

DROID

Universal CV Processor

User manual

for firmware version blue-3

August 18, 2023



Contents		
1 Installation	4	5.8 The P8S8 controller 41 5.9 The B32 controller 42 5.10 The M4 motor fader controller 43
2 Creating DROID patches	5	6 The G8 expander 47 6.1 Introduction 47 6.2 Installation 47 6.3 Using the G8 in patches 48
2.1 Getting started	5	7 The X7 expander 49 7.1 Quick start 49 7.2 General overview 49 7.3 Installation 50 7.4 USB access to your SD card 50 7.5 MIDI 51 7.6 MIDI through 54 7.7 Four gate outputs 55 7.8 Eight multi color LEDs 55 7.9 Fast patch upload via Sysex 55 7.10 Software update for the X7 57 7.11 Some technical details 58
2.2 Working with the Forge	9	8 The R2M/R2C controller bridge 59 8.1 Introduction 59 8.2 Setup with one master 59 8.3 X7 connected to the master 59 8.4 X7 in the skiff 60 8.5 Controllers before the R2M/C bridge 60 8.6 More than one bridge 60 8.7 Setup with two masters 60
2.3 Using the master's inputs and outputs	10	9 Droid under the hood 61 9.1 How the module's state is saved 61 9.2 The order of the circuits 62 9.3 Displaying the value of a register 62 9.4 Displaying current values 65 9.5 Controller latency 66
2.4 Numbers and voltages	11	10 Firmware upgrade 67 10.1 What version do you have? 67
2.5 Multiply and add, attenuation and offset	12	
2.6 Internal connections	12	
2.7 Controllers	13	
3 Advanced patching concepts	15	
3.1 One knob - multiple functions	15	
3.2 Presets	19	
3.3 Tap tempo	21	
4 Creating DROID patches with a text editor	22	
4.1 General procedure	22	
4.2 Basic structure of the patch file	23	
4.3 Finding a problem in your DROID patch	23	
4.4 Inputs, outputs and other registers	26	
4.5 Specifying numbers in your patch	28	
4.6 Attenuating and offsetting inputs	28	
4.7 Internal patch cables	30	
4.8 Using outputs as inputs	30	
4.9 Using inputs as outputs	30	
4.10 Parameter arrays	31	
4.11 Comments & spaces	31	
4.12 More than one patch on the memory card	31	
5 Controllers	32	
5.1 Installing the controllers	32	
5.2 How to use controllers in your patch	33	
5.3 Troubleshooting	35	
5.4 The P2B8 controller	37	
5.5 The P4B2 controller	38	
5.6 The P10 controller	39	
5.7 The S10 controller	40	

