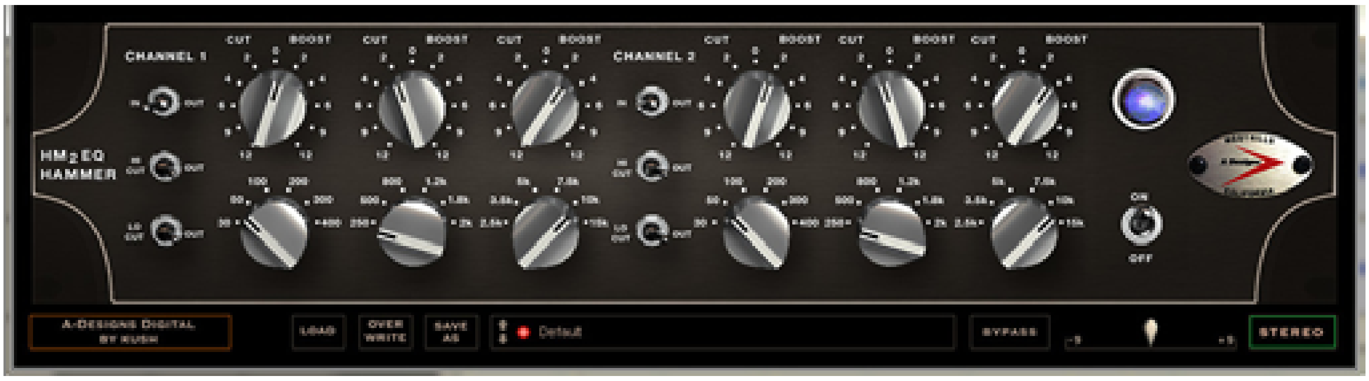
A teensy based Kush EQ controller

The Kush Hammer eq is a vst plugin, based on a dual channel eq, and lends itself fairly easily to having a dedicated controller. If you examine the plugin, it has a relatively small number of controls, based around 2 channels, each with 3 frequency ranges.



Channel 1	3 switches, for in/out,hi-cut/out, and lo-cut/out 3 controls for cut/boost 3 controls for frequency
Channel 2	is a repeat of channel 1
2 modes	stereo linked, or dual mono
And of course,	
	power/bypass buttons.

There is also an output gain trim, and some preset load/save/overwrite functions.

So I set about constructing a hardware controller for this software plug-in, as I find using a mouse frustrating at the best of times. I needed to think a little about the kind of controls that I like to use, and came up with a few ideas.

It seemed sensible to control the cut/boost and frequency controls with rotary encoders, rather than simple potentiometers, as this would allow two-way communication between the controller and the plugin, so that changes in one of them would reflect in the other. The cut/boost are smooth controls, and the frequency controls are switched, so it makes sense to use a smooth encoder for the cut/boost, and detented encoders for the frequency controls. I decided to make up just one set of controls, to control stereo mode, and have a function to switch the controls to control left/right channels seperately if using in dual mono. I did this by using rotary encoders with push click buttons for the frequency controls, and using these buttons to select left/centre/right on the three encoders. I wanted some visual feedback, so incorporated an LCD screen to display values, and to assist in preset management. An optical encoder was also used for the output trim.

I've found preset management rather frustrating in the past in cubase, so I thought that I'd copy all the midi controller values of the presets to EEPROM, and have this hardware controller respond to midi program change. I presume that this could be done equally well with an SD card. This requires naming the presets, and I incorporated a ps/2 keyboard port via a keyboard module.

After a little more thought, I concluded that the requirement in components would be:

