolume vol;    // This holds the information for the partition on the card
FatReader root;   // This holds the information for the filesystem on the card

uint8_t dirLevel; // indent level for file/dir names    (for prettyprinting)
dir_t dirBuf;     // buffer for directory reads

WaveHC wave;      // This is the only wave (audio) object, since we will only play one at a time
```

First thing that must be done is initializing the SD card for reading. This is a multistep process. In the setup loop you can see the multiple checks we do as we proceed through the initialization process.

Here are the steps:

1. **Wake up and print** out the to Serial Monitor that we're running. (**OPTIONAL**)
2. **Check how much free RAM we have** after we have used the buffer for storing Wave audio data, make sure its more than 100 bytes and keep an eye on it as you modify your code. This test can be removed, it's for your use. (**OPTIONAL**)
3. **Set the pin modes for the DAC** control lines. These should not be changed unless you've modified them in the library as well. It's probably best to keep them as-is.
4. **Initialize the SD card** and see if it responds. We try to talk to it at 8MHz. If you have a waveshield 1.0 you may need to use 4MHz mode so comment out one line and uncommment the other to swap which method is used. If the card fails to initialize, print out an error and halt.
5. **Allow partial block reads**. Some SD cards don't like this so if you're having problems, comment this out first! (**OPTIONAL**)
6. **Try to find a FAT partition** in the first 5 slots. You did format the card to FAT format, right? If it can't find a FAT partition it will print out that it failed, so make sure you format it again if it's giving you trouble.
7. **Print out what kind of FAT** partition was found. (**OPTIONAL**)
8. **Try to open up the root directory.** If this doesnt work, something is messed up with the formatting. Try to format it again!
9. Finally, print out the files found, one after the other in the directories on the card. This is great for debugging and will show you what you've got on there. Since we don't have long filename access and use the 'base' 8.3 format to define files, you'll need to see what the files are named on the partition and this helps a lot. (**OPTIONAL**)

```
void setup() {
  Serial.begin(9600);           // set up Serial library at 9600 bps for debugging

  putstring_nl("\nWave test!");  // say we woke up!

  putstring("Free RAM: ");       // This can help with debugging, running out of RAM is bad
  Serial.println(freeRam());

  // Set the output pins for the DAC control. This pins are defined in the library
  pinMode(2, OUTPUT);
  pinMode(3, OUTPUT);
  pinMode(4, OUTPUT);
  pinMode(5, OUTPUT);

  //  if (!card.init(true)) { //play with 4 MHz spi if 8MHz isn't working for you
  if (!card.init()) {         //play with 8 MHz spi (default faster!)
    putstring_nl("Card init. failed!");  // Something went wrong, lets print out why
    sdErrorCheck();
    while(1);                           // then 'halt' - do nothing!
  }

  // enable optimize read - some cards may timeout. Disable if you're having problems
  card.partialBlockRead(true);

  // Now we will look for a FAT partition!
  uint8_t part;
  for (part = 0; part < 5; part++) {     // we have up to 5 slots to look in
    if (vol.init(card, part))
      break;                            // we found one, lets bail
  }
  if (part == 5) {                      // if we ended up not finding one  :(
    putstring_nl("No valid FAT partition!");
    sdErrorCheck();      // Something went wrong, lets print out why
    while(1);                           // then 'halt' - do nothing!
  }

  // Lets tell the user about what we found
  putstring("Using partition ");
  Serial.print(part, DEC);
  putstring(", type is FAT");
  Serial.println(vol.fatType(),DEC);     // FAT16 or FAT32?

  // Try to open the root directory
  if (!root.openRoot(vol)) {
    putstring_nl("Can't open root dir!"); // Something went wrong,
    while(1);                            // then 'halt' - do nothing!
  }

  // Whew! We got past the tough parts.
  putstring_nl("Files found:");
  dirLevel = 0;
  // Print out all of the files in all the directories.
  lsR(root);
}
```

Looking for files in a directory

OK now that you've initialized the card, we perform a recursive list of all files found. This is useful for debugging and

ALSO shows how you can navigate the file system.

To start, pass a directory object (like **root**) to **lsR**() which will do the following:

1. Read a file from the directory. The files are read in the order they are copied into the directory, **not** alphabetical order!
2. If the directories are the special links "." (current directory) or ".." (upper directory) it ignores them and goes to step 1 again.
3. It prints out spaces to create a nicely formatted output. Each level of directory gets 2 spaces.
4. It prints out the name of the file in 8.3 format.
5. If it is a subdirectory, it makes a new object and opens up the subdirectory. Then it prints out all of the files in that new directory.
6. It continues to step 1 until there are no more files to be read.

