XDJ100SX

Open Source Standalone Deck

Rev: 1.0
Author: Marc Monka
Date: October 2025

I wanted to take an old DJ deck to convert into a modern one and to be standalone.

What standalone stands for?
Any DJ deck has a computer inside, usually under a Linux operating system which is 100% optimized for that device + a software which is the one that can read, play and manipulate tracks with different features. The software also has a dedicated look/skin so perhaps for a standard user this does not look like a computer / software, but it certainly is.

In conclusion, a standalone deck is a controller that controls a dedicated software that is running in a mini computer where everything fits together in a box.

Goal
Recently AlphaTheta dropped their CDJ-3000X, the successor of the 3000 which apart from a bigger screen and some additional hardware additions, almost all features could be upgradable with a firmware / software update. This is a way of business, as many other brands like Apple's iPhone which gets renewed every year, and makes the previous release obsolete, or scheduled obsolescence.

For both standard and professional users, it's a way of having to spend lots of money to invest in something that won't be upgradeable and will be obsolete in less than 5 years, feeling also that it should be thrown away or to sell to the second hand market.

That's one of the reasons I wanted to make a fully functional, modern 20 year old device using open source software and make it friendly for any DJ user used to CDJ decks.

Be in mind I am not the first to do it. There are some other cool projects and modifications made by some other users. Unfortunately as far as I could find, they were not 100% open, meaning they depended on some licensed tools.

And so the second reason:

I wanted to make this project 100% open, not only in terms of sharing the code, but also using 100% open source software, and open source OS.

This project can be replicated to any other device. Codes should slightly change but the basis is the same.

How to make it work?

We just need to use an existing deck, and make all buttons to send MIDI through a firmware. The firmware will be Arduino based. This is the first open source part of this project.

But why MIDI? The goal is to have a standalone deck with lots of features that any other deck can do. There is already an existing DJ application which is 100% open source and works in Mac, Windows and Linux with different system architectures: Mixxx. This app has an amazing community of developers, djs and enthusiasts that make that anyone can DJ with any system, DJ setup. It's open and we will be able to customize and adapt to our project. This app can send and receive MIDI so it's the easiest way to communicate with it.

How then? Using a computer that we will attach inside the device. This computer will run Linux which is also an open source operating system. There are some options, I decided to go for the established Raspberry Pi Foundation that contributes to the Linux and open source community.

The CDJ-100S:

Requirements used for each deck:
Raspberry Pi 3B+
SD Card 64 GB Class A2 U3
Teensy LC
Power Supply 220V AC / 5V DC 3A
Capacitive Touchscreen 5" DSI Port
PCB Cables
Stereo audio cable
Rotary Encoder with button
USB connector Type A
Terminal connectors

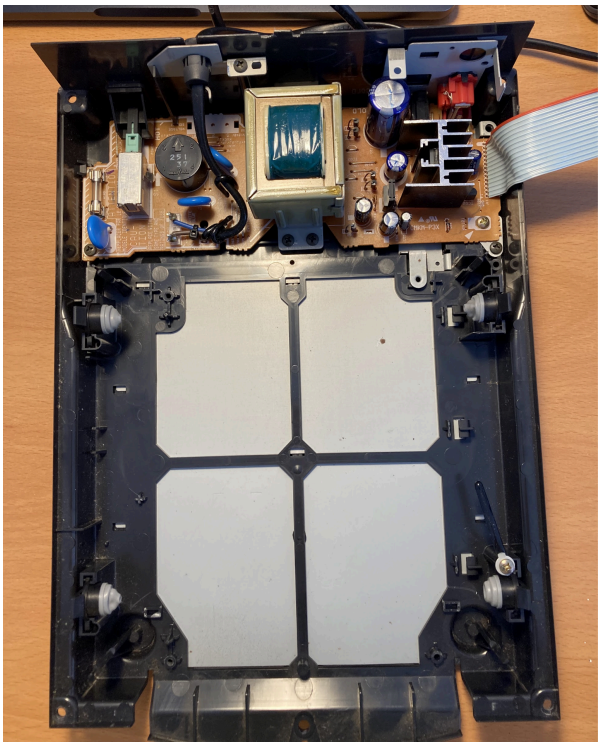
Tools needed:
Multimeter
Soldering Station
Screwdrivers
Cutting Plier
Arduino IDE app
Visual Studio Code
Midi Monitor

Phase 0: disassembling

Disassemble all the CDJ and remove all PCB and not desired parts: CD unit, main PCB and power supply. Some cables are attached to the front panel, so we also need to disconnect all of them.

What we want to keep:

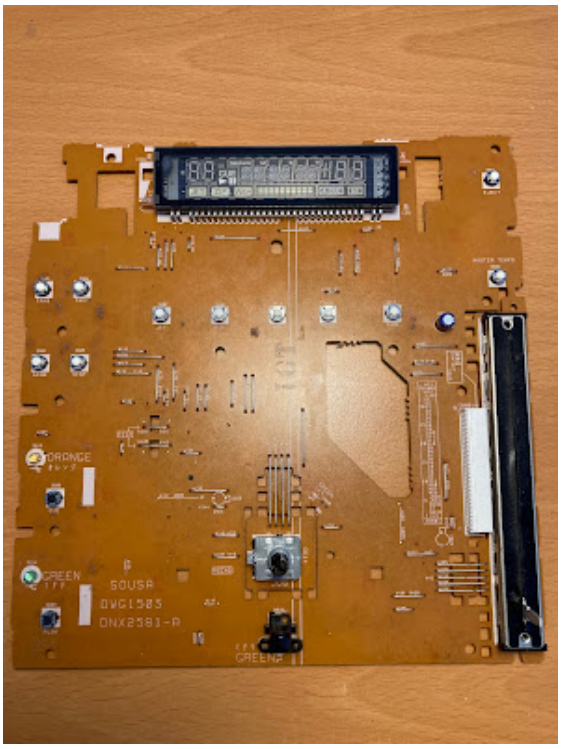
- Power switch (on/off): it's attached to the board, so we can just break/cut the board to keep the part where the switch is soldered.
- RCA audio connector: it's also soldered on the board. We need to desolder it.
- Main power cable.

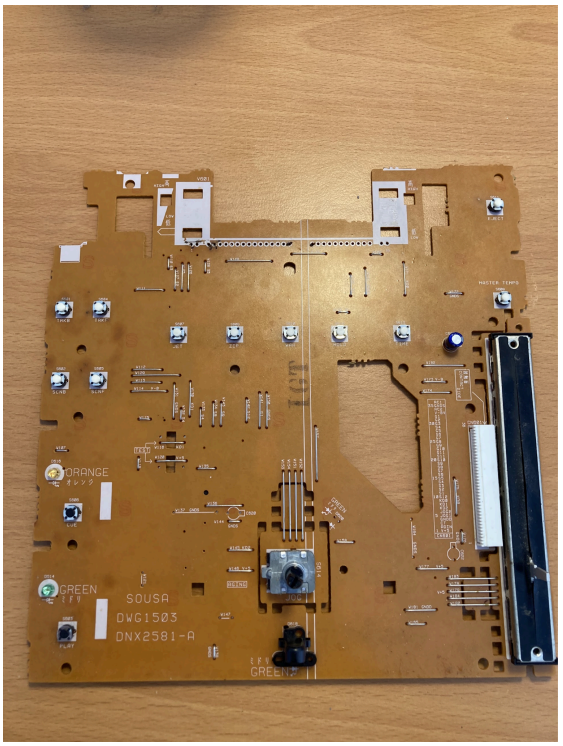


First figure: after removing the front panel.

Right figure: after removing the main PCB.

Front panel: we need to remove the old screen desoldering all pins (it may take a while):





Phase 1: soldering

The first thing we have to do is to check all original buttons and leds (in addition to the ones we want to add). This is important so we must know how many connections we have before getting the Teensy/Arduino PCB.