An enclosure

11 LED button switches

7 rotary encoders, with knobs

4 bearings, for shaft extension

4 shaft couplers

2.8cm LCD display

PS/2 keyboard interface / port

EEPROM chip

Power supply

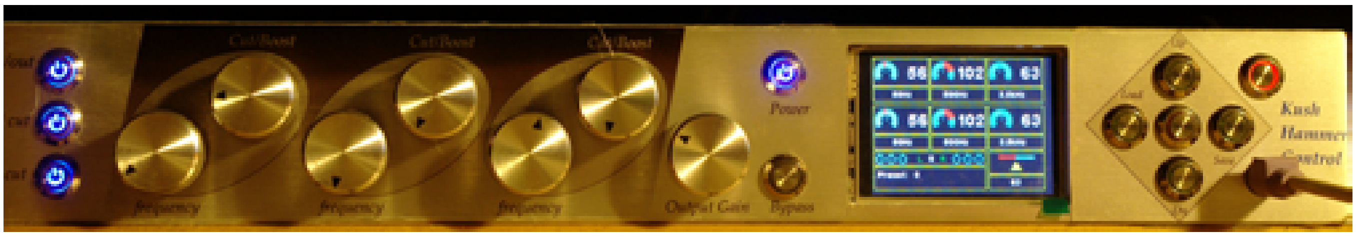
Hardware 5pin MIDI ports

MCP23017ES GPIO IC, x3

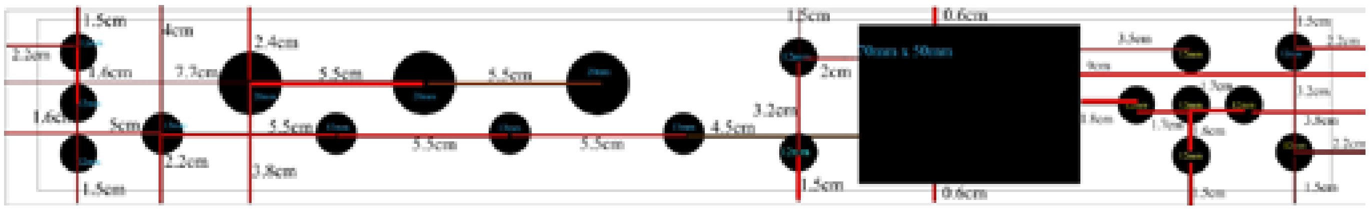
4116R DIL resistor IC, x2

Teensy 3.2 microcontroller

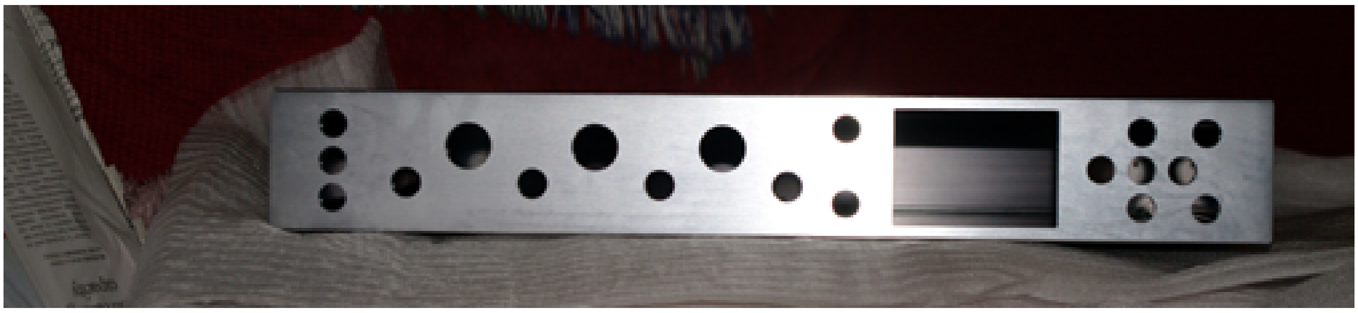
Here's how the project worked out:



The enclosure I bought from Breeze Audio, via AliExpress. If you send them a diagram, they'll cut holes, and I've found them extremely helpful in pointing out where I've made errors in diagrams eg where holes would conflict with supports or retaining screws of the enclosure. Highly recommended. This 4306 enclosure came to around 60 pounds all in, and feels nice and solid. This is the diagram I sent them, to have the holes cut:



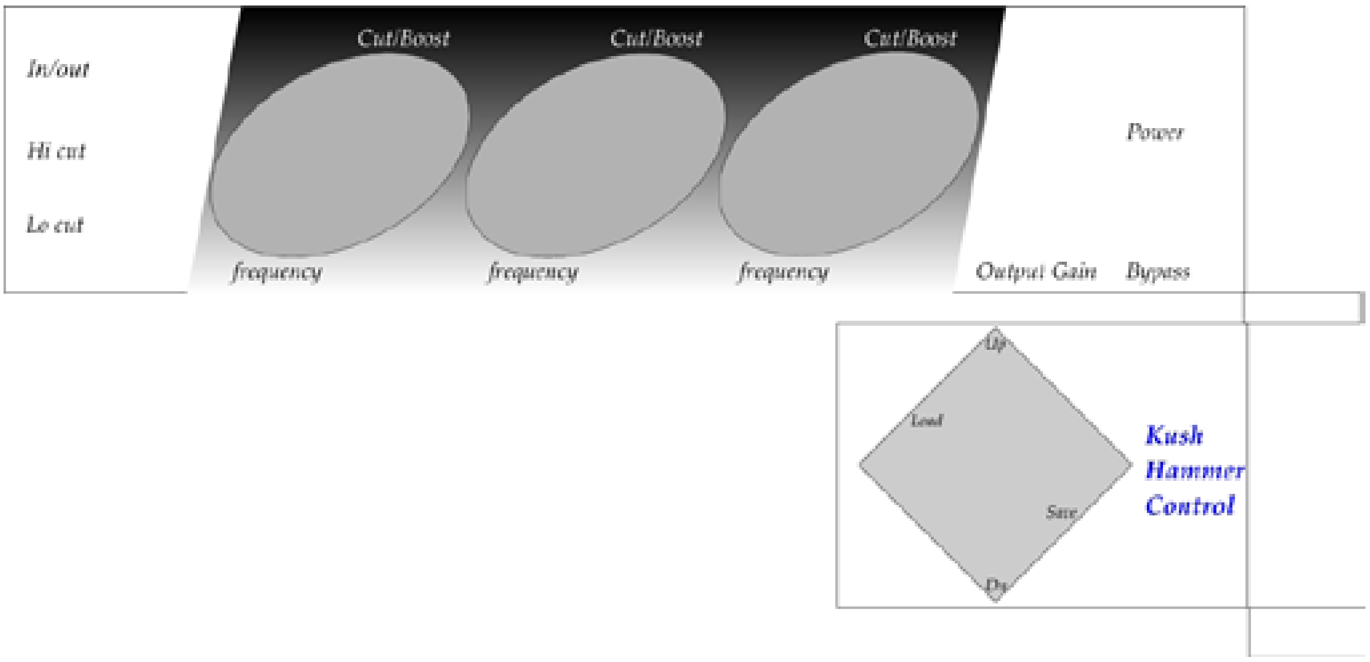
and this is how it arrived:



This was perfect, and comprised

> 12 x 12mm holes, 11 for LED button switches and 1 for the PS/2 port
>
> 3 20mm holes for optical rotary encoders
>
> 4 13mm holes for bearings for shafts to the remaining rotary encoders
>
> 1 5cm x 7cm rectangular hole, for the LCD

I needed to dremel out the rear a touch for the LCD pcb to fit, but got there. The labelling was done with a trimmed inkjet transparent vinyl window label sticker. This took a couple of attempts to get right, and I basically put the sticker on, then cut through to the holes with a craft knife. The diagram for the sticker was

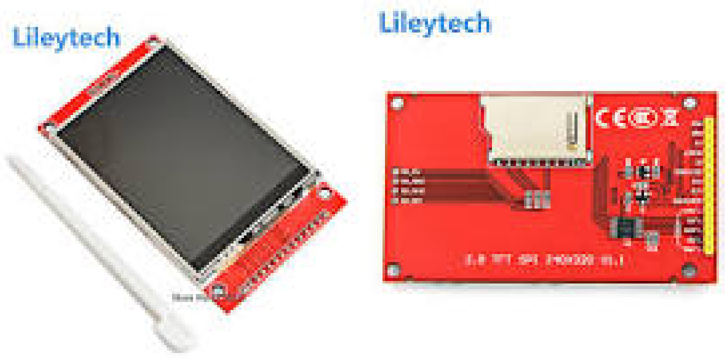


The 13mm holes were for some small bearings, as the bottom row of encoders are on 6mm shafts so I could place them in the case behind the other encoders. So, I used some 6mmx13mmx5mm radial ball bearings, which were £2.50 for 10 on ebay. This was really due to the physical size of the optical encoder bodies - they have a 39mm diameter, so direct placement of one above another takes up quite a bit of room. The shafts are a piece of 6mm aluminium rod, hacksaw'ed to size, and attached to the encoders with 6x6mm CNC stepper motor couplers, around £1 each. The ps/2 port, bearings and the large encoders are held in with epoxy.