10.2	Normal update procedure	67
10.3	Upgrade from green to blue	69
11	Calibration, factory reset and other maintenance stuff	70
11.1	The maintenance mode	70
11.2	Factory reset	71
11.3	Calibration of the outputs	71
11.4	Using your own SD card	72
12	Hardware	73
13	Reference of all circuits	75
13.1	adc - AD Converter with 12 bits	78
13.2	algoquencer - Algorithmic sequencer	80
13.3	arpeggio - Arpeggiator - pattern based melody generator	92
13.4	beroulli - Random gate distributor	100
13.5	burst - Generate burst of pulses	101
13.6	button - Does all sorts of useful things with buttons	103
13.7	butongroup - Connected buttons	108
13.8	calibrator - VCO Calibrator	111
13.9	chord - Chord generator	115
13.10	clocktool - Clock divider / multiplier / shifter	121
13.11	compare - Compare two values	124
13.12	contour - Contour generator	126
13.13	copy - Copy a signal	131
13.14	crossfader - Morph between 8 inputs	132
13.15	cvlooper - Clocked CV looper	133
13.16	dac - DA Converter with 12 bits	136
13.17	delay - A tape delay for CVs, gates and numbers	138
13.18	droid - General DROID controls	141
13.19	euklid - Euclidean rhythm generator	143
13.20	explin - Exponential to linear converter	145
13.21	faderbank - Create multiple virtual faders in M4 controller	147
13.22	fadermatrix - Matrix of up to 4x4 virtual motor faders	149
13.23	firefacecontrol - Control a RME Fireface interface (experimental)	154
13.24	flipflop - Simple flip flop	156
13.25	fold - CV folder - keep (pitch) CV within certain bounds	157
13.26	fourstatebutton - Button switching through 4 states (OBSOLETE)	159
13.27	gatetool - Operate on triggers and gates, modify gatelength	160
13.28	lfo - Low frequency oscillator (LFO)	163
13.29	logic - Logic operations utility	169
13.30	math - Math utility circuit	172
13.31	matrixmixer - Matrix mixer for CVs	174
13.32	midifileplayer - MIDI file player	177
13.33	midiin - MIDI to CV converter	183
13.34	midiout - CV to MIDI converter	190
13.35	midithrough - MIDI routing through X7	199
13.36	minifonion - Musical quantizer	200
13.37	mixer - CV mixer	204
13.38	motoquencer - Motor fader sequencer	205
13.39	motorfader - Create virtual fader in M4 controller	226
13.40	notchedpot - Helper circuit for pots (OBSOLETE)	230
13.41	notebuttons - Note Selection Buttons	231
13.42	nudge - Modify a value in steps using two buttons	234
13.43	octave - Multi-VCO octave animator	237
13.44	once - Output one trigger after the Droid has started	239
13.45	polytool - Change number of voices in polyphonic setups	240
13.46	pot - Helper circuit for pots	242
13.47	quantizer - Non-musical quantizer	248
13.48	queue - Clocked CV shift register	249
13.49	random - Random number generator	250
13.50	recorder - Record and playback CVs und gates	251
13.51	sample - Sample & Hold Circuit	257
13.52	select - Copy a signal if selected	258
13.53	sequencer - Simple eight step sequencer	259
13.54	slew - Slew limiter	263
13.55	spring - Physical spring simulation	265
13.56	superjust - Perfect intonation of up to eight voices	267
13.57	switch - Adressable/clockable switch	269
13.58	switchedpot - Overlay pot with multiple functions (OBSOLETE)	271
13.59	timing - Shuffle/swing and complex timing generator	273
13.60	togglebutton - Create on/off buttons (OBSOLETE)	275
13.61	transient - Transient generator	277
13.62	triggerdelay - Trigger Delay with multi tap and optional clocking	279

1 Installation

Controller connector

The connector for the controllers has 6 pins (two rows of three pins) and is used for connecting a chain of **B32**, **P2B8**, **P4B2**, **B32**, **P10**, **P8S8** and **M4**. Also the **X7** is connected here. An X7 must always be the first in the chain.

Programming port

The 6 pin programming port is not mounted in a box. **Caution: Do not connect anything to this port!** It is solely for the initial programming in our labs. Later firmware upgrades are done via the Micro SD card.

Power connector

The power connector has 10 pins (two rows of five pins). Use the shipped 10 pin ribbon cable in order to connect it with the bus board of your Eurorack case. **Important: Put the red stripe down!**



Do not mix up the connectors! This will destroy your electronics. Do not force in cables in the wrong orientation or with the wrong number of pins! Do not attach anything to the programming port.

Expansion port for G8 expanders

The connector for the G8 expanders has 8 pins (two rows of four pins). Here you can add up to four G8 expanders for an additional 8 - 32 gate inputs/outputs. Please refer to page [47](#) for details.

2 Creating DROID patches

2.1 Getting started

The **DROID** is a very flexible universal processor for control voltages (CV) in a Eurorack modular system. It can do almost any CV task you can imagine, such as sequencing, melody generation, slew limiting, quantizing, switching, mixing, working on clocks and triggers, creating envelopes and LFOs or other fancy voltages, or any combination of these at the same time! While doing this, it is very precise both in voltage and in timing.

To bring your **DROID** to life, you create a *Droid patch* and load it to your master.

What is a Droid patch? Well, the **DROID** is like a self contained modular system for CV in a module. In order to avoid confusion with “real” modules – the building blocks in a Droid patch are called *circuits*. There are very simple circuits like a mixer for CVs. And there are also very complex circuits like an sophisticated algorithmic trigger

sequencer called **algoquencer** (see page 80).

Much like real modules, the circuits have input and output jacks. These are called inputs or outputs, or sometimes also “parameters”. Each of them can be set to a fixed value, wired to one of **DROID**’s physical inputs or outputs, set by a knob or button on a Droid controller or internally wired to other circuits in order to create more complex patches.