The CDJ100S has:

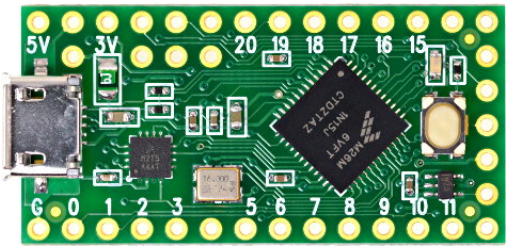
Name	Type	In/out	Number of pins	Analog / Digital?
Eject	Button	IN	1	Digital
Track Search Previous	Button	IN	1	Digital
Track Search Next	Button	IN	1	Digital
Search Back	Button	IN	1	Digital
Search Forward	Button	IN	1	Digital
Cue	Button	IN	1	Digital
Play/Pause	Button	IN	1	Digital
Jet	Button	IN	1	Digital
Zip	Button	IN	1	Digital
Wah	Button	IN	1	Digital
Hold	Button	IN	1	Digital
Time mode / Auto Cue	Button	IN	1	Digital
Master Tempo	Button	IN	1	Digital
Pitch	Slider	IN	1	Analog
Jog Wheel	Encoder	IN	2	Digital
Cue Led	Led	OUT	1	Digital
Play Led	Led	OUT	1	Digital
Internal Led	Led	OUT	1	Digital
CD Led	Led	OUT	1	Digital
Browse (addition)	Encoder	IN	2	Digital
Load (addition)	Button	IN	1	Digital
TOTAL			**23**	

So the CDJ100S has a total of 20 pins. It does not have any encoder to browse so we will add one encoder with a button and we will be able to navigate through the library and load a track. This adds 3 additional pins, so we need a board with at least 23 pins.

The pitch slider and encoders need a tension of 3,3 V to work. We must know that our board also has a Vout of 3,3 V. Also, the pitch slider has to be read as analog so we need analog input.

Now that we know that we need 23 pins we check in the market what's the best option. To work with Mixxx we will use the MIDI protocol, so we have to be sure that our board is compatible with MIDI libraries.

I've found the Teensy LC has 26 pins, it does have analog/digital and 3,3V Vout too.

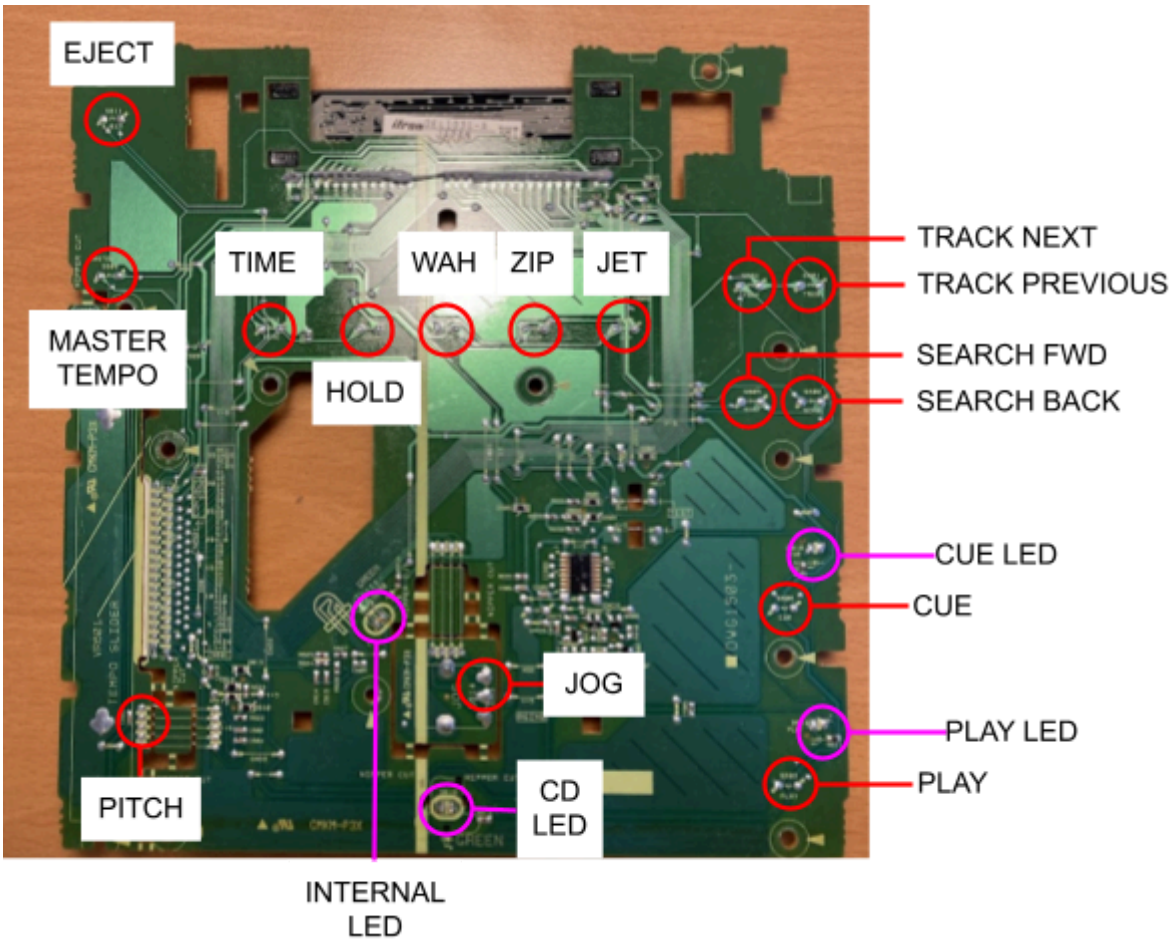


Note: any board which is compatible with Arduino and follows the above requirements with the desired Pins will work.

So we need to solder all the pins to our board. This is what I did for the Teensy LC:

Name	Type	In/out	PIN Number
Eject	Button	IN	0
Track Search Previous	Button	IN	1
Track Search Next	Button	IN	2
Search Back	Button	IN	3
Search Forward	Button	IN	4
Cue	Button	IN	5
Play/Pause	Button	IN	6
Jet	Button	IN	7
Zip	Button	IN	8
Wah	Button	IN	9
Hold	Button	IN	10
Time mode / Auto Cue	Button	IN	11
Master Tempo	Button	IN	12
Pitch	Slider	IN	14 + 3,3V
Jog Wheel	Encoder	IN	15 & 20
Cue Led	Led	OUT	16
Play Led	Led	OUT	17
Internal Led	Led	OUT	18
CD Led	Led	OUT	19
Browse (addition)	Encoder	IN	21 & 22 +3,3V
Load (addition)	Button	IN	23

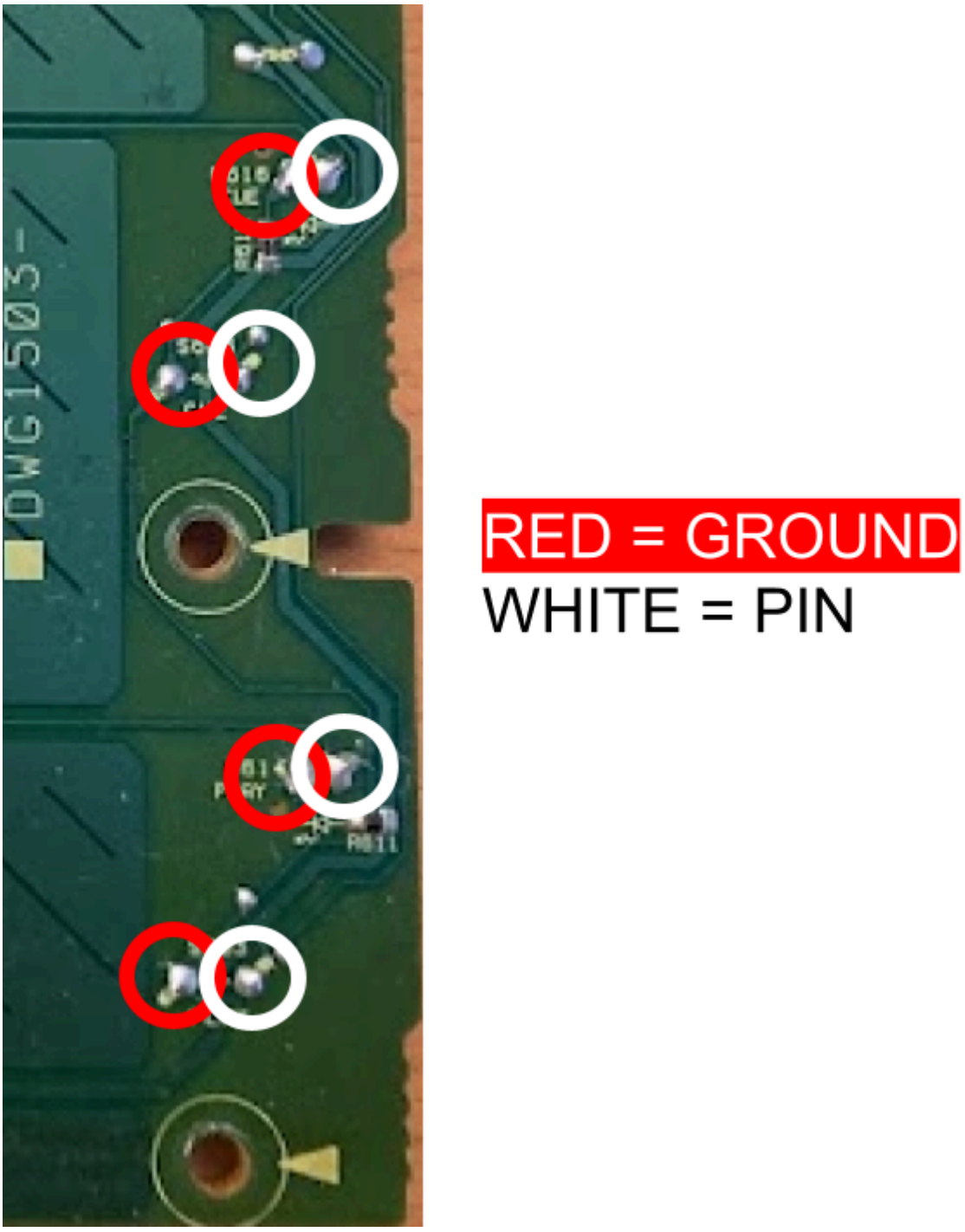
Now we need to know where we have to solder the board, so we identify where are them, using the multimeter:



Note: each pin must be connected to the ground too, this means we need the pin and a second wire to be grounded to every pin.

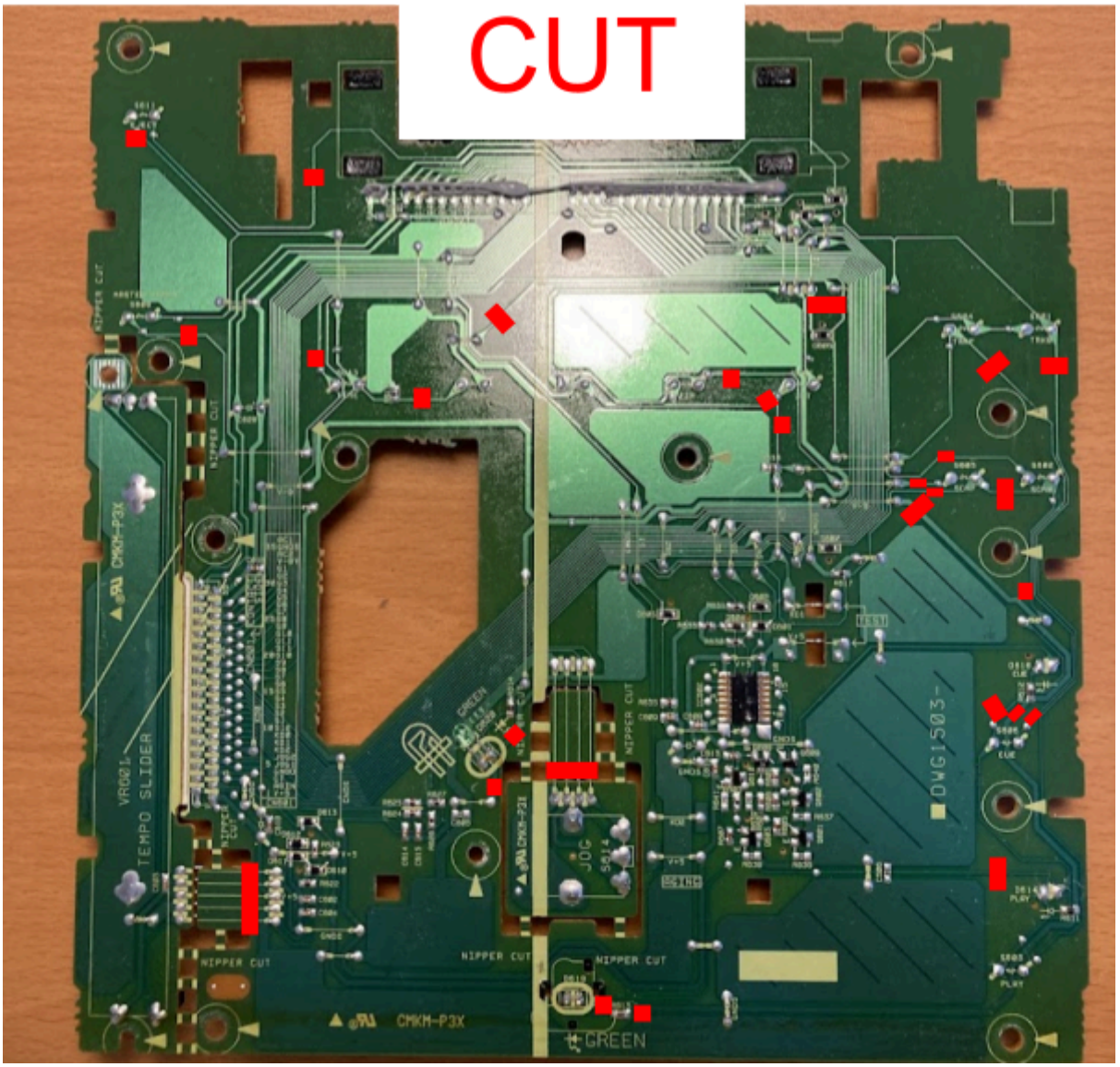
If you are not used to this, basically you have to make a new ground line and this ground line will be soldered to the arduino board to the ground pin.

In this figure, Play and Cue buttons and leds:

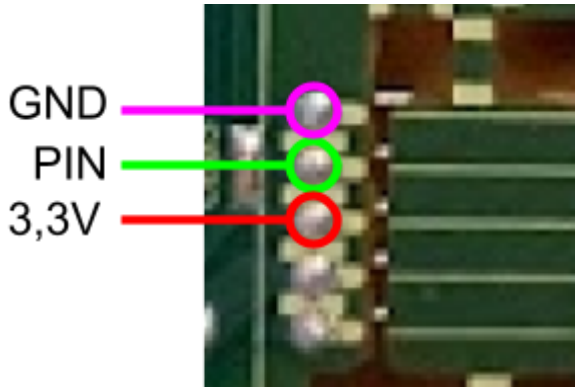


So with every PIN you must put its ground to the ground line. Polarity only matters for the LEDs here.