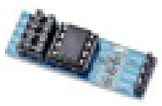
These are the bearings and couplers:

I used 7 rotary encoders, 4 smooth and 3 detented. For these, I used 4 5-24v 600ppr optical encoders from ebay, around 8 pounds each including postage, and 3 cheap ky-40 encoders on mounted boards, which are 20 detents per rotation and incorporate a push switch, on ebay at around 5 pounds for 10 of them.

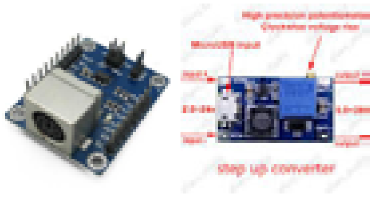
The LCD display is a 2.8in spi ili9341 touch screen, around 6 pounds.



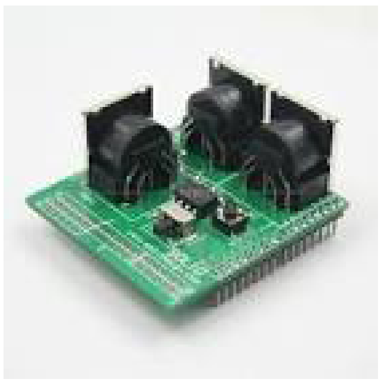
I used an AT24C256 i2c eeprom module, under £1. These have 32k of eeprom, which is plenty for saving pre-sets. They use the I2C bus, and you can change the I2C address with jumpers. I just left this at base address.



A PS2 keyboard driver module, was around 2 pounds. They say 5V on the board, but I couldn't get them to work with that. I used a small step-up board to increase the voltage to just over 6V from the 5V supply - these were 2 pounds for 5 of them, and have a small screw on the side to change the voltage. Don't just wire them up - they can step up to 24V, and mine where at around 12V output when delivered, so measure and adjust first. The ps/2 connector on the front of the case is the end of a male to female ps/2 extension cable, so that I could put the keyboard module elsewhere in the case as its a touch bulky. I I trimmed off some of the plastic on the end connector to fit it into a 12mm hole, then glued it in place.

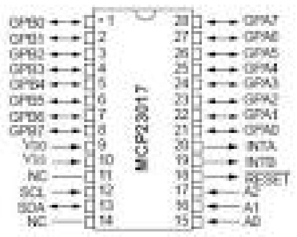


The MIDI shield's are around £6 on aliexpress. Not really necessary, but means that this controller can function as both usb and physical MIDI. You could probably build one for less, but these are opto isolated, and are easy and convenient to simply wire up to the tx/rx pins, as well as 5V/gnd of course. Note the on/off switch - this shield uses a hardware serial port, so need to switch this to share this port with the arduino environment when programming a chip. This was attached to the rear of the case with epoxy, and I drilled 15mm holes through the case to line up the sockets. The MIDI thru wasn't needed, so access to the port wasn't necessary.



The 12mm LED button switches are around a pound each if you shop around. I've used some with rings, and some that have the power symbol cut into them. There are plenty of suppliers on aliexpress, with different colours and LED voltages available, and I use them in my little projects a lot. 3.3V - 6V led voltages work OK, via resistors. For most of the suppliers I've used, if you've ordered a particular voltage, they send the appropriate resistor with it. Very helpful, at least to begin with. Some don't need a resistor, it's built in, but if you feed an LED without a resistor it won't last long, so a little trial and error is called for. I prefer the high flush buttons for tactility, and they feel fairly positive in use. 4 pins on the rear, 2 for the switch and 2 for the LED.

The MCP23017 io expander chip. These are under a pound each, and very useful. 16 input/outputs, and I've used 3 of them in this, for various uses. 2 are used to write the LEDs for the button switches on one side of the chip, and read the button switch presses on the other side, and the 3rd is used to read the parallel output from the keyboard module. These MCP23017ES use the I2C bus, and you can change the address using three pins on the chip. Three unique addresses are required - each chip needs its own address, and can't conflict with the address of anything else on the i2c bus, such as the eeprom module.



I used a 5v power supply, which I had reclaimed from an old 10/100 network switch. This is used to power everything, so I cut the red wire in the panel mount usb extension cable to the teensy so that I could power the teensy from the psu as well, and the external usb connector is epoxy glued on the rear of the enclosure. The encoders say that they're 5-24v, but powering them directly from the teensy is a little weak, as they'll skip counts or be slow to respond from a usb supply. If you want to use an external supply in addition to USB power, you'll need a common ground - so attach the GND of the dc output from the psu to the GND of the teensy. The reclaimed psu I used is for a 240v ac mains supply, so in the AC input receptacle I ensured that the psu is 2A fused, with an additional 2A line fuse on the DC output, and I earthed the chassis. I'm a little reluctant to cut traces on the teensy chip, which is the other option when powering externally, and don't want to fry the teensy chip by double end feeding it :)

The teensy 3.2 chip. An arduino type chip, but with some excellent extras. Most usefully for this, it has usb-MIDI, so it appears as a class compliant midi device, and it is both 3.3 and 5v tolerant. These are around 25 pounds, so more expensive than say a nano, but the additional functionality, interrupts on every pin (which is perfect for the encoders), and greater programming space is extremely useful. My soldering skills are on the low side, so I used the one with pins that'll fit on a breadboard.

Breadboard, dupont cables, and 4116R DIL resistor network (for the LEDs), came to maybe a fiver. The 4116R chips are basically 8 resistors in a DIL chip, and are very convenient for use on a breadboard. Saves a little clutter, and gives a more secure connection than just pushing wire resistors into holes. I still had to use a couple of resistors, though - the I2C bus needs pull up resistors to work properly, so need to attach 4.7K resistors between the I2C bus lines (pins 18 and 19), and +5V. Without these resistors, the I2C bus will either not work at all, or, even worse, work erratically, making fault finding problematic. I attached the resistors at the MCP23017 connection, for physical convenience more than anything.

The knobs are knurled 25mm cnc aluminium, with a grub screw 6mm shaft hole. These were around £15 for 10. Its suprisingly hard to find any that don't have pointers/dots on them, though.