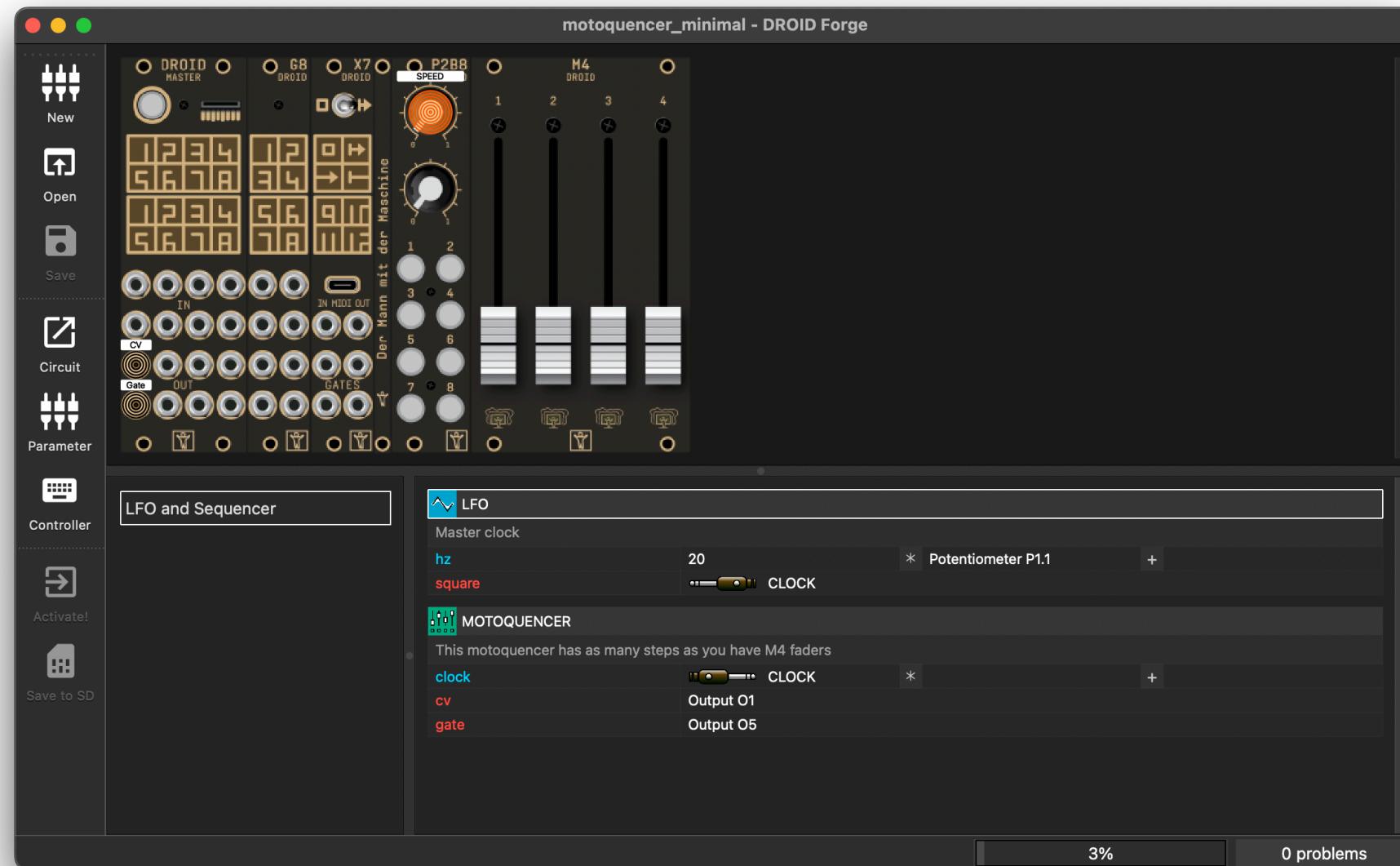
A Droid patch lists all the circuits you want to use and describes how they are connected and how the parameters are set.

Technically, a patch is a small text file with the name **droid.ini**, which is located on the micro SD card in the SD slot of the master. You can create and modify this file with any text editor you like, and the chapter *Writing*

Droid patches with a text editor goes in all length through the structure of that file (see page 22).