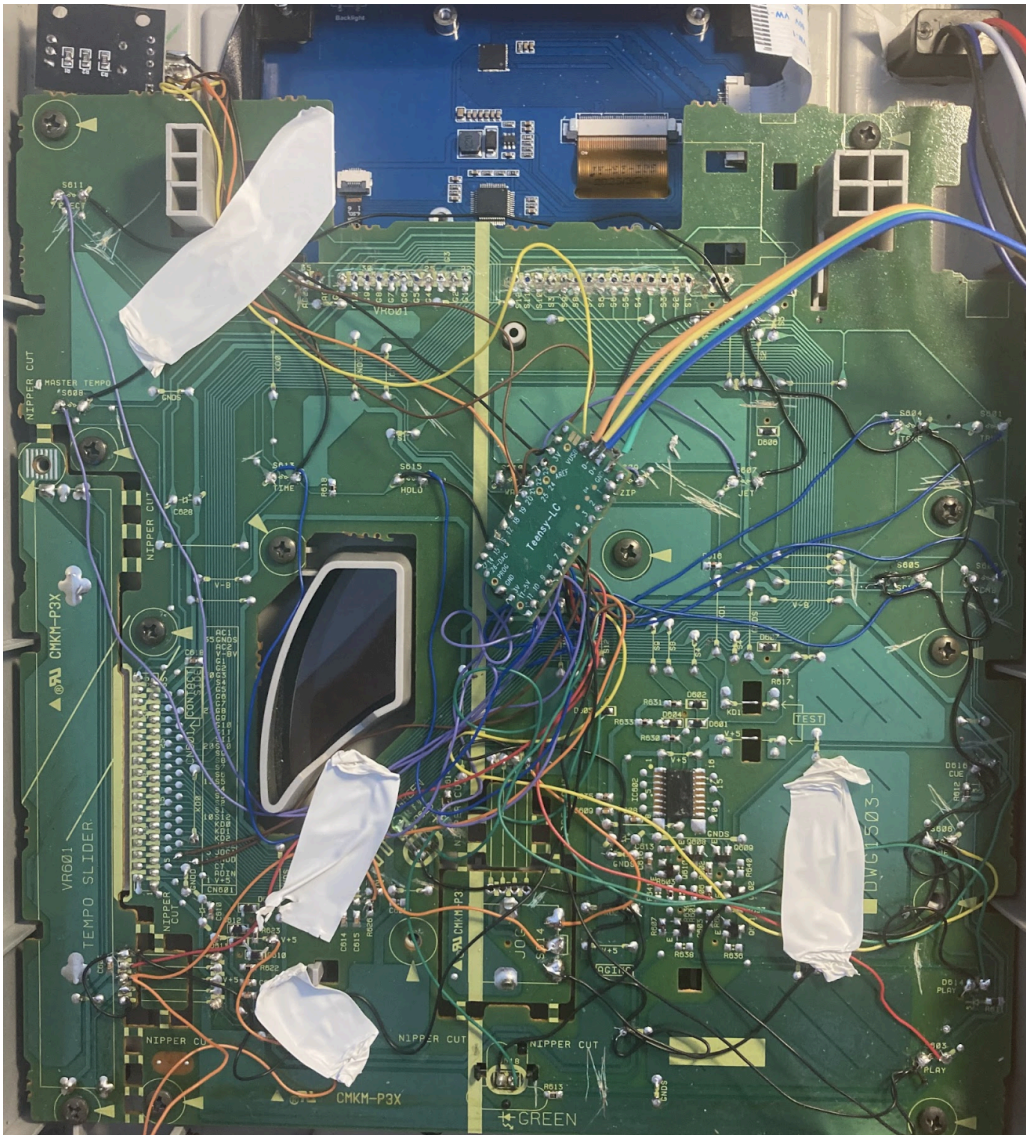
The CDJ100S has also some shorted pins that will make some of our new design not to be working, so we need to carefully cut them in order to make the new ground line. A multimeter is your friend here. We need to cut these:



The Pitch slider has several solder points, after checking in the CDJ service manual and using a multimeter I found these are the ones we have to use:

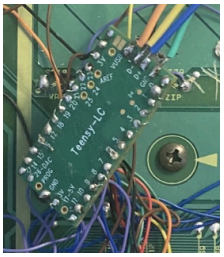


So with all done, it should look something like this:



Note the Teensy board is floating in the center. You can also see the new browser encoder on the top left.

Apart from that, I later soldered the 4 USB pins instead of using an USB connector.



Phase 2: Firmware

Our friend here is the Arduino IDE app. You may ignore this part if you want to upload my code directly.

Implementation

So basically every time we do something, Arduino should send a MIDI message.

Type of MIDI messages:

Note On / Note Off : it can be called from a button or switch.
Control Change (CC): sends different values according to the position of the physical component. This can be called from knobs and sliders (jog, browse and pitch).

Also, the default led is considered a switch, so it has two states: on & off so it receives a Note On / Off.

This sounds easy, and I was not used to it until I saw that there were bounces.

In the real world we have bounces which means that any button, knob, etc, can have interference or physical bounce so we need to add a delay in every Pin so we can avoid a repeat touch. I've found there is already an implementation in the Bounce library that fixes this so we can avoid writing a new code. We only need to include it in our code and call it for every button. In my code, I added 50 ms of delay. This is maybe not the best for the CUE buttons if we want to make fast rhythms, so you may want to change to a lower value.

```
#include <Bounce.h>
Bounce cue_boto = Bounce (cue_pin, 50);
```

For the encoders, we also need to implement if we are going forward or backwards. There is also the Encoder library which keeps track of the direction.

```
#include <Encoder.h>
Encoder jog(jogA_pin, jogB_pin);
long lastPosition_jog = 0;
```

For the Pitch Slider, I've found several ways to make it work, but the ResponsiveAnalogRead library worked the best.

```
#include <ResponsiveAnalogRead.h>
ResponsiveAnalogRead analog(pitch_pin, true);
```

Leds

This device has 4 leds: Cue, Play, Internal and CD. The first two will turn on as expected. The internal and CD are the ones we have to decide what to do. So I decided that the internal will blink to the track beat as Mixxx can send Beat through MIDI. For the CD led, it is going to blink when the track is ending. Mixxx can also send a 'true' value when the track is about to finish.

So the code is basically a *digitalWrite(pin, HIGH)* when the arduino receives the MIDI note. In the example below, I set '61' to be the MIDI note that Mixxx sent while the track is playing or while it is paused.

While playing: Mixxx sends the note 61 with a velocity=127
While paused: Mixxx sends the note 61 with a constant loop velocity of 127 and 0 (blinking).

```
const int PLAY_NOTE_INDICATOR = 61;
[...]
void handleNoteOn(byte channel, byte note, byte velocity) {

if (note == PLAY_NOTE_INDICATOR) {
  if (velocity > 0) {
    digitalWrite(ledPlay, HIGH); //led enabled
  }
  else {
    digitalWrite(ledPlay, LOW); //led disabled
  }
```

For the CD led, I set the note to be 64. Then Mixxx sends 'true' if the track is ending and 'false' if it's not. I needed to make a loop to make a blink.

```
#include <elapsedMillis.h>
const int LEDCD_NOTE_INDICATOR = 64;
bool blink = false;
unsigned long previousTime = 0;

if (note == LEDCD_NOTE_INDICATOR) {
   blink = (velocity > 0);
   }
}
if (blink){
 if (millis () - previousTime >= 1000){
   previousTime = millis();
   digitalWrite(ledCd, !digitalRead(ledCd));
 }
}
else{
   digitalWrite(ledCd, LOW);
}
```

Main setup

For every pin we need to set if it's an INPUT and OUTPUT:

```
pinMode(eject_pin, INPUT_PULLUP);
[...]
pinMode(ledCue, OUTPUT);
[...]
```

We also need to initialize our MIDI receive functions and both encoders.

```
usbMIDI.setHandleNoteOn(handleNoteOn);
usbMIDI.setHandleNoteOff(handleNoteOff);
jog.write(0);
browse.write(0);
lastPosition_browse = 0;
```

Void Loop

We need to update every component and define what they do. For the buttons, I used the '*fallingEdge*' and '*risingEdge'* to read the state (pressed / unpressed):

```
eject_boto.update();

if(play_boto.fallingEdge()){
  usbMIDI.sendNoteOn(60, 127, channel);
 }
 if(play_boto.risingEdge()){
  usbMIDI.sendNoteOff(60, 0, channel);
 }
```

Midi values

These are the MIDI values I implemented. You can use any other. The translated hex note is important for the mapping later.

Name	Midi Channel	Type	MIDI Note	Hex Note
Eject	1	Note On	63	0x3F
Track Search Previous	1	Note On	64	0x40
Track Search Next	1	Note On	65	0x41
Search Back	1	Note On	66	0x42
Search Forward	1	Note On	67	0x43
Cue	1	Note On	61	0x3D
Play/Pause	1	Note On	60	0x3C
Jet	1	Note On	68	0x44
Zip	1	Note On	69	0x45
Wah	1	Note On	70	0x46
Hold	1	Note On	71	0x47
Time mode / Auto Cue	1	Note On	72	0x48
Master Tempo	1	Note On	62	0x3E
Pitch	3	CC	7	0x07
Jog Wheel	2	CC	20	0x14
Cue Led	1	Note On	62	0x3E
Play Led	1	Note On	61	0x3D
Internal Led	1	Note On	63	0x3F
CD Led	1	Note On	64	0x40
Browse (addition)	3	Note On	70 & 71	0x46 & 0x47
Load (addition)	1	Note On	60	0x3C

Apart from that, we must know the hexadecimal translations for the message type. The message also contains the MIDI channel:

Note On: 0x90, 0x91, 0x92, … (90 for MIDI channel 1, 91 for channel 2, 92 for channel 3)
Note Off: 0x80, 0x81, 0x82, …
CC: 0xB0, 0xB1, 0xB2, … (B0 for MIDI channel 1, B1 for channel 2, B2 for channel 3)

Phase 3: Midi Mapping

Mixxx allows us to make custom mappings. These mappings are made using XML code which is similar to HTML. It's basically a tree where you can define what MIDI message makes software action. This goes in conjunction with an optional Javascript file, which allows us to make complex mappings, variables or making a button to make more than one action.