So, total cost for this project has been around 150 pounds all in. This could certainly be done for much cheaper, of course, but to be honest I'm fed up of using plasticy controllers that feel like toys. I wanted something that feels solid, substantial, is transparent to use, and with simple visual information that is clear and comprehensive. Yeah, I know.

Background / about me

I don't have a background in either electronics or programming. I started looking at these things from scratch around 2 years ago, as a new hobby/interest. Music has been a life-long passion, though, so I have a small bedroom studio, which is used really just as a sort of central control centre for playing, as I don't record very much. I simply want to have everything plugged in, and be able to just play without faffing about for half an hour to set things up. So, I started to look at control surfaces. I own a behringer bcr, a couple of JL fadermasters, and an MCS that I bought from ebay. They're OK as far as they go, but in the end, I'd like something that felt more substantial than the BCR, wasn't in a tabletop/mixer type format so that I could either stack/rackmount one on top of another as horizontal space is limited, and had more visual feedback. So, I thought about building my own - how hard can it be? Hmm. A dark road, indeed :)

General Comments

The sketches for this project were written in arduino 1.66, with the teensyduino extension. I've had a few frustrations with updates of the arduino environment, where libraries that I've used for earlier projects will no longer compile, and having more than one version of arduino installed has caused cryptic problems too, with eg serial monitor not working in one of them. Some updated libraries have slightly different commands, so editing older sketches to reflect new commands is a little time-consuming. Mostly not unresolvable, but have occasionally been a little tricksy for me to get my head around as a relative newbie. As I say, I don't have a background in electronics or programming, but I work shifts, and can get to spend a few uninterrupted hours a week on my hobbies occasionally. Sometimes the frustration of the problem can be rewarding once realised :)

I started by buying some bits and pieces to get started and worked through most of the tutorials, which was useful, and then a bit of experimentation and expansion. I read around quite a bit on the forums, and tried hard not to ask any questions that had not already been asked, or to which the answer could be "let me google that for you"... So, more of a lurker than active, as I don't have enough to offer, but thought I'd throw this project out for consideration. I'm sure that there are better ways to do some of the things I've done here, and my self-taught coding can almost certainly be improved upon. So, I'm probably not as efficient as I could be, but it does work for me.

In Use

The layout of the controller shows 3 LED buttons on the left hand side, these are for

In/out for the selected channel

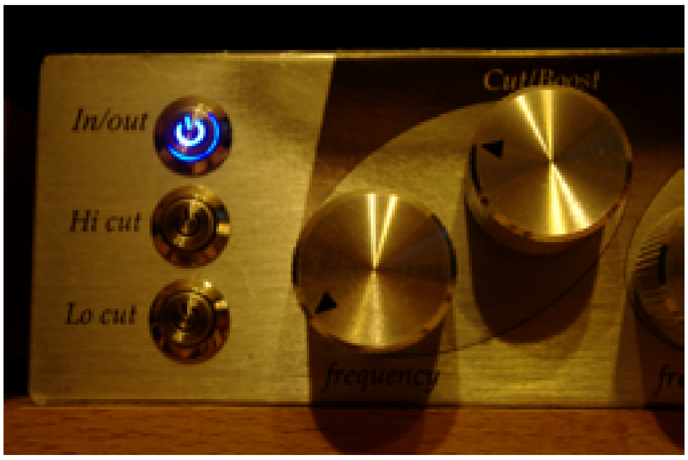
HiCut in/out for the selected channel

LowCut for the selected channel.

There are then 6 encoders in two rows

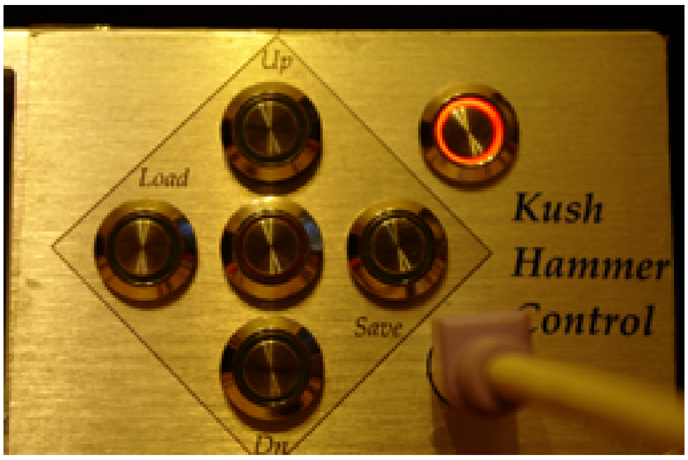
top three are for cut/boost values

bottom three are to select frequency to cut/boost.



The bottom three are push encoders, so can push these encoder to select what channel the encoders will apply to - left encoder to control left channel, middle encoder to select stereo, and right encoder to select right. The LCD will update to show L,S or R in white to show what channel the encoders are operating on, The 3 in/out states will update on the three LED buttons on the left of the controller to reflect the selected channel, and will also display on the LCD. The output gain value is overall, and independent of stereo/left/right. Power and bypass buttons are for plugin use, as I generally use the kush hammer as an insert.

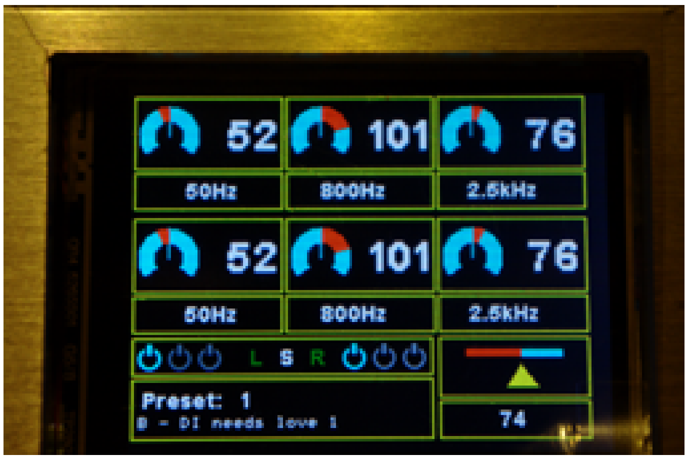
The 5 push buttons to the right of the LCD are for preset management.



Press the load button, and the load button will flash, to show in load mode, and the up/select/down buttons in the middle will illuminate. Use the up/down buttons to scroll through presets, the values will update as you do so. To select a preset, use the centre button, and it will keep all the loaded values. To cancel, press the flashing load button again, which will put the values back to what they were before.