```
/*
 * list recursively - possible stack overflow if subdirectories too nested
 */
void lsR(FatReader &d)
{
  int8_t r;                     // indicates the level of recursion

  while ((r = d.readDir(dirBuf)) > 0) {     // read the next file in the directory
    // skip subdirs . and ..
    if (dirBuf.name[0] == '.')
      continue;

    for (uint8_t i = 0; i < dirLevel; i++)
      Serial.print(' ');        // this is for prettyprinting, put spaces in front
    printEntryName(dirBuf);           // print the name of the file we just found
    Serial.println();           // and a new line

    if (DIR_IS_SUBDIR(dirBuf)) {   // we will recurse on any direcory
      FatReader s;                 // make a new directory object to hold information
      dirLevel += 2;               // indent 2 spaces for future prints
      if (s.open(vol, dirBuf))
        lsR(s);                    // list all the files in this directory now!
      dirLevel -=2;                // remove the extra indentation
    }
  }
  sdErrorCheck();                  // are we doign OK?
}
```

There is also a helper called **printEntryName** which prints out the file in a nice format. Files are named in **8.3** format, an older and simpler way of addressing files. It's a little less pretty than "Long Name Format" so watch out to see what your files are renamed as. For example "Bird song.wav" may be renamed to "BIRDSONG.WAV" or "BIRDSO~1.WAV" !

```
/*
 * print dir_t name field. The output is 8.3 format, so like SOUND.WAV or FILENAME.DAT
 */
void printEntryName(dir_t &dir)
{
  for (uint8_t i = 0; i < 11; i++) {     // 8.3 format has 8+3 = 11 letters in it
    if (dir.name[i] == ' ')
        continue;         // dont print any spaces in the name
    if (i == 8)
        Serial.print('.');           // after the 8th letter, place a dot
    Serial.print(dir.name[i]);      // print the n'th digit
  }
  if (DIR_IS_SUBDIR(dir))
    Serial.print('/');       // directories get a / at the end
}
```

One thing that appears in **loop**() is **dir.rewind**(). The reason we rewind a directory is that our Arduino code is very simple. It can go through the files in a directory but only 'forward', not backward (FAT format is kinda like that). So if you skipped a file and want to go back, or you've gone through the directory, you will need to call **rewind**() to set it back to the beginning!

Playing all the files

The digital audio player plays all files in the card. To do that it recursively looks in every directory, just like **lsR**() above so the code looks somewhat similar. The big difference is we call the **play**() routine to play a file!

To start, pass a directory object (like **root**) to **lsR**() which will do the following:

1. Read a file from the directory. The files are read in the order they are copied into the directory, **not** alphabetical order!
2. If the directories are the special links "." (current directory) or ".." (upper directory) it ignores them and goes to step 1 again.
3. It prints out spaces to create a nicely formatted output. Each level of directory gets 2 spaces.
4. It prints out the name of the file in 8.3 format.
5. If it is a subdirectory, it makes a new object and opens up the subdirectory. Then it plays all of the wave files in that new directory.
6. If it isn't a subdirectory, it will try to play the file by opening it as a Wave object. That requires looking through the file and trying to find a Wave header, etc. If it doesn't succeed it will print out that it's not valid and skip to the next file.
7. If the wave file is valid, it will finally start the file by calling **play**() on the Wave object.
8. While the wave sound file is playing, it prints out a dot every 100 ms or so.
9. It continues to step 1 until there are no more files to be read.