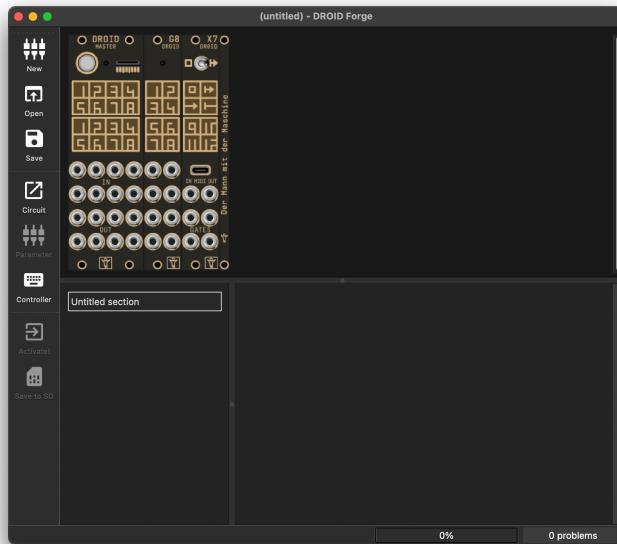
However, starting in November 2022 there is a new application for Mac and Windows called the *Droid Forge* - or simply the *Forge*. That’s the new graphical tool for creating patches and makes working with the Droid super easy. The Forge is available for free download for on <https://shop.dermannmitdermaschine.de/pages/downloads>.

Working with the Forge is highly recommended. However, in this manual you will find lots of examples that refer to the *text representation* in **droid.ini**, because it’s much easier for showing just small portions of a patch than a full sized screen shot of the Forge. And it is straight forward to recreate these examples in the Forge.



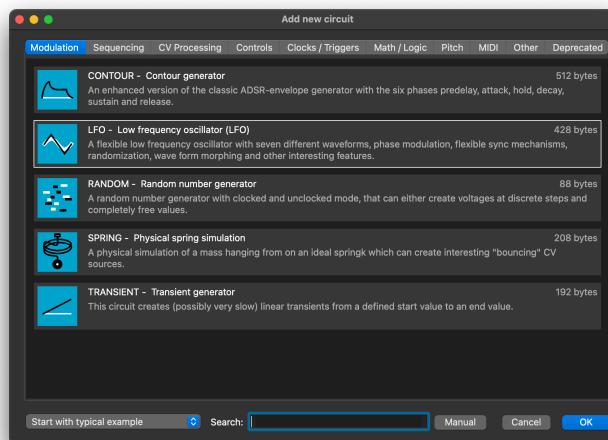
A first patch example - step by step

So let's start! First install the Droid Forge. Download it from the upper link and install it to your Windows PC or Mac. After starting it you get a window like in the screenshot above. The Window is divided into three areas:

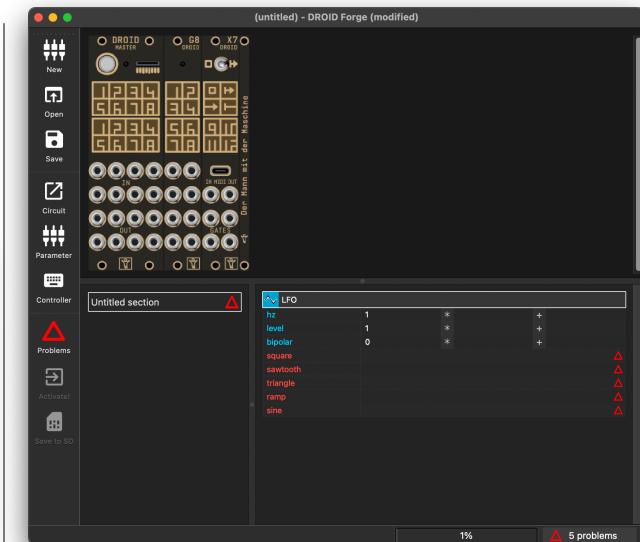


- At the top there is the *rack view*, where you see the Droid modules that you are working with
- At the bottom right is the *patch view*, where you see the circuits and their parameters
- At the bottom left is the list of *sections*. They are for dividing your patch into sections and make it easier to read.