Press the save button, and the save button will flash to show in save mode, and the up/select/down buttons in the middle will illuminate. Use the up/down buttons to scroll through presets, and then select which preset to save to with centre select button. Once selected, the up/down buttons will go through characters, or for speed, can use the output gain encoder to choose a character, and then use the centre select button to choose that character and move to next character. Incorporated into the character set to the bottom is the clear and backspace functions, and at the top of the character set is the finish, to write the data and name to the eeprom. It's also possible to use a ps/2 keyboard to input characters, which will automatically go to next character.

Really, that's it as far as the controller goes. LCD display shows cut/boost value as a gauge and numerically as midi cc value, as well as frequency in Hz. The top 6 are for the left channel, bottom 6 for the right channel, so of course these will read the same if being used in stereo. The in/out states are below that, with the left channel states on the left of the channel indicator, and the right channel states on the right. These can be read top-to-bottom left-to-right. Finally, in the bottom right hand corner of the LCD is the output gain level, shown as a coloured line with pointer, and the midi cc level underneath.



I'd considered taking photographs throughout the build, but realised fairly quickly that its like looking at spaghetti :) So, a talk-through seems far more sensible.

The LCD screen

The SPI ili9341 screen has 320x240 resolution, and I've used various sizes of these screens with the same library. The one I used here is 2.8 inch. This particular display has a touchscreen, but I didn't use it - not enough pins directly available, and I don't really like touchscreens anyway. This is powered with 3.3V from the teensy, and for this controller use:

VCC	3.3V
GND	GND
CS	pin 10
RST	pin 6
DC	pin 9
SDI/MOSI	pin 11
SCK	pin 13
LED	3.3V
SDO/MISO	pin 12

This works well with the excellent marekburiak library, ILI9341_due, and the sketch needs to have

```
#include <SPI.h>//library for the SPI functions, that the LCD screen needs to operate

#include <ILI9341_due_config.h>

#include <ILI9341_due.h>

#include <SystemFont5x7.h>//

#include "Arial_bold_14.h"//a nicer less blocky font for larger characters :) This font file needs to be placed in
the sketch folder, so will need to copy and paste it there

//For the screen, here are the definitions for the pins that the screen is wired to

#define TFT_RST 6
#define TFT_DC 9
#define TFT_CS 10

ILI9341_due tft = ILI9341_due(TFT_CS, TFT_DC, TFT_RST);//this allocates the definitions used. Standard
                                                  hardware SPI pins are assumed for the rest
```

any functions in the library are prefaced by what you've called it - in this case, I just called it tft.

Then, just need to initialise the library in the setup().

The Encoders

Wiring these is pretty straight forward, and I use the standard encoder library.

```
#include <Encoder.h>
```

Nice to use, and straightforward. Define the encoder with

```
Encoder enc1(14,15);//pins for the rotary encoder
```

So the rotary encoder, called enc1, is connected to pins 14 and 15 on the teensy. Using 7 encoders, so each encoder needs to attach to two pins -

```
Encoder enc2(16,17);
Encoder enc3 (20,21);
Encoder enc4 (5,4);
Encoder enc5 (22,23);
Encoder enc6 (2,3);
Encoder enc7 (8,7);
```

Encoders 1,2,3 and 7 are high resolution optical encoders, and encoders 4,5 and 6 are cheaper mechanical encoders with push switches. The optical encoders have 4 wires:

Black	gnd
Red	5V
Green	encoder pin a
white	encoder pin b

You do need to be careful with the wiring, as they come with a warning that gnd/vcc reversed will burn out the output triode.

If the encoders are reading the wrong way round, either swap the pins, or change definition eg

```
Encoder enc1(14,15)
```
to
```
Encoder enc1(15,14)
```

Pin allocations aren't critial, as all pins on the teensy have interrupts, but need to reserve some pins for other uses. Pins 18/19 are used for the I2C bus, and pins 0/1 will be used for tx/rx to the hardware midi port. Just make a note of what you've put where - after a while, peering onto a breadboard with a magnifying glass trying to see through a mass of dupont cables becomes tedious.

The mechanical encoders have 5 pins:

CLK	pin a
DT	pin b
VCC	
GND	
SW	switch - will be attached to a pin on MCP23017 chip.

Each of these encoders needs a couple of variables, for comparison purposes - so, I've just called them np and op for new position and old position.

```
 long op1=0;
 long np1=63;//initial values for the encoder
 long op2=0;//old positions and new positions of all the encoders
 long np2=63;
 long op3=0;
 long np3=63;
 long op4=0;
 long np4=1;
 long op5=0;
 long np5=1;
 long op6=0;
 long np6=1;
 long op7=0;
 long np7=1;
```

Initial values don't matter, as will be allocating values to the encoders later, just need to get them started.

so, to read the values from the encoders is straightforward -

```
        np1=(enc1.read()/16);
```

The new position of the encoder is read; the divide by 16 is because these particular optical encoders are 600 pulses per rotation - far too high a resolution for MIDI, so I just divided it down. Overkill, but they're reasonably affordable and are satisfying in use.

```
        np4=(enc4.read()/4);
```

This divides the quadrature mechanical encoders by 4, so that each detent reads as + or - 1.

At the end of the loop, make the old position the same as the new position, and then in the loop can read the encoder again. Can then compare the new value to the old one to indicate a change.

Buttons and LEDs

Next come the buttons and LEDs. All of these are attached to MCP23017 chips, which are wired to the teensy on the I2C bus, pins 18 and 19. Look at the data sheet, and make sure the chip is the right way round - I've wired these up backwards before now, which has caused me some head-scratching, and

pin	9	VCC	
	10	GND	
	11	not connected	
	12	pin 19 on teensy - SCL	
	13	pin 18 on teensy - SDA	
	15	VCC	Pins 15, 16 and 17 set the I2C address of the chip, 0 to 7
	16	GND	So this configuration gives an address of 1. Important, as
	17	GND	other devices on I2C bus later.
	18	VCC - reset pin. Pulls low to reset chip, so 5V as reset not used here.	