```
/*
 * play recursively - possible stack overflow if subdirectories too nested
 */
void play(FatReader &dir)
{
  FatReader file;
  while (dir.readDir(dirBuf) > 0) {    // Read every file in the directory one at a time
    // skip . and .. directories
    if (dirBuf.name[0] == '.')
      continue;

    Serial.println();            // clear out a new line

    for (uint8_t i = 0; i < dirLevel; i++)
       Serial.print(' ');       // this is for prettyprinting, put spaces in front

    if (!file.open(vol, dirBuf)) {       // open the file in the directory
      Serial.println("file.open failed");  // something went wrong :(
      while(1);                           // halt
    }

    if (file.isDir()) {                   // check if we opened a new directory
      putstring("Subdir: ");
      printEntryName(dirBuf);
      dirLevel += 2;                      // add more spaces
      // play files in subdirectory
      play(file);                        // recursive!
      dirLevel -= 2;
    }
    else {
      // Aha! we found a file that isnt a directory
      putstring("Playing "); printEntryName(dirBuf);       // print it out
      if (!wave.create(file)) {           // Figure out, is it a WAV proper?
        putstring(" Not a valid WAV");     // ok skip it
      } else {
        Serial.println();                 // Hooray it IS a WAV proper!
        wave.play();                      // make some noise!

        while (wave.isplaying) {          // playing occurs in interrupts, so we print dots in realtime
          putstring(".");
          delay(100);
        }
        sdErrorCheck();                   // everything OK?
//        if (wave.errors)Serial.println(wave.errors);     // wave decoding errors
      }
    }
  }
}
```

dap_hc.pde

The full sketch is in the library zip! (https://adafru.it/aQ7)

The Play6_HC Example

Get more RAM & Flash!

Before you try to play audio, you'll want to free up some Arduino RAM, so that you don't end up with a nasty stack-overflow. Running out of RAM is hard to debug and frustrating, and likely if you're using a '168.

Follow these instructions (https://adafru.it/c0C) on how to get more RAM by reducing the input Serial library buffer. You don't need to do this if you're using an ATmega328 (https://adafru.it/c0B).

Note that the library is pretty big (about 10K) so if you want to do a lot more, I suggest upgrading to an ATmega328 (https://adafru.it/c0B). The shield was designed with the expectation that this upgrade would be available.

A tour of play6_hc.pde

This is a tutorial of the waveHC library (https://adafru.it/kAa).

It's detailed and a little daunting. But stick with it since much of the code is going to be duplicated from this sketch to others!

Make sure you install the library by downloading it from the link above and sticking WaveHC folder in the **libraries** folder.

In case you need the sketch we're referring to here, it's at the bottom of the page.

Initialize the card

The hard work of playing music is all done right on the Arduino. This lets us skip having an MP3 decoder or other dedicated chip. That means more control and flexibility, but more firmware! Lets take a tour through the canonical sketch "dapHC.pde" this is a Digital Audio Player (dap) sketch using the Wave HC librarry. We used to use the Adafruit AF_Wave library but Mr. Fat16 did a fantastic job rewriting our code and making it faster, smaller and better. So we're going to use his library in this tutorial :)

Download the dapHC.pde sketch and read along! The first thing we need are some objects. The tough part is talking to the card. All cards are manufactured and formatted a little different. And the formatting has many layers - the card format - the partition format and the filesystem formatting. At the beginning of the sketch we have the #include to get the library header files in and an object for storing information about the card **card**, partition volume **vol** and filesystem **root**. We also have an object for holding the current file information **f** and an object for storing information about a single wave file **wave**.

```
#include "WaveUtil.h"
#include "WaveHC.h"


SdReader card;    // This object holds the information for the card
FatVolume vol;    // This holds the information for the partition on the card
FatReader root;   // This holds the information for the filesystem on the card
FatReader f;      // This holds the information for the file we're playing


WaveHC wave;      // This is the only wave (audio) object, since we will only play one at a time

#define DEBOUNCE 100  // button debouncer
```

First thing that must be done is initializing the SD card for reading. This is a multistep process. In the setup loop you can see the multiple checks we do as we proceed through the initialization process.

Here are the steps:

1. **Wake up and print** out the to Serial Monitor that we're running. (**OPTIONAL**)
2. **Check how much free RAM we have** after we have used the buffer for storing Wave audio data, make sure it's more than 100 bytes and keep an eye on it as you modify your code. This test can be removed, it's for your use. (**OPTIONAL**)
3. **Set the pin modes for the DAC** control lines. These should not be changed unless you've modified them in the library as well. It's probably best to keep them as-is.
4. **Initialize the SD card** and see if it responds. We try to talk to it at 8MHz. If you have a waveshield 1.0 you may need to use 4MHz mode so comment out one line and uncommment the other to swap which method is used. If the card fails to initialize, print out an error and halt.
5. **Allow partial block reads**. Some SD cards don't like this so if you're having problems, comment this out first! (**OPTIONAL**)
6. **Try to find a FAT partition** in the first 5 slots. You did format the card to FAT format, right? If it can't find a FAT partition it will print out that it failed, so make sure you format it again if it's giving you trouble.
7. **Print out what kind of FAT** partition was found. (**OPTIONAL**)
8. **Try to open up the root directory.** If this doesn't work, something is messed up with the formatting. Try to format it again!