Now, we should define what we want to do with every button. For this model, these are the actions I decided to map:

Name	Midi Channel	Type	Hex Note	Action	Second action (shift)
Eject	1	Note On	0x3F	Browse Back	-
Track Search Previous	1	Note On	0x40	Loop Half	-
Track Search Next	1	Note On	0x41	Loop Double	-
Search Back	1	Note On	0x42	Search Back	Faster Search back
Search Forward	1	Note On	0x43	Search Forward	Faster Search forward
Cue	1	Note On	0x3D	Cue	Go to start
Play/Pause	1	Note On	0x3C	Play/Pause	-
Jet	1	Note On	0x44	Button 1	Depends on button mode
Zip	1	Note On	0x45	Button 2	Depends on button mode
Wah	1	Note On	0x46	Button 3	Depends on button mode
Hold	1	Note On	0x47	Shift Button	-
Time mode / Auto Cue	1	Note On	0x48	Button Mode	-
Master Tempo	1	Note On	0x3E	Master Tempo	Tempo Range
Pitch	3	CC	0x07	Pitch	-
Jog Wheel	2	CC	0x14	Jog	-
Cue Led	1	Note On	0x3E	Cue Indicator	-
Play Led	1	Note On	0x3D	Play Indicator	-
Internal Led	1	Note On	0x3F	BPM Led	-
CD Led	1	Note On	0x40	End warning	-
Browse (addition)	3	Note On	0x46 & 0x47	Browse Up/Down / Waveform Zoom in/out	-
Load (addition)	1	Note On	0x3C	Load Track / Load playlist	-

I wanted to have multiple modes (layers), so the three original *Jet, Zip & Wah* can make different actions.

I decided to have 6 layers / button modes:

1st MODE: HOT CUE: A, B, C
2nd MODE: HOT CUE: D, E, F
3rd MODE: HOT CUE: G, H
4th MODE: LOOP ROLL: ¼, ½, 1
5th MODE: BEATJUMP: LEFT, RIGHT, DEFINE LENGTH
6th MODE: KEYSHIFT: SEMITONE DOWN, SEMITONE UP, RESET

To implement the different modes, we can create a variable, but the issue here is that we should have visual feedback on the skin and after doing some research I did not find a way to have this visual variable in the skin, so I managed to map every mode to a non used parameter in the mixer. So these are the EQ Kill, so:

1st MODE: DECK2 LOW KILL
2nd MODE: DECK2 MID KILL
3rd MODE: DECK2 HI KILL
4th MODE: DECK3 LOW KILL
5th MODE: DECK3 MID KILL
6th MODE: DECK3 HI KILL

Of course we can even add the DECK4 (and DECK1 as we are not using the internal mixer here), but I kept it to 6 modes.

The Javascript file has to be included in the XML:

```
<scriptfiles>
        <file filename="XDJ100SX.js" functionprefix="XDJ100SX" />
    </scriptfiles>
```

The play button is that simple:

```
<control>
    <group>[Channel1]</group>
    <key>play</key>
    <description>PLAY</description>
    <status>0x90</status>
    <midino>0x3C</midino>
    <options>
    <normal/>
    </options>
</control>
```

So the main parameters are:

group: Deck number
key: mixxx action
status: MIDI message type
midino: MIDI message note

But let's see now the CUE button, which I want also that a Shift button + CUE make the player go to the beginning of the track (like any other controller):

```
<control>
            <group>[Channel1]</group>
            <key>XDJ100SX.cue</key>
            <description>CUE</description>
            <status>0x90</status>
            <midino>0x3D</midino>
            <options>
            <script-binding/>
            </options>
        </control>
```

So:

key: name_of_the_javascript_object.name_of_the_function
options: we have to add <script-binding/> to let it know that this 'key' is in the javascript file

It calls the function *XDJ100SX.cue* that I declared in the Javascript file.

Javascript

I took some variables from some other controllers so I could easy implement them into my code.

Additionally, the toggle button mode that toggles between the different EQ Kills:

```
//Button Mode

XDJ100SX.buttonMode = function (channel, control, value, status, group) {

  if (value > 0) {
    // Go to next mode
    XDJ100SX.currentMode = (XDJ100SX.currentMode + 1) % 6;

    // Set all to 0
    engine.setValue("[Channel2]", "filterLowKill", 0);
    engine.setValue("[Channel2]", "filterMidKill", 0);
    engine.setValue("[Channel2]", "filterHighKill", 0);
     engine.setValue("[Channel3]", "filterLowKill", 0);
    engine.setValue("[Channel3]", "filterMidKill", 0);
    engine.setValue("[Channel3]", "filterHighKill", 0);

    // Enable the current one
    if (XDJ100SX.currentMode === 0) {
        engine.setValue("[Channel2]", "filterLowKill", 1); //Mode 1
    } else if (XDJ100SX.currentMode === 1) {
        engine.setValue("[Channel2]", "filterMidKill", 1); //Mode 2
    } else if (XDJ100SX.currentMode === 2){
        engine.setValue("[Channel2]", "filterHighKill", 1); //Mode 3
    }
    else if (XDJ100SX.currentMode === 3){
        engine.setValue("[Channel3]", "filterLowKill", 1); //Mode 4
    }
    else if (XDJ100SX.currentMode === 4){
        engine.setValue("[Channel3]", "filterMidKill", 1); //Mode 5
    }
    else{
        engine.setValue("[Channel3]", "filterHighKill", 1); //Mode 6
    }
  }
};
```

And then an example for the 3 buttons in the first mode (currentMode = 0):

```
XDJ100SX.button = function(buttonNumber){
  return function (channel, control, value, status, group){
    if(value === 127){

      //Hot Cue A, B, C
      if(XDJ100SX.currentMode === 0){
        if(XDJ100SX.shiftPressed){
          engine.setValue(group, "hotcue_" + buttonNumber + "_clear", 1);
        }
        else{
          engine.setValue(group, "hotcue_" + buttonNumber + "_activate", 1);
        }
      }
```

As I added a browse encoder, I mapped the button to make different things:

When we see the waveform -> zoom in / zoom out
When we see the library -> scroll up / down

In Mixxx we can also switch between the *Tabs* declared in the Skin, so:

```
XDJ100SX.browseDown = function(channel, control, value, status, group) {
  if (value === 127) {
    // llegim el control actual
    var currentTab = engine.getValue("[Tab]", "current");
    if (currentTab === 0) { // 0 = overview
      engine.setValue("[Channel1]", "waveform_zoom_down", 1); // 1 = library
    }
    // Enviem moviment al browser
    engine.setValue("[Library]", "MoveDown", 1);
  }
};

XDJ100SX.browseUp = function(channel, control, value, status, group) {
  if (value === 127) {
    var currentTab = engine.getValue("[Tab]", "current");
    if (currentTab === 0) {
      engine.setValue("[Channel1]", "waveform_zoom_up", 1);
    }
    engine.setValue("[Library]", "MoveUp", 1);
  }
};
```

Phase 4: Skin

The goal here is not only to integrate the Skin to the small 5 inches touchscreen but also to make it look like a standard Pioneer/AlphaTheta deck as most of the DJs are used to this layout.

After doing some research, I found the user *Timewasternl* showed a video where he contributed at creating a custom Mixxx skin that looked similar to a 2 Pioneer deck so he converted a DDJ-400 controller into a standalone controller. So I took that skin and customized my way.

It's a bit tricky to work with the sizes here if you are not familiar with HTML and CSS.

The Mixxx skin works with XMLobjects and QSS which is similar to CSS code.