Using this library:

```
#include <Adafruit_MCP23017.h>
```

and will also need

```
#include <Wire.h>
```

Again, straightforwards to use. Need to define with

```
Adafruit_MCP23017 mcp;//so, just named it mcp
```

so, for a second chip, I used

```
Adafruit_MCP23017 mcp2;//so, just named it mcp2, as its the second chip
```

and for the third chip,

```
Adafruit_MCP23017 mcp3;
```

and then in setup(), initialise with

```
mcp.begin(1);//the 1 is the I2C address
```

so for chip 2, use

```
mcp2.begin(7);//the 7 is the I2C address, set with pins 15,16,17
```

and for chip 3, I used

```
mcp3.begin(3);//the 7 is the I2C address, set with pins 15,16,17
```

Still in setup(), need to define whether the pins on the chip are inputs or outputs, as it's a GPIO - so, this is done with

```
        mcp.pinMode (0, INPUT);//inputs for button switches, outputs to LEDs
        mcp.pinMode (1, INPUT);
        mcp.pinMode (2, INPUT);
        mcp.pinMode (3, INPUT);
        mcp.pinMode (4, INPUT);
        mcp.pinMode (5, INPUT);
        mcp.pinMode (6, INPUT);
        mcp.pinMode (7, INPUT);
        mcp.pinMode (8, OUTPUT);
        mcp.pinMode (9, OUTPUT);
        mcp.pinMode (10, OUTPUT);
        mcp.pinMode (11, OUTPUT);
        mcp.pinMode (12, OUTPUT);
        mcp.pinMode (13, OUTPUT);
        mcp.pinMode (14, OUTPUT);
        mcp.pinMode (15, OUTPUT);

        mcp.pullUp (0, HIGH);//built in weak pull up resistors, to reduce button bounce. Work quite well.
        mcp.pullUp (1, HIGH);
        mcp.pullUp (2, HIGH);
        mcp.pullUp (3, HIGH);
        mcp.pullUp (4, HIGH);
        mcp.pullUp (5, HIGH);
        mcp.pullUp (6, HIGH);
        mcp.pullUp (7, HIGH);
```

Wire up the LED button switches so

LED+	output pin
LED-	to GND via resistor (I used the 4116R 200 ohm resistor chip)
Switch	input pin
Switch	ground

So there are 4 wires per button switch - it rapidly starts getting awkward to negotiate all the dupont cables. I used dupont cables with one end trimmed off to solder to the button terminals. Again, how they're wired up doesn't matter, but make a note of what's gone where. Needle nosed pliers are your friend.....

The LED states are used by

eg

```
          mcp3.digitalWrite (14, HIGH);//power button
          mcp3.digitalWrite (15, LOW);//bypass button
```

so, this will put pin 14 on mcp3 to HIGH, which turns on the power LED.

To then read a button switch, use eg

```
button_1_in_out_new=mcp.digitalRead(4);//so, switch button attached to pin GPA4 on the mcp23017
```

Bear in mind that if a button/switch is pressed, it reads 0, and if not pressed it reads 1. Confused me a little to begin with. It's worth testing all the switches and LEDs - my soldering isn't the best, and 4 wires across the LED button is a little cramped. Obviously, the encoder switches don't have associated LEDs, but status will be given a visual indication on the LCD later.

So, the way I wired these up was:

button 1	in/out for channel	mcp(4)	LED	mcp(8)
button 2	hicut for channel	mcp(5)	LED	mcp(9)
button 3	locut for channel	mcp(6)	LED	mcp(10)

button_enc_6	used for select right	mcp(7)
button_enc_5	used for select stereo	mcp(1)
button_enc_4	used for select left	mcp(2)

mcp2 is for the ps/2 keyboard, with the parallel output pins from the module wired to pins 0-7, and the TTL key pressed wired to pin 8.

button_load	loading presets	mcp3(7)	LED	mcp3(12)
button_save	saving presets	mcp3(3)	LED	mcp3(9)
button_power	vst insert power	mcp3(1)	LED	mcp3(14)
button_bypass	plugin bypass	mcp3(0)	LED	mcp3(15)
button_up	up button	mcp3(5)	LED	mcp3(13)
button_down	down button	mcp3(4)	LED	mcp3(10)
button_select	centre button	mcp3(6)	LED	mcp3(11)

The power button for the controller hardware is a latching LED button switch, to switch the 5V from the psu to everything else.

Keyboard Module

The ps/2 keyboard module has 3 outputs, serial, i2c and parallel. However, the i2c output is as an i2c master, at i2c address 4, so I discounted its use over i2c directly so I didn't have more than 1 i2c master - the i2c bus is also used for the mcp23017 chips as well as the eeprom module. The serial port on pins 0 and 1 I used for the hardware midi ports, so I used the 8 line parallel output from the keyboard module, and attached them to pins 0 to 7 on an mcp23017 chip. Then, by reading the pins on the mc23017, this gave the scan code as a variable, keyboard_input_number, for the key that was pressed.

```
for (int i=0;i<8;i++)
{
bitWrite (keyboard_input_number,i,(mcp2.digitalRead (i)));
}
```

Then, I wrote a lookup table,

```
ps_2_to_ascii();
```

where I converted the scan code to an equivalent ascii code. If the shift button had been pressed, then I changed the ascii code to its shifted equivalent, via another lookup table

```
ps_2_to_ascii_shifted();
```

The TTL output pins on the keyboard module will indicate if a key is pressed, so I read that on pin 8 on the mcp chip. This is reversed from the button switches used elsewhere, as it reads 1 if a key has been pressed. It's slightly limited in use, and the keyboard reader could be expanded upon a lot more, but for the purposes of just typing in a preset name from time to time it was unnecessary to make it fully functional. The preset save function will take characters from 3 sources - either the up/down buttons, encoder 7, (the output gain encoder), or the keyboard, so the keyboard isn't really needed. However, part of the point of this project was for me to learn, so I thought that this was worthwhile. An advantage of reading the keyboard in this way, though, is that it can be plugged/unplugged at any time, and can be viewed as essentially as 104 uniquely identifiable buttons, although the obvious limitation is that multiple keys at the same time only register the last key pressed. As I mentioned above, the module has 5V printed on the board, but I needed to feed it over 6V for it to work.

EEPROM

Next is the eeprom chip. This is easy enough to use, with only 4 wires - 2 to the I2C bus (pins 18 and 19 on the teensy), and VCC and GND.