```
void setup() {
  // set up serial port
  Serial.begin(9600);
  putstring_nl("WaveHC with 6 buttons");

   putstring("Free RAM: ");       // This can help with debugging, running out of RAM is bad
  Serial.println(freeRam());      // if this is under 150 bytes it may spell trouble!

  // Set the output pins for the DAC control. This pins are defined in the library
  pinMode(2, OUTPUT);
  pinMode(3, OUTPUT);
  pinMode(4, OUTPUT);
  pinMode(5, OUTPUT);

  // pin13 LED
  pinMode(13, OUTPUT);

  // enable pull-up resistors on switch pins (analog inputs)
  digitalWrite(14, HIGH);
```

```
  digitalWrite(14, HIGH);
  digitalWrite(15, HIGH);
  digitalWrite(16, HIGH);
  digitalWrite(17, HIGH);
  digitalWrite(18, HIGH);
  digitalWrite(19, HIGH);

  //  if (!card.init(true)) { //play with 4 MHz spi if 8MHz isn't working for you
  if (!card.init()) {         //play with 8 MHz spi (default faster!)
    putstring_nl("Card init. failed!");  // Something went wrong, lets print out why
    sdErrorCheck();
    while(1);                           // then 'halt' - do nothing!
  }

  // enable optimize read - some cards may timeout. Disable if you're having problems
  card.partialBlockRead(true);

// Now we will look for a FAT partition!
  uint8_t part;
  for (part = 0; part < 5; part++) {     // we have up to 5 slots to look in
    if (vol.init(card, part))
      break;                            // we found one, lets bail
  }
  if (part == 5) {                      // if we ended up not finding one  :(
    putstring_nl("No valid FAT partition!");
    sdErrorCheck();      // Something went wrong, lets print out why
    while(1);                           // then 'halt' - do nothing!
  }

  // Lets tell the user about what we found
  putstring("Using partition ");
  Serial.print(part, DEC);
  putstring(", type is FAT");
  Serial.println(vol.fatType(),DEC);     // FAT16 or FAT32?

  // Try to open the root directory
  if (!root.openRoot(vol)) {
    putstring_nl("Can't open root dir!"); // Something went wrong,
    while(1);                            // then 'halt' - do nothing!
  }

  // Whew! We got past the tough parts.
  putstring_nl("Ready!");
  dirLevel = 0;
}
```

Button interfacing

We want to play a sound each time a button is pressed. We will use a function called check_switches() that goes through the 6 buttons (digital 14 through 20) to see if they have been pressed. If so, we play **SOUND1.WAV** (for example) completely through. The function that we call here that does the playing is called **playcomplete**() and we pass the name of the Wave file in quotes just like you see here.

```
void loop() {
  //putstring(".");            // uncomment this to see if the loop isnt running
  switch (check_switches()) {
    case 1:
      playcomplete("SOUND1.WAV");
      break;
    case 2:
      playcomplete("SOUND2.WAV");
      break;
    case 3:
      playcomplete("SOUND3.WAV");
      break;
    case 4:
      playcomplete("SOUND4.WAV");
      break;
    case 5:
      playcomplete("SOUND5.WAV");
      break;
    case 6:
      playcomplete("SOUND6.WAV");
  }
}

byte check_switches()
{
  static byte previous[6];
  static long time[6];
  byte reading;
  byte pressed;
  byte index;
  pressed = 0;

  for (byte index = 0; index < 6; ++index) {
    reading = digitalRead(14 + index);
    if (reading == LOW && previous[index] == HIGH && millis() - time[index] > DEBOUNCE)
    {
      // switch pressed
      time[index] = millis();
      pressed = index + 1;
      break;
    }
    previous[index] = reading;
  }
  // return switch number (1 - 6)
  return (pressed);
}
```

Playcomplete & Playfile

Here is where we open the file and play it.

Playcomplete is very simple, it just calls a function that starts the audio playback and then sits in a loop doing nothing.

Playfile is the important function. It finds and opens the file and plays it.

1. It first sees if we're already playing any audio. If so, it stops it.
2. Now it opens the root directory and looks for the file by the name we requested. If it can't find it, the function

returns.
3. If it finds it, it tries to turn it into a Wave file object, looking for the right header in the file. If not it also returns.
4. If it succeeds, it begins to play.