The first thing is to declare that we will use only 1 deck in the skin.xml file:

```
<attribute config_key="[Master],num_decks">1</attribute>
```

We have also to declare the different *Tabs* the skin will have, so in this case I copied the CDJ3000 tabs:

1: Overview (waverform, main deck view)
2: Library
3: Hot Cue (to be able to trigger hot cues from the screen - like the XDJ 1000 MK2)
4: Beat Loop
5: Beat Jump
6: Key Shift
7: Stems

Note: the Stems feature is available from Mixxx 2.6 which is the one I used. To work with Stems you must analyze them before using a third party app. At this moment, Mixxx can't analyze it.

The deck.xml file contains the bottom part of the deck (quantize button, time, overview waveform, pitch, tempo, key, and also **the button mode with a custom TEXT**:

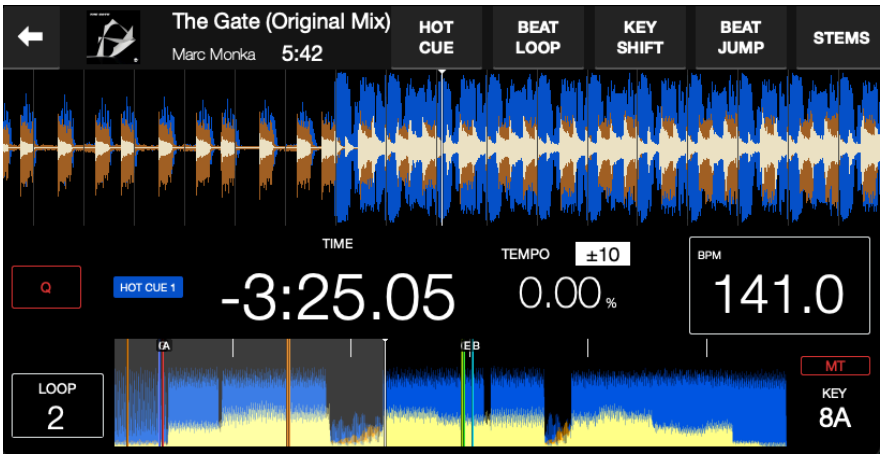
```
<                    <Label>
                    <ObjectName>Mode1</ObjectName>
                    <Size>65f,40f</Size>
                    <Text>HOT CUE 1</Text>
                    <Connection>
                     <ConfigKey>[Channel2],filterLowKill</ConfigKey>
                     <BindProperty>visible</BindProperty>
                    </Connection>
                    </Label>

                    <Label>
                    <ObjectName>Mode2</ObjectName>
                    <Size>65f,40f</Size>
                    <Text>HOT CUE 2</Text>
                    <Connection>
                     <ConfigKey>[Channel2],filterMidKill</ConfigKey>
                     <BindProperty>visible</BindProperty>
                    </Connection>
                    </Label>

                    <Label>
                    <ObjectName>Mode3</ObjectName>
                    <Size>65f,40f</Size>
                    <Text>HOT CUE 3</Text>
                    <Connection>
                     <ConfigKey>[Channel2],filterHighKill</ConfigKey>
                     <BindProperty>visible</BindProperty>
                    </Connection>
                    </Label>

                    <Label>
                    <ObjectName>Mode4</ObjectName>
                    <Size>65f,40f</Size>
                    <Text>ROLL</Text>
                    <Connection>
                     <ConfigKey>[Channel3],filterLowKill</ConfigKey>
                     <BindProperty>visible</BindProperty>
```

Apart from all XML files, I modified the original QSS file, so I wanted to put the same colors, font type, weight and borders like in any other Pioneer/AlphaTheta deck.



I also modified the startup image and progress bar with the project's logo:

The only thing I could not change is the library layout. According to the Mixxx community, it's included in the build, so it should be modified in the main C files and the re-build.

Phase 5: Linux System

In this stage, we just need to install the operating system, install Mixxx and then transfer both Skin and MIDI mapping through SCP / SSH. Of course the mapping and skin will work in any other environment such as Windows or MacOS.

For this project I used the Raspberry Pi OS that you can download (or use their app) from the Raspberry site. This OS is based on Linux.

I used the Pi OS Lite for this project, and I wanted to avoid loading the Desktop and to have additional apps and services.

When following the install instructions, you have to configure your WiFi, give the device a host name and also a username and password to access it. Then you can enable SSH.

So all the rest is made using any other computer connected to the same network, through the terminal using SSH.

Now we need to download Mixxx, configure, build and install. You should follow the Mixxx website to build it for your Linux system.

Depending on your system and OS, it might require installing some other dependencies other than the ones included with the Mixxx files.

Now that we have Mixxx installed we need to configure our operating system to run Mixxx on boot. Apart from that, we want to hide all startup messages, warnings and the splash screen.

Depending on the OS Linux system, it can be a bit different, so try and experiment with the different options.

To start Mixxx, I'm using openbox, a lightweight window manager:

sudo apt install openbox

It might require some dependencies, install them all.

And then we have to include this to the boot using a .xinitrc file:

nano ~/.xinitr

Add the following (replace xdj100sx for the current user) (also use the Mixxx installation directory)

```
#!/bin/bash

# Start the window manager
openbox &

# Run Mixxx
exec /home/xdj100sx/mixxx/build/mixxx
```

Save and exit.

Now add run permissions to the file:

chmod +x ~/.xinitrc

Now we want to hide our mouse cursor, as we want to have the maximum user experience as any other standalone deck.

Create bash_profile file:

nano ~/.bash_profile

Add the following:

```
if [ -z "$DISPLAY" ] && [ "$(tty)" = "/dev/tty1" ]; then

  startx -- -keeptty >~/.xorg.log 2>&1 -nocursor

fi
```

Save and exit.

Send your files using scp from a local machine:

scp -r /path/to/local/dir user@remotehost:/path/to/remote/dir

Linux does not have all the fonts by default, so we need to install additional font types in order to have the skin to look like it was designed. I've found a free version of Heveltica Neue was the best. Also, we need some additional fonts to show arrows and icons like Beatjump left/right icons.

Create this folder:
mkdir -p ~/.local/share/fonts

Move your downloaded HelvelticaNeue font to the previous folder we created:
mv ~/Downloads/HelveticaNeue.ttc ~/.local/share/fonts/

Install the Noto fonts:
sudo apt install fonts-noto-core

Create this folder and edit the file:

mkdir ~/.config/fontconfig/
nano ~/.config/fontconfig/fonts.conf

Add the following:

```
<?xml version="1.0"?>
<!DOCTYPE fontconfig SYSTEM "fonts.dtd">
<fontconfig>
  <match target="pattern">
    <test name="family">
      <string>Helvetica Neue</string>
    </test>
    <edit name="family" mode="prepend">
      <string>Noto Sans</string>
    </edit>
  </match>
</fontconfig>
```

Update the system fonts:
fc-cache -fv

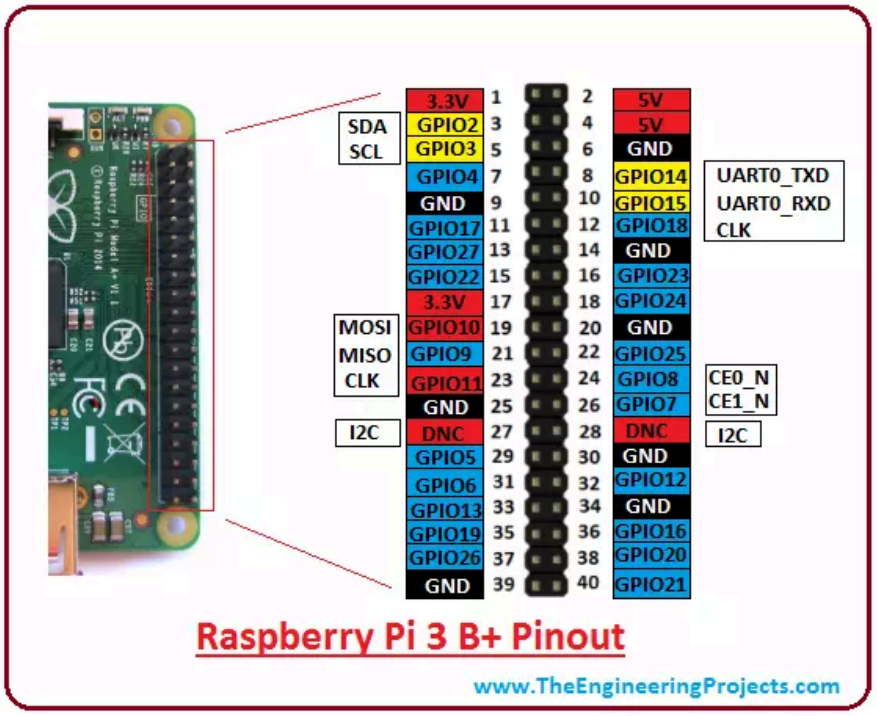
USB Automount

This OS version does not have an “automount” feature, so everytime we connect a USB stick it won’t mount by default. The *USBmount* does this job easily:

git clone https://github.com/rbrito/usbmount.git
cd usbmount
sudo apt install build-essential debhelper fakeroot
dpkg-buildpackage -us -uc -b
cd ..
sudo apt install ./usbmount_0.0.24_all.deb
reboot