Now let's create a first simple patch. From the *Edit* menu choose *New circuit...*. This opens a dialog for adding a circuit to your patch:



Select the LFO circuit and click *OK*. This adds an LFO to your patch. Because the setting at the bottom left is set to *Start with typical example*, your LFO will already have a couple of inputs and outputs defined:



Input are written in blue, outputs in red. You learn about all available parameters of a circuit in its chapter here in this manual. Have a look at the LFO circuit on page [163](#). For example:

- hz** sets the speed of the LFO in cycles per second.
- level** defines the maximum voltage level of the output
- bipolar** changes the range from 0 V ... 10 V to -10 V ... 10 V, if set to 1.

The outputs provide various wave forms of the LFO.

If you want to add more inputs or outputs, choose *New parameter...* from the *Edit* menu or press the icon *Parameter* in the toolbar. And of course every action in the Forge has a keyboard shortcut, in this case ⌘ N (or Ctrl N on Windows).

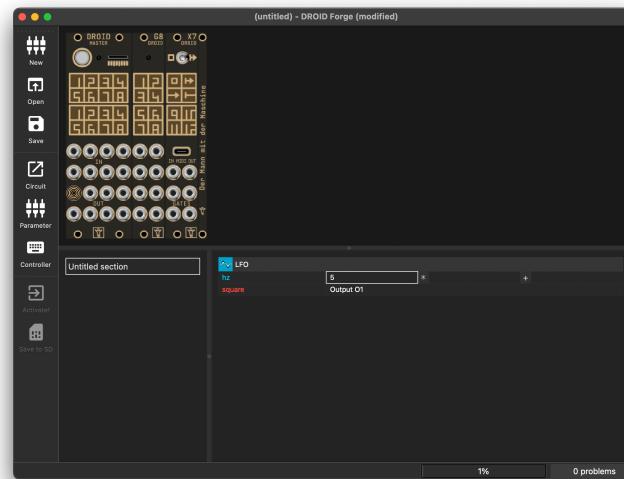
Now move the cursor to the row **square**, either with the cursor keys or by clicking with the mouse. Move the cursor to the second column.

In the rack view, click on the Droid master on the first jack in the third row of jacks. That jack is called "Output 1" or simply **O1**. This inserts *Output O1* as a value for the **square** parameter. The LFO will now send a square wave to output 1 of the Droid master.

Move the cursor to the second column of the parameter **hz** and type **5** and hit the enter key.

Move the cursor to the *first* column of all other parameters and delete those rows by hitting the backspace key so that you just have two lines left. We don't need these parameters for now.

This is how it should look like when your are finished:



Your first patch is ready!

There are two ways to load the patch to your master. The first is by manually swapping the SD card:

- Pull the memory card from your master and put it into a card reader in your Mac / PC. After a couple of seconds the toolbar icon *Save to SD* becomes active.
- Press that icon to copy your patch to the SD card. It will automatically be ejected afterwards.
- Put the SD card back to your master and press the master's button. That loads the patch and the LED for output 1 will start flashing in 5 Hz (five times a second).

The second way to deploy a patch is much more convenient, but needs an attached **DROID X7** expander (see page [49](#) for more details on the X7). With the X7 you can deploy the patch via MIDI sysex:

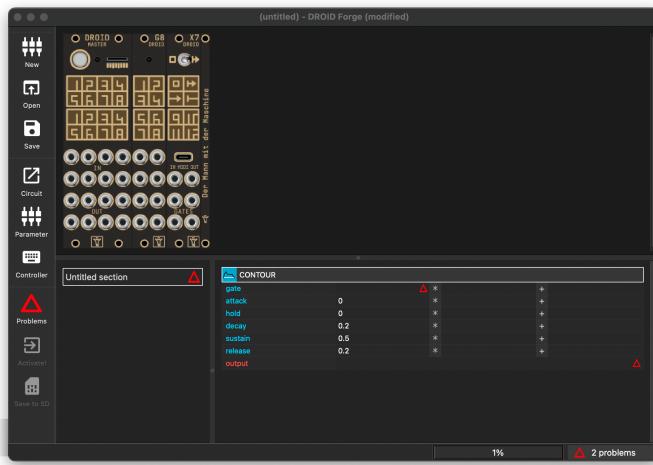
- Wire the X7 with the shipped USB-C to classic USB cable to your Mac / PC.
- Set the switch on the X7 to the *right*. After a short delay the *Activate!* icon in the Forge become active.
- Click *Activate!*. Your patch will immediately be loaded and become active.