```
// Plays a full file from beginning to end with no pause.
void playcomplete(char *name) {
  // call our helper to find and play this name
  playfile(name);
  while (wave.isplaying) {
  // do nothing while its playing
  }
  // now its done playing
}

void playfile(char *name) {
  // see if the wave object is currently doing something
  if (wave.isplaying) {// already playing something, so stop it!
    wave.stop(); // stop it
  }
  // look in the root directory and open the file
  if (!f.open(root, name)) {
    putstring("Couldn't open file "); Serial.print(name); return;
  }
  // OK read the file and turn it into a wave object
  if (!wave.create(f)) {
    putstring_nl("Not a valid WAV"); return;
  }

  // ok time to play! start playback
  wave.play();
}
```

play6_hc.pde

```
#include <FatReader.h>
#include <SdReader.h>
#include <avr/pgmspace.h>
#include "WaveUtil.h"
#include "WaveHC.h"


SdReader card;    // This object holds the information for the card
FatVolume vol;    // This holds the information for the partition on the card
FatReader root;   // This holds the information for the filesystem on the card
FatReader f;      // This holds the information for the file we're play

WaveHC wave;      // This is the only wave (audio) object, since we will only play one at a time

#define DEBOUNCE 100  // button debouncer

// this handy function will return the number of bytes currently free in RAM, great for debugging!
int freeRam(void)
{
  extern int  __bss_end;
  extern int  *__brkval;
  int free_memory;
```

```
  if((int)__brkval == 0) {
    free_memory = ((int)&free_memory) - ((int)&__bss_end);
  }
  else {
    free_memory = ((int)&free_memory) - ((int)__brkval);
  }
  return free_memory;
}

void sdErrorCheck(void)
{
  if (!card.errorCode()) return;
  putstring("\n\rSD I/O error: ");
  Serial.print(card.errorCode(), HEX);
  putstring(", ");
  Serial.println(card.errorData(), HEX);
  while(1);
}

void setup() {
  // set up serial port
  Serial.begin(9600);
  putstring_nl("WaveHC with 6 buttons");

   putstring("Free RAM: ");       // This can help with debugging, running out of RAM is bad
  Serial.println(freeRam());      // if this is under 150 bytes it may spell trouble!

  // Set the output pins for the DAC control. This pins are defined in the library
  pinMode(2, OUTPUT);
  pinMode(3, OUTPUT);
  pinMode(4, OUTPUT);
  pinMode(5, OUTPUT);

  // pin13 LED
  pinMode(13, OUTPUT);

  // enable pull-up resistors on switch pins (analog inputs)
  digitalWrite(14, HIGH);
  digitalWrite(15, HIGH);
  digitalWrite(16, HIGH);
  digitalWrite(17, HIGH);
  digitalWrite(18, HIGH);
  digitalWrite(19, HIGH);

  //  if (!card.init(true)) { //play with 4 MHz spi if 8MHz isn't working for you
  if (!card.init()) {         //play with 8 MHz spi (default faster!)
    putstring_nl("Card init. failed!");  // Something went wrong, lets print out why
    sdErrorCheck();
    while(1);                           // then 'halt' - do nothing!
  }

  // enable optimize read - some cards may timeout. Disable if you're having problems
  card.partialBlockRead(true);

// Now we will look for a FAT partition!
  uint8_t part;
  for (part = 0; part < 5; part++) {     // we have up to 5 slots to look in
    if (vol.init(card, part))
      break;                             // we found one, lets bail
  }
```

```
  if (part == 5) {                     // if we ended up not finding one  :(
    putstring_nl("No valid FAT partition!");
    sdErrorCheck();      // Something went wrong, lets print out why
    while(1);                          // then 'halt' - do nothing!
  }

  // Lets tell the user about what we found
  putstring("Using partition ");
  Serial.print(part, DEC);
  putstring(", type is FAT");
  Serial.println(vol.fatType(),DEC);     // FAT16 or FAT32?