This uses an eeprom library

```
#include <extEEPROM.h>
```

then define the eeprom with

```
extEEPROM eeprom1(kbits_256, 1, 64,0x50);//device size,number of devices, page size, bus
                                          address - the 0x50 is the default
```

and initialise in the setup() with

```
eeprom1.begin();//initialise the eeprom
```

I Only use very basic funtions in this sketch, with no error checking. To read a byte of information from a location, use

```
variable=eeprom1.read(address);
```

and to write to a location, use

```
eeprom1.write (address, value);
```

256 kbits is 32 kb - so, the address range is from 0 to 32768. I made the presets a fixed size of 64 bytes, comprising up of 32 bytes of controller information, and 32 bytes for the preset name. This 64 bytes isn't used fully, but it makes saving and loading simpler by just using an offset - so, preset 1 will start at 64, preset 2 will start at 128 and so on.

The preset structure I used was a simple array, too:

```
preset[0]=left_in_out_1_state;
preset[1]=left_in_out_2_state;
preset[2]=left_in_out_3_state;
preset[3]=left_cut_boost_1;
preset[4]=left_cut_boost_2;
preset[5]=left_cut_boost_3;
preset[6]=left_frequency_1;
preset[7]=left_frequency_2;
preset[8]=left_frequency_3;

preset[9]=right_in_out_1_state;
preset[10]=right_in_out_2_state;
preset[11]=right_in_out_3_state;
preset[12]=right_cut_boost_1;
preset[13]=right_cut_boost_2;
preset[14]=right_cut_boost_3;
preset[15]=right_frequency_1;
preset[16]=right_frequency_2;
preset[17]=right_frequency_3;

preset[18]=stereo_in_out_1_state;
preset[19]=stereo_in_out_2_state;
preset[20]=stereo_in_out_3_state;
preset[21]=stereo_cut_boost_1;
preset[22]=stereo_cut_boost_2;
preset[23]=stereo_cut_boost_3;
preset[24]=stereo_frequency_1;
preset[25]=stereo_frequency_2;
preset[26]=stereo_frequency_3;

preset[27]=output_gain;
preset[28]=channel_selected;
```

To initialise the eeprom, I copied the preset names, made them all the same length and made an array, here called preset_name_list, which I then loaded onto the eeprom with a short sketch, and a random data array, preset_2_values. I overwrote the blank data with actual data once the controller was up and running, as the save routines start with the preset name intact for an overwrite.

Short version:

```
String preset_name_list[61];

char p0[27]={“Preset                    “};
char p1[27]={“B - DI needs love 1       “};
char p2[27]={“B - DI needs love 2       “};
char p3[27]={“B - DI needs love 3       “};

etc



int c;
int b;
byte preset_2_values[29]={1,1,1,64,64,64,5,5,5,1,1,1,64,64,64,5,5,5,1,1,1,64,64,64,5,5,5,64,1};


for (int a=0;a<61;a++)//quickie to put the list of preset names into the eeprom
{
  b=(a*64)+32;
  c=(a*64);
 String preset_2_name=preset_name_list[a];

   for (int k=0; k<32; k++)//
  {
    eeprom1.write (c+k,preset_2_values[k]);
    delay(10);
    eeprom1.write (b+k,preset_2_name[k]);
    delay(10);

  }
  Serial.print (“Preset written: “);
Serial.println (a);
}
```

Otherwise, the value of any address on a blank eeprom is 255, which messes up the fairly tight graphics space on the LCD when read. Not the end of the world, but inelegant :)

I used preset 0 for an initial default start of all values flat, and have it so that the controller can respond to program change messages 1-128 rather than 0-127.

MIDI shield

Again, very straightforward to use. It's an arduino shield, but only need to wire up the tx/rx pins to pins 0 and 1 on the teensy, and feed it 5v and gnd. As I mentioned above, don't forget about the on/off switch, on to use MIDI, off to be able to program via arduino environment. I used the standard MIDI library in parallel with the usbMIDI functions of the teensy - so, everytime there's a usbMIDI message, this is duplicated with a MIDI message. I used the same setHandles, OnControlChange and OnProgramChange , for both, so if either MIDI or usbMIDI is received, it has the same action. Also don't forget that it's a shield, so be careful to wire the rx/tx the right way round.

Limitations and Considerations

What its trying to achieve is complete hardware control of a software plugin. One issue that I found a little frustrating is that its not possible to assign a midi controller to the stereo/LR selection in the plug in. The simplest way round this is for the plug in to be in LR all the time, and use the hardware controller to assign identical values for right and left if stereo mode selected on the hardware. I did code in stereo cc's to begin with, and left them a little undeveloped in the sketch, as dual mono works well. Their main reason for inclusion at all is for copying presets - if a stereo preset is selected in the plugin, it will send all the stereo values to the hardware. If the hardware is then put into L or R, the preset can be saved in the hardware as dual mono as a starting point.

Again in the plug-in, if changing from stereo to L/R, the values jumped to previous L/R values. This was a little jarring, so in this hardware controller this will just disconnect the link betwen the channels and leave them at their current values.

Also in the plug-in, if changing from L/R to stereo, the plugin takes the left values and assigns to the right. In this hardware controller, I changed this to whichever side was being previously controlled taking priority - so if currently on left controls, then right values become same as left, and if on right controls, left controls become same as right.

I think I tied myself in a few logic knots trying to square up the stereo/left/right configurations, especially for the transitions between them, but don't *think* I missed anything....

The MIDI CC's that I used are commented in a list in the OnControlChange. I tried to pick CC's that were unassigned or general purpose, and used MIDI channel 16, but obviously these can all be changed to suit. In Cubase, it's set up as a generic remote, and taken out of AllMidiIn, to avoid un-planned for conflicts.