2.2 Working with the Forge

Before we have a deeper look at how Droid patches work, let's first have a closer look at the Forge.

Problems

Your patch can have *problems*. These are inconsistencies that would confuse your DROID, if you load it. One example is a parameter line without a value. In order to avoid such trouble, the Forge does not let you load a patch while it has problems.



As you see from the screenshot, there is a red triangle in the toolbar and also a note in the statusbar telling you that there are two problems. If you click on either of them, your cursor will jump to the next unsolved problem. Fix these and you will be able to load the patch.

When loading a patch does not work

As we have seen in the first section, the two toolbar icons for loading a patch are only active, when that is possible. If you encounter problems with *Save to SD*, please check:

- Make sure your micro SD card in the card reader of your computer.
- Make sure it is an SD card that already has been used in the Droid. New and empty cards will not be accepted.
- If unsure, check with your Finder or Explorer, if the card is really accessible.

In case of a problem with *Activate!*, check the following:

- This button only works if you have an X7 expander attached to your master.
- Check the correct wiring of the X7.
- The switch of the X7 must be in the right position.
- The X7 must be connected with a USB cable to your Computer.
- USB-C to USB-C do not work! Use the cable shipped

with the X7 or a similar one.

- If the icon still does not get active, try putting the X7 switch to the middle position and after a small pause right again.

Working with sections

In the bottom left of the Forge you see a pane with the entry *Untitled section*. Sections are a good way to organize more complex patches. Each section contains a list of circuits - and thus a part of your patch. You can move around sections with drag & drop. You can duplicate, rename and delete them and do many other practical things.

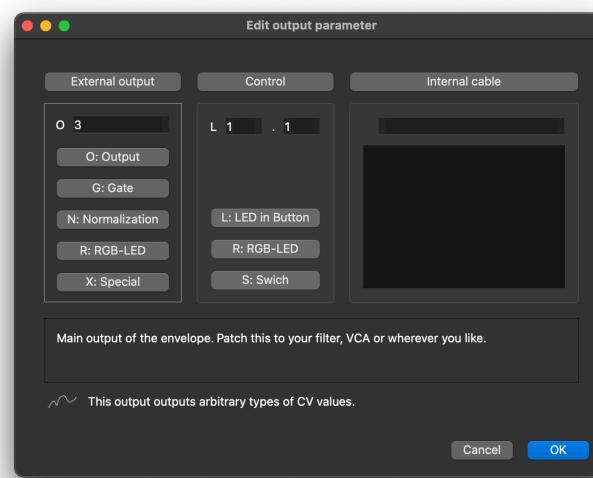
2.3 Using the master's inputs and outputs

Inputs and outputs

Your master has eight CV inputs and eight CV outputs, both ranging from -10 V to +10 V. The inputs are abbreviated with **I1**, **I2**, ..., **I8**, the outputs with **01**, **02**, ..., **08**. These jacks allow your Droid patch to communicate with the outside world. The abbreviations **01** and so on are also called *registers*.

To use an output, you need to connect an output parameter of a circuit to it. There are several ways to do this:

- Click on the output jack in the image of the master while the cursor is right next to an output parameter.
- Type the output's name while the cursor is at that position, e.g. **03**.
- Press *enter* while the cursor is next to an output. That opens a dialog where you can see all options.



For inputs it's much the same. Move the cursor into the second column, right next to the input name, and assign one of the inputs.

Input normalization

Eurorack modules know the concept of *input normalization*. This means that an input gets some default signal when nothing is patched in the jack. The DROID supports this by offering the registers **N1** ... **N8**. These behave like *outputs* that are internally connected to the normalizations of the input jacks.

When circuit send an output signal to **N1**, this signal is seen by input **I1**, as long as nothing is patched into that input. This allows you to create more flexible patches. You might for example have an internal clock in your patch (created with an LFO circuit) that can be overridden by patching something into **I1**.

To do that, send your internal LFO clock signal to **N1**. Then let the rest of the patch use **I1** as clock input.