  // Try to open the root directory
  if (!root.openRoot(vol)) {
    putstring_nl("Can't open root dir!"); // Something went wrong,
    while(1);                            // then 'halt' - do nothing!
  }

  // Whew! We got past the tough parts.
  putstring_nl("Ready!");
}

void loop() {
  //putstring(".");            // uncomment this to see if the loop isnt running
  switch (check_switches()) {
    case 1:
      playcomplete("SOUND1.WAV");
      break;
    case 2:
      playcomplete("SOUND2.WAV");
      break;
    case 3:
      playcomplete("SOUND3.WAV");
      break;
    case 4:
      playcomplete("SOUND4.WAV");
      break;
    case 5:
      playcomplete("SOUND5.WAV");
      break;
    case 6:
      playcomplete("SOUND6.WAV");
  }
}

byte check_switches()
{
  static byte previous[6];
  static long time[6];
  byte reading;
  byte pressed;
  byte index;
  pressed = 0;

  for (byte index = 0; index < 6; ++index) {
    reading = digitalRead(14 + index);
    if (reading == LOW && previous[index] == HIGH && millis() - time[index] > DEBOUNCE)
    {
      // switch pressed
```

```
      time[index] = millis();
      pressed = index + 1;
      break;
    }
    previous[index] = reading;
  }
  // return switch number (1 - 6)
  return (pressed);
}


// Plays a full file from beginning to end with no pause.
void playcomplete(char *name) {
  // call our helper to find and play this name
  playfile(name);
  while (wave.isplaying) {
  // do nothing while its playing
  }
  // now its done playing
}

void playfile(char *name) {
  // see if the wave object is currently doing something
  if (wave.isplaying) {// already playing something, so stop it!
    wave.stop(); // stop it
  }
  // look in the root directory and open the file
  if (!f.open(root, name)) {
    putstring("Couldn't open file "); Serial.print(name); return;
  }
  // OK read the file and turn it into a wave object
  if (!wave.create(f)) {
    putstring_nl("Not a valid WAV"); return;
  }

  // ok time to play! start playback
  wave.play();
}
```

AFwave Lib.

DISCONTINUED!

This documentation is for historical reference only!

> AF_Wave is no longer supported or used, it does not work with the SDHC SD Cards, so we put it to pasture. Please use the superior WaveHC instead!

Get more RAM & Flash!

Before you try to play audio, you'll want to free up some Arduino RAM, so that you don't end up with a nasty stack-overflow.

Follow these instructions (https://adafru.it/c0C) on how to get more RAM by reducing the input Serial library buffer. You dont need to do this if you're using an ATmega328 (https://adafru.it/c0B).

Note that the library is pretty big (about 10K) so if you want to do a lot more, I suggest upgrading to an ATmega328 (https://adafru.it/c0B). The shield was designed with the expectation that this part would be available.

A tour of the AF_Wave library

This is a description of the AF_Wave library, which is the 'default' library for the Wave shield. However, there is an 'updated' and superior library, WaveHC written by Mr Fat16 (https://adafru.it/c0D) in the forums. This library is powerful, works with more cards and card formatting issues, and uses less space. This tutorial is here for those who want to use the classic AF_Wave library but we suggest you also check out WaveHC (https://adafru.it/aQ7). It's very similar to AF_Wave so you can probably switch between the two.

We have a runthrough of WaveHC over here (https://adafru.it/dMB).

Initialize the card

The first thing that must be done is initializing the SD card for reading. You should copy & paste this code from the examples since there's really only one way to do it.

Note that this here is a snippet, use the examples in the library for the 'full listing.'

```
AF_Wave card;


void setup()
{
  ...

  if (!card.init_card()) {
    putstring_nl("Card init. failed!"); return;
  }
  if (!card.open_partition()) {
    putstring_nl("No partition!"); return;
  }
  if (!card.open_filesys()) {
    putstring_nl("Couldn't open filesys"); return;
  }

 if (!card.open_rootdir()) {
    putstring_nl("Couldn't open dir"); return;
  }
}


...
```

This code will try to initialize the card, open the partition table, open the FAT16 filesystem and finally open the root directory. If it fails it will print out an error message.

Looking for files

There isn't a lot of interface code for going through the root directory. Basically you can reset the directory (start over from beginning) and get the name of the next file. The files are not organized alphabetically but rather in the order that they were created on the card.

You'll need to make a character array 13 characters long to store the 8.3 + terminating 0 of the file. Here is an example of displaying the name of each file available. When done, it resets the directory.