Basic Structure

Conceptually quite a simple series of ideas, really.

```
    initialise all values
        update the screen

    Check for changes in channel to control
        if changed L/R/S
            put all appropriate values into variables
            assign encoders and buttons to selected channel
            update the screen

    Check for any external MIDI input
        If data
            put all appropriate values into variables
                update the encoders to reflect new values
                update the screen

    Check for any changes from encoders and buttons
            put all appropriate values into variables
                send MIDI data
`               update the screen

    If loading a preset
            save current variables to temporary location , in case of cancelled loading
            use buttons to select
                load data from eeprom
                send MIDI data
```

```
                    update the screen

          if cancelled loading

                    reload saved temporary variables

                    update the screen


If saving a preset

          save current name to temporary location, in case of cancelled saving

          select preset slot to save to

                    give it a name

                    write all current variables to eeprom
```

I tried to break up the sketch into smaller parts, and actually ended up with a fair number of functions:

assign_data_to _encoders

when channel changed, puts the variables for the selected channel into the encoders

button_enc_updates

when channel changed, updates the screen to show selected channel

channel_updates

when channel changed, re-assigns variables and controllers depending on what the channel is changed from or to.

in_out_values

when in/hicut/locut pressed, updates the LCD and turns LED's on/off depending on channel

init_display

updates all variables on the LCD. There is some duplication of function here, as each individual variable also has an update function, but I found it useful for self-clarity to have a seperate function to do everything in one go, mainly after loading a preset.

keyboard_input_reader

reads the scan code from the ps/2 interface via an MCP23017 IC, and then calls

ps2_to_ascii

to change scan code to ascii code, checks for a shift, and if shift key has been pressed

ps2_to_ascii_shifted

left_cut_boost_value_updates

3 of these, 1 for each of the 3 values. Updates the graphics on the LCD to reflect new values.

left_frequency_value_updates

3 of these, 1 for each of the 3 values. Uipdates the graphics on the LCD to reflect new values.

right_cut_boost_value_updates

right_frequency_value_updates

stereo_cut_boost_value_updates

stereo_frequency_value_updates

As above, 3 of these for each, all to update the graphics on the LCD to reflect new values.

output_gain_value_update

just one of these, updates the graphics on the LCD to reflect new value. Output gain is independent of channel, applies to final output.

These individual updates were used instead of using the init_display for speed - a full screen update over the SPI take around 1/4 second, so just updating what has changed is much faster.

load_button_pressed

saves current values, reads buttons for preset selection, reads eeprom and assigns variables for selected preset, updates LCD to reflect new values, calls send_all_controller_data to send MIDI data out. Also sends out a MIDI program change for the selected preset.

save_button_pressed

slightly more complex part of the code, as this allows selection for preset location to save to, then a somewhat basic ability to write a name for it. This was slightly more involved, to allow for character selection via buttons, encoder or ps/2 keyboard.

on_control_change
on_program_change

if control change MIDI data in, will update the variables to reflect new value, then update the LCD.
If program change, will load the preset depending on patch number, then update LCD.

power_symbols

updates the LCD screen with power symbols to reflect the in/out states for L/S/R.

read_buttons

reads the button states using reads of the MCP23017 chips, assigns values, updates LCD / LEDs, and sends relevant MIDI data.

read_encoders

reads the encoders, assigns values depending in which channel is being controlled, updates LCD and sends MIDI data.

reset_buttons

makes the variables for all button presses equal to the last read position, so that next time around the loop can compare values to look for a change. Just seemed a little simpler to put them all in one place.

stereo_to_right_left

If channel changed, assigns variable values of stereo channel to both right and left channel.

valueboxes

graphics background on the LCD, basically draws rectangles.

Obviously this isn't a complete breakdown of the whole code, but hopefully gives an idea of the structure. I tried to document everything in the sketch, and all the voids are spread out across tabs, to make looking at any particular section a little easier. I doubt that anyone would be that interested in making an exact clone of this, but if you're curious about making DIY MIDI controllers, I'd be delighted if you've found this at all interesting. This is the first project that I've put online, at a level that I'd guess is at a step up from raw beginner, and I'm fairly pleased with it.

I apologise if any of my use of terminology is incorrect, I can only reitereate that this is all self-taught, mainly through frustration, experimentation and frying more components than I care to discuss :) My wiring is all either hand-soldered or on breadboard, so really does look like a rats nest - without some patience, a magnifying glass and needle-nosed pliers I'd be totally lost. My coding is what I've gleaned from reading around tutorials, moments of understanding from snippets that others have put online, and some experimental guesswork, so is probably far from elegant. Without the ground work that so many others have put in to make the environment, libraries and forums feasible, I could never have come close to doing this kind of thing. So, my thanks, for all your work and inspiration.

As far as I am aware, everything I've used in this project is open source. I'll just repeat what seems to be a common theme, then - please feel free to use anything, in total or part, that you've found here, if its for non-commercial use. It's not a holy object, and it could certainly be improved upon. If you really like what I've done in this description/breakdown, then you can maybe buy me a cup of coffee/beer, or a cigar, if you're really flush :) I'm not selling anything, I can't offer support for software that I'm learning about myself, and I certainly don't want to take credit for the work of other people. This is my first sizeable project, so I'm testing the water, really, to gauge interest in the kind of thing I'm trying to do, and in the usefulness of this kind of write-up - feedback would be most appreciated.

paypal.me/PaulHeskethUK

Things I've found Useful

I have to give thanks to Paul Stoffregen, and you can't really be using a teensy without seriously looking at

https://www.pjrc.com/

The ili9341_due library is available on github, the excellent work by marekburiak.

http://marekburiak.github.io/ILI9341_due/

The MCP23017 library is by adafruit, again on github

https://github.com/adafruit/Adafruit-MCP23017-Arduino-Library

The eeprom library is also on github, and I fully appreciate that I've barely touched the surface of it

https://github.com/JChristensen/extEEPROM

I think all the other libraries were installed with the arduino environment/teensyduino.

If I've missed anyone, then please accept my apologies, and let me know.

I don't think there's much point in a page of links to tutorials, let me google that for ya.....

Components have beeen bought from numerous suppliers, but mainly on ebay worldwide and aliexpress. The example photos of components earlier were stock photos, not taken by me, but taken from where I bought them from. I've been very impressed with Breeze Audio on aliexpress, their selection of cases and helpfulness have been marvellous, and I will definitely use them again. Another site that I've used in the past for cases has been DIYerzone, which was good, and I then promptly botched the drilling of it. Bugger. As a general thing, I've looked at components/breakout boards etc from the US, but shipping/customs can be on the very high to absurd side. China is generally more available, if a little slow to deliver sometimes. When buying from Aliexpress, I try and keep each order under the VAT threshold of £15, to avoid import charges. Obviously that can't be the case for an enclosure, though, so have to bite the bullet with that. Enclosures have been a large consideration in what I want to do - I'm a very tactile person, so want to have controls that feel positive and solid.

Guess that's about it, if you've read to here then I'm flattered - take care.

Paul Hesketh.