Turn on/off

This raspberry pi does not have any switch to turn it on/off. Instead, we can attach the original CDJ switch to the GPIO pins. In our case, following the datasheet, we need to use the GPIO3 and the ground. In this Pi, they correspond to PIN 5 and PIN 6:



So edit:
sudo nano /boot/firmware/config.txt

Add to the end of the file:

dtoverlay=gpio-shutdown,gpio_pin=3,active_low=0,gpio_pull=up

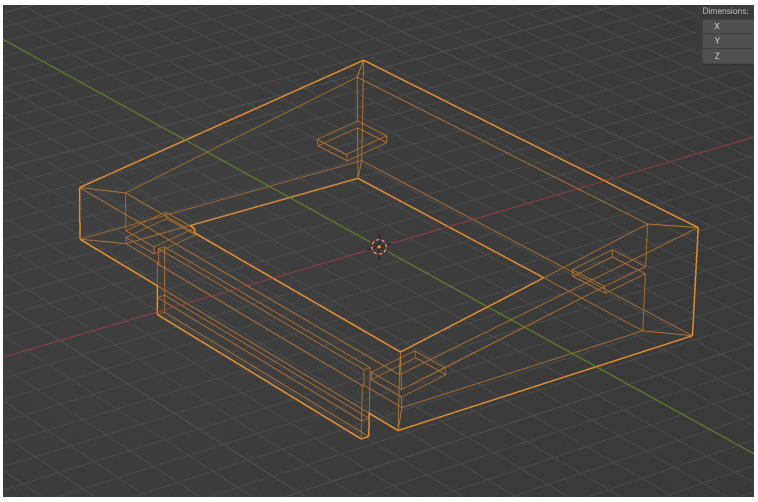
This way Raspberri will turn on when connected to the power, but will turn off using the original switch. If it’s always connected to the power supply, the switch will act as off and also on.

Phase 6: Physical skin

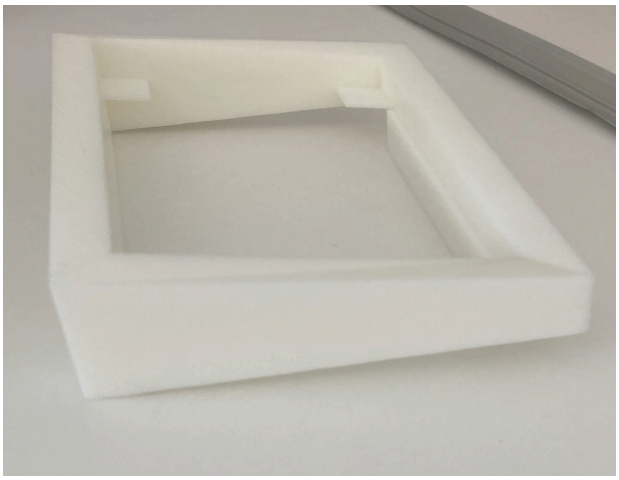
To finish this project completely, I wanted to attach the screen to the device.

First thing, we need to **cut the original screen hole** to fit the size of our new screen:

Now we need to make a 3D model to fit the screen with the screw holes. If you are not used to it, it may take a while. I used Blender to make the model which is also an open source application of 3D modeling.

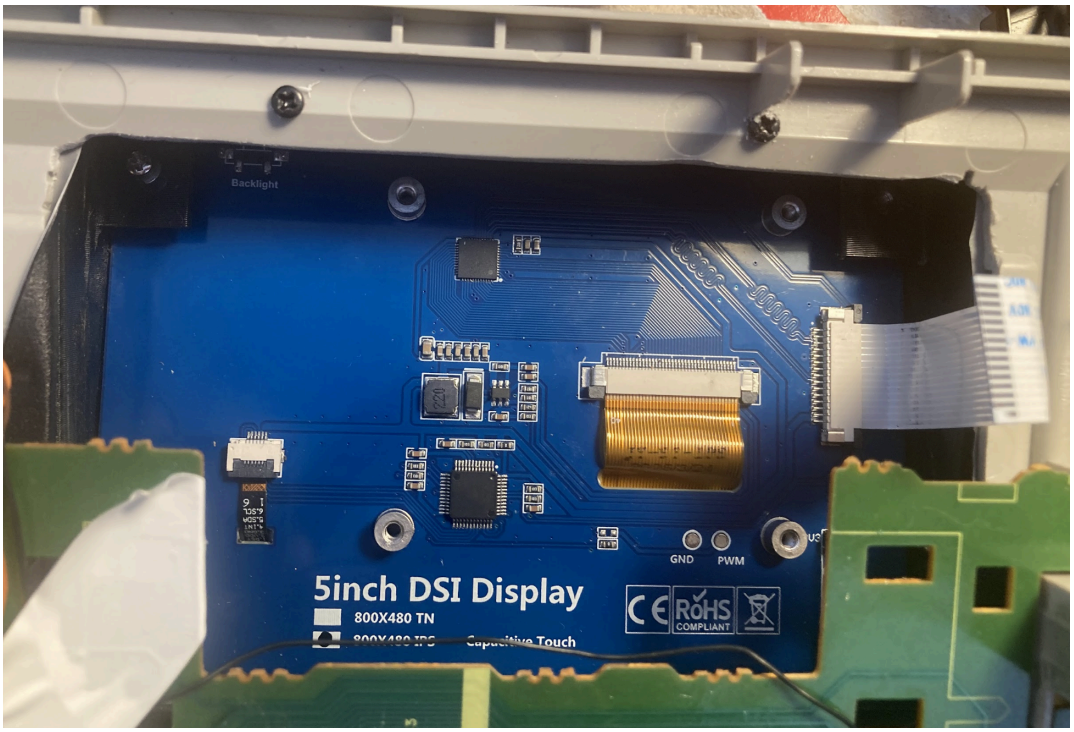


Then I printed a prototype to test. I had to make some adjustments as the screen needed 2 mm to fit in.

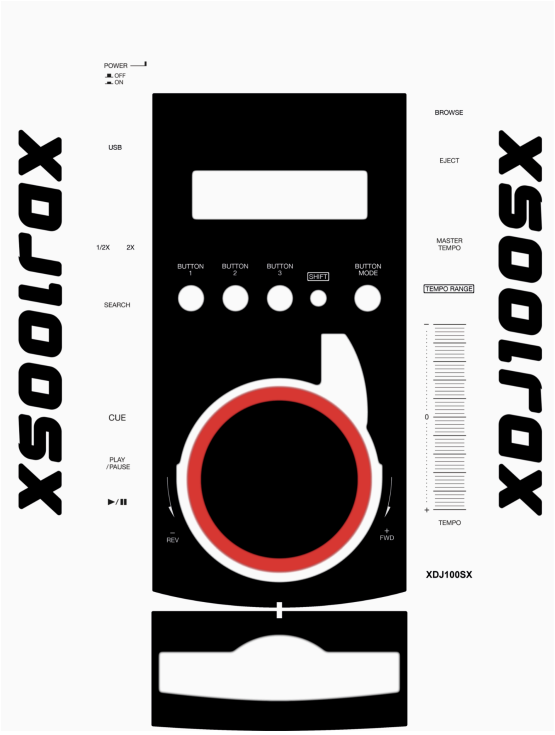


The final result was almost perfect, but I still needed an additional millimeter on the bottom part of the screen. I used a dremel tool to polish that part until the screen fit on it.