```
void ls() {
  char name[13];
  int ret;

  card.reset_dir();
  putstring_nl("Files found:");
  while (1) {
    ret = card.get_next_name_in_dir(name);
    if (!ret) {
       card.reset_dir();
       return;
    }
    Serial.println(name);
  }
}
```

Opening a file for playing

There are two steps to opening a file for playing. The first is to just open the file itself, then the file must be converted to a wavefile. That means the file is read and checked for a wavetable header. To open a file, you just need the name, you can pass in a string such as "MYSOUND.WAV" or read through the directory and use the name returned from **get_next_name_in_dir**(). Since long names arent supported (to keep the library smaller) you may want to use **ls**() function above to see what the 8.3 format name of the file is.

```
AF_Wave card;
File f;
Wavefile wave;      // only one!



void playfile(char *name) {
   f = card.open_file(name);
   if (!f) {
      putstring_nl(" Couldn't open file"); return;
   }
   if (!wave.create(f)) {
     putstring_nl(" Not a valid WAV"); return;
   }


  ...
}
```

Playing the file

Finally we can play the file! It's quite easy, once the wavefile has been opened as above, simply call **wave.play**() to being playback. The Arduino plays audio in an interrupt, which means that **wave.play**() returns immediately. You can then mess with sensors, print feedback or buttons or whatever.

While the wavefile is playing, you can check its status with the variable **wave.isplaying**. If the variable is 1 then the audio is playing. If its 0 that means it has finished playing.

You can stop playback by calling **wave.stop**()

Closing the file

When you're done playing audio from a file, you must close it! You can close the file by calling **card.close_file(f)** where **f** is the file you created using **card.open_file(name)**

Changing sample rate

This is sort of strange, but may be useful if, say, you have a sine wave or sample that youd like to change the pitch of or if you'd like to 'fast forward' through some music.

The sample rate (i.e. 22kHz) is stored in **wave.dwSamplesPerSec**. It will be initially set to whatever the wave file is supposed to be. Simply assign a new sample rate to the variable to change it on the fly.

See here for more information. (https://adafru.it/c0y)

Saving & restoring the play position

If, say, you want to know where along in the wave file you are, that information is also available in **wave. in**

wave.getSize() (the number of bytes in the entire wave) and **wave.remainingBytesInChunk** (how many bytes are left to play).

You can set the current place to play from using **wave.seek**(), the Arduino will immediately start to fastforward to that location. For example, **wave.seek**(0) will take you to the beginning, **wave.seek(wave.getSize()/2)** will take you to the middle of the file.

Volume adjust

You can change the volume of the audio 'digitally' on the fly. Note that this doesn't change the volume control potentiometer, it actually just reduces the digital values going to the DAC. Thus the quality of the audio will be degraded. However, it may come in handy so it has been included. Since it slows down playback a bit, it is not enabled by default. To enable digital volume control, open up **wave.cpp** in the library folder and look for the line **#define DVOLUME 0** and change the **0** to a **1**. Then delete all the files in the library folder that end with .o, this will force the software to recompile the library when the sketch is compiled.

The volume is controlled by a variable in the Wavefile object. For example, if you have **Wavefile wave** at the top of your sketch, then you can set the volume by calling **wave.volume** = **4**. The volume can be set from 0 to 12. A volume value of 0 is maximum, and 12 is silence. Anything higher than 12 will be the same as 12.

See here for more information (https://adafru.it/c0y).

Examples

Getting Stack overflow errors?

These examples are all tested to work with IDE v13 or higher, so try to use that if possible!

Get more RAM & Flash!

This library uses a lot of RAM, if you are using an older '168 or '8 Arduino, you must upgrade to an ATmega328 (https://adafru.it/c0B). The shield was designed with the expectation that this upgrade would be available.

Generating speech

If you want a human voice in your project, you can use the free generator at AT&T Text-to-Speech demo page (https://adafru.it/c0E).

It will create a 16KHz, 16-bit audio file so you can use the audio 'right out of the box.'

Sound sample library

Here is huge collection of C.C. Attribution licensed sound samples! (https://adafru.it/c0F) A lot of it is already mono, 16 or 22KHz.

Digital audio player