I could use the original screws to fix the screen into it and also the original CDJ screws to fit the 3D case into the original case:



Once this was done, I wanted to create an original vinyl printed adhesive to put it in the original case. I've found some companies that make DJ skins so I could download a template to edit and make my own, adding the custom buttons and custom functions.



And so looks that good:

Phase 7: Assemble everything

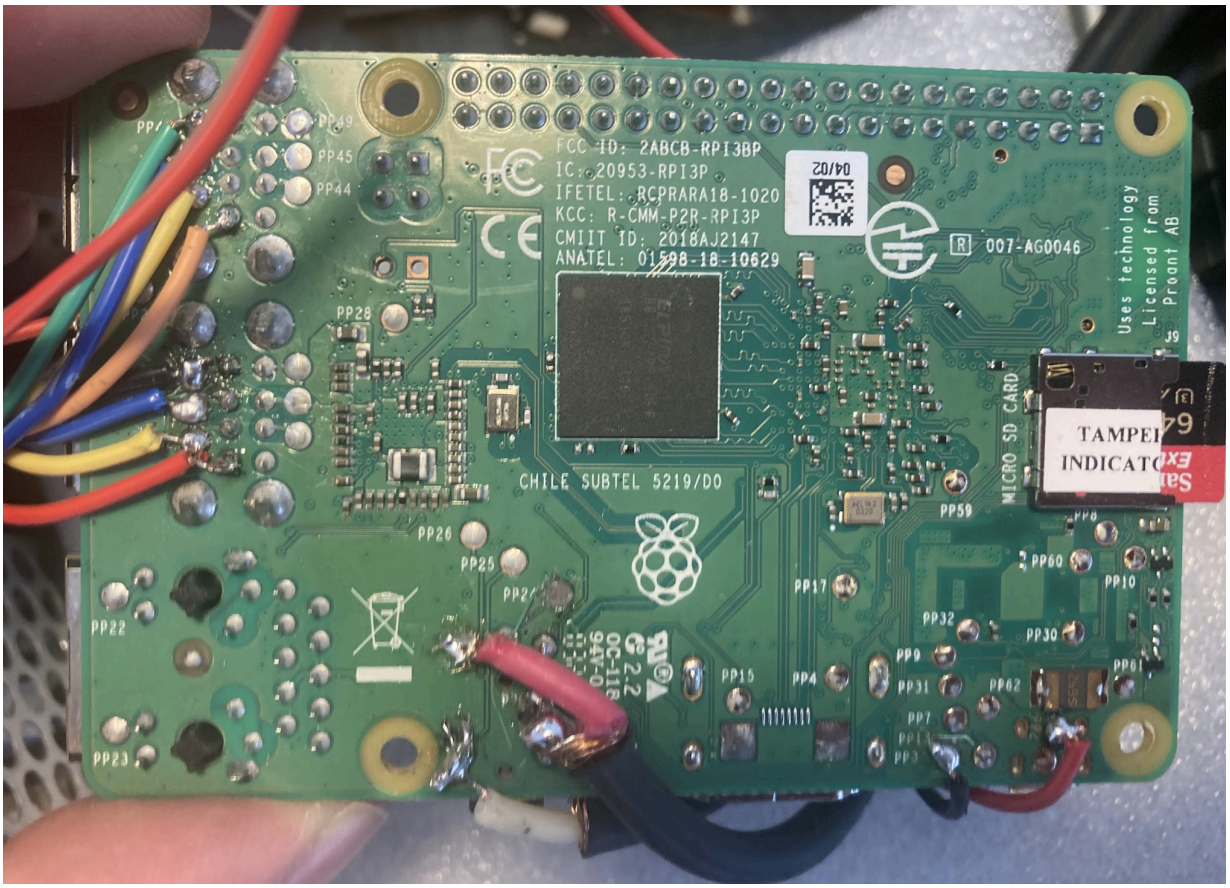
So we are in the final stage: assembling.

We need to:

- Connect Pi to the power supply
- Connect the Arduino Board to the Pi.
- Connect the external USB connector to the Pi.
- Connect the RCA connector to the Pi audio output.

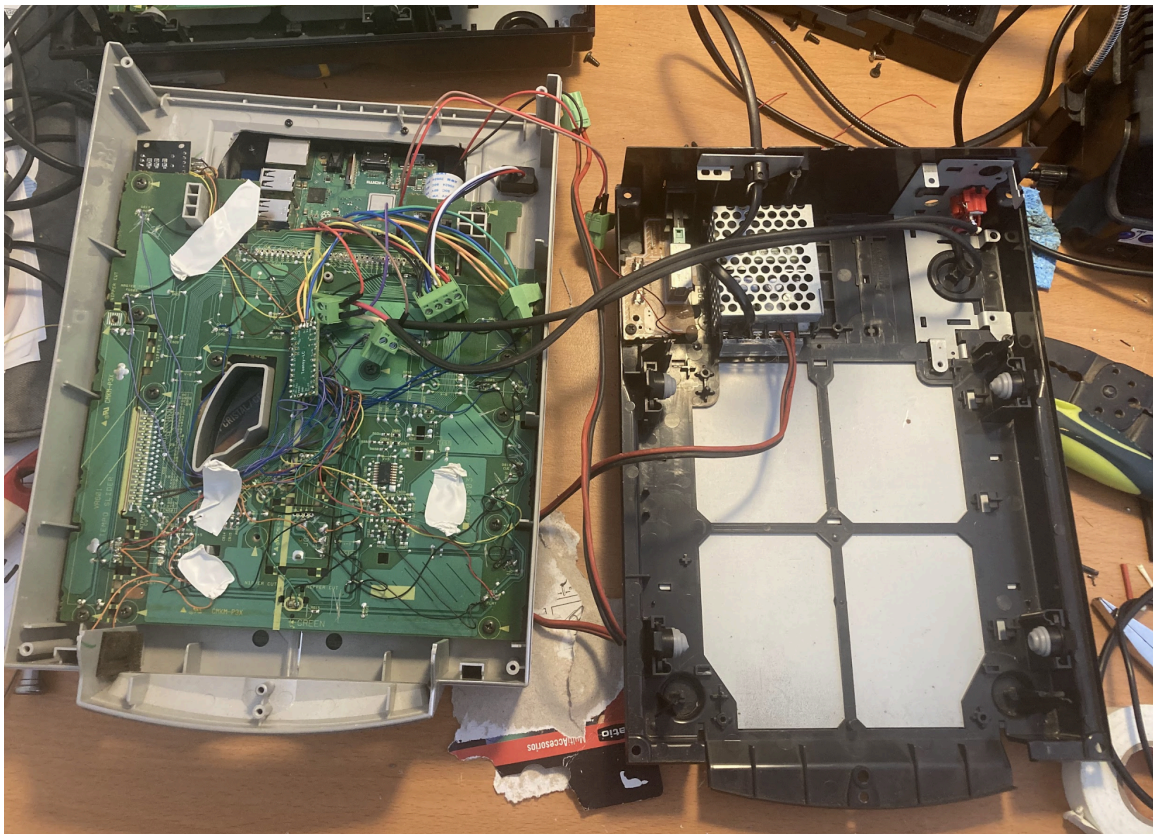
As the 3D case I designed does not have space to include the USB and audio connectors, we need to solder all cables into the Pi directly.

So basically we have to look at the Pi datasheet or use the multimeter to be sure what the connectors are.



Both USB on the left. Audio on the bottom left, and power on the bottom right.

We need the wires to be as long as we can assemble with no issues. I used some terminal connectors to join hardware connections to external connectors.



The power supply is attached to the original case with some silicone.

Before adding all the screws, it's better to test it out, buttons, audio, connectors. If it's all right, then we can put all the screws.

Improvements to do

Make an Ad-Hoc Wifi connection and share the USB through network with a second deck, so they will see each device USB and play their music. (a kind of Pro-DJ Link). Or use the RJ45 connection instead.

Improve the Library Skin so it can look similar to the Pioneer/Alphatheta. To do this, we need to re-build Mixxx.

Add an audio DAC instead of using the original Pi output. The Pi output is quiet and gets easily distorted if we put everything to 0dBFS.

Contact: info@marcmonka.com

Instagram

Facebook