This is the simplest example. It plays every audio file it finds on the SD card in a loop. This sketch is also included in the library.

1. Sketch (https://adafru.it/cnZ) (for the waveHC library (https://adafru.it/aQ7) also check the WaveHC library zip for any newer revision) (https://adafru.it/aQ7).

PI party!

This example shows how to use the AT&T text-to-speech website to speak the first n digits of pi. The number is stored in flash, each digit is spoken one at a time.

1. Zip file containing the digits 0 thru 9 and 'point' (https://adafru.it/co0) place the wave files onto the SD card in the root directory.
2. Sketch (https://adafru.it/co1) (for the waveHC library also check the WaveHC library zip for any newer revision) there's a walkthrough here (https://adafru.it/c0w).

6 buttons, 6 sounds, multiple possibilities!

Here is a collection of different playback techniques with 6 buttons (connected to analog 0-5).

You can change the # of buttons and what they're wired to easily. This isn't an exhaustive list but will give you some ideas about how you can do a lot with the WaveHC library (https://adafru.it/aQ7). Note that all the changes occur in loop(). The button checking and everything else is the same.

1. Play the wave file all the way through, and only once (https://adafru.it/co2).
2. Play one wave file all the way through, in a loop (https://adafru.it/co3).
3. Play all the pressed wave files all the way through, in a loop (https://adafru.it/co4).
4. Play the wave file only when the button is held down and only once (https://adafru.it/co5) (kind of like a musical

keyboard).

5. Play the wave file only when the button is held down and loop it (https://adafru.it/co6) (kind of like a sampler keyboard).
6. Play the wave file all the way through and loop it, but allow other buttons to interrupt (https://adafru.it/co7).
7. Play the wave file all the way through once, but allow other buttons to interrupt (https://adafru.it/co8).

Playing sound based on input

This similar example plays 6 different files (https://adafru.it/co9) but its for WaveHC library (https://adafru.it/aQ7) (there's a walkthrough here). (https://adafru.it/c0x)

This one plays 4 different files depending on serial characters (https://adafru.it/coa), good if you have say an xbee you want to use.

Changing the playback rate

By messing with the playback interrupt, you can change the speed of playback for an interesting effect (https://adafru.it/c0R).

Here is the sketch (https://adafru.it/coc), connect the potentiometer to analog pin 0 (or change the code).

Wave Shield Voice Changer

Speak like everyone's favorite baritone Sith lord or sing along with the Lollipop Guild! (https://adafru.it/c0T)

Volume control via software

By changing the #define DVOLUME 1 in wave.cpp and recompiling you can do rudimentary software volume control as in this sketch (https://adafru.it/cod).

Downloads

Arduino WaveHC Library

Newer! Better! Sparklier! Download the WaveHC library by clicking here (https://adafru.it/kAb)

For information how to use and install libraries, see our tutorial! (https://adafru.it/aYG) Then check out the examples (https://adafru.it/c0y) for how to use it.

This library pretty much requires a '328 Arduino, see our upgrade tutorial if you have a '168 Arduino (https://adafru.it/c0B).

Arduino AF_Wave library

This library is no longer supported, but I'm keeping it here for history sake - please use only WaveHC above!

Download the latest library here (https://adafru.it/cmC) (currently: Feb 18, 2008 which now supports the 328P and arduino v13 under mac and windows & probably linux).

For information how to use and install libraries, see our tutorial! (https://adafru.it/aYG) Then check out the examples (https://adafru.it/c0y) for how to use it.

Before you try to play audio, you'll want to free up some Arduino RAM, so that you don't end up with a nasty stack-overflow. Follow these instructions (https://adafru.it/c0W) on how to get more RAM by reducing the input Serial library buffer.

The library is based off of Roland Riegel's AVR FAT16 code (https://adafru.it/c0X), but pared down quite a bit to reduce flash and RAM usage. There are also some strange optimizations to make it play audio better.

Demo waves

If you need some audio to test with, here is a synthesized voice saying "Hello world." (https://adafru.it/cmD)

Here is the collection of numbers for the pispeak example! (https://adafru.it/cmE)

Schematics & Layout

1. Schematics for v1.1 in PNG format (https://adafru.it/cmF)
2. v1.1 Schematics (https://adafru.it/cmG) and board layout (https://adafru.it/cmH) in EagleCAD format
3. Schematics for v1.0 in PNG format (https://adafru.it/cmI)
4. v1.0 Schematics (https://adafru.it/cmJ) and board layout (https://adafru.it/cmK) in EagleCAD format

Buy Kit

Buy Kit (https://adafru.it/alH)

Forums

Forums (https://adafru.it/aOM)