Using the G8 gates expander

You can connect up to four G8 expanders to your master. Each G8 gives you eight additional gate inputs or outputs. Each jack of the G8 can be used as an input or output, depending on how you use it in your patch.

In the Forge there is one G8 being displayed in your rack view per default. If you don't have a G8 or you have more than one, you can fix that in the **View** menu. When the current patch actively uses any of the G8 jacks, the needed G8s are always being displayed. Use your G8 ether by clicking on one of the jacks in its image, or press *Enter* for a guided dialog and select *G: Gate*, or simply type e.g. **G2.7** for gate 7 on the second G8 expander.

Note: The G8 cannot output continuous CV values. When used as output it either sends 0 V or 5 V. And inputs see a high signal at a voltage about 0.75 V.

Please refer to page [47](#) for more details on the G8.

2.4 Numbers and voltages

How voltages are converted

DROID is a CV processor that inputs and outputs control voltages. But internally it works with just numbers, because this is much more convenient. Here is how the **DROID** operates:

1. When reading voltages from the *input jacks*, these are converted from the range -10 V to +10 V into the number range from -1 to +1.
2. All circuits operate on these numbers.
3. When sending numbers to the *output jacks*, the numbers are converted back from -1 to +1 to the voltage range -10 V to +10 V.

This means that if the **DROID** reads a voltage of 2.5 V at one of its inputs, in the **DROID** patch this will appear as **0.25**. Or if you send a value of **0.5** to one of the outputs, it will output exactly 5.0 V. This is in fact very convenient as you will see.

In your patch you can either write **2.5V** or **0.25**. Both mean the same. It's up to you which of both you prefer.

Voltages out of range

The **DROID**'s hardware cannot work with voltages beyond ± 10 V. This is no limitation, since Eurorack has a maximum voltage range of ± 12 V and barely any module reaches even 10 V at its output. Many digital modules are even limited to the range 0 V...5 V.

That means that any voltage out of that range appearing at an input is simply truncated. Send -10.8 V at an input and **DROID** will see it as -10 V. Or send the number 1.1 to an output (which would be 11 V) and it will output 10 V nevertheless.

But: internally - in your **DROID** patch - numbers can get arbitrarily low or high. So in intermediate steps it's absolutely no problem to work with larger numbers. Some cir-

cuits even require such numbers. E.g. in the **minifonion** (see page 200) you specify the root note B by saying **root = 11**. On the side of the jacks that would mean 110 V, but that's not relevant here.

For those of you wanting to dig more into the details of number processing: **DROID** works internally with 32 bit floating point values. The exponent is 8 bits. The largest number is slightly above 30000000000000000000000000000000 (a 3 with 38 zeroes).

The smallest number greater than zero is approximately 0.000000000000000000000000000000011 (that's 37 zeroes after the decimal point). The negative range is similar.

One word about the G8 expander: its outputs can only output two possible voltages: 0 V and 5 V. The rule is: any number ≥ 0.1 sent to one of its **G** registers will set its output to 5 V, any other number to 0 V.

2.5 Multiply and add, attenuation and offset

As you might have noticed, input parameters of circuits have *three* columns where you can enter values, whereas outputs just have one. These three columns are:

A: Input value

B: multiplication / factor / attenuation

C: offset

So the value that's actually used by the input is $A \times B + C$. That's much like Eurorack modules that have an additional potentiometer for CV attenuation (hence multiplication) and/or offset.

The special thing about DROID is: Even the attenuation and the offset can themselves be CVs (come from external sources, other circuits, etc.). So essentially every input has a small VCA and mixer included.

2.6 Internal connections

One important concept for building more interesting patches is adding connections between circuits. These connections are called *internal cables*.

Consider the following example: You have one LFO circuit that outputs a square wave, which should be used as a clock signal. That clock shall trigger an envelope circuit (called **contour**).

Let's assume you want to create a cable from the **square** output of the LFO to the **gate** input of the envelope. To do this, move the cursor to the second column of the square output and press **=** (equals). This starts creating a cable. You will see an indicator in the statusbar.

Now move the cursor to the target of the cable: the pa-

rameter value of the gate input. Here press **=** again (or enter, if you like). This opens a small dialog for giving the cable a name. Choose a nice name that helps you understand what's going on later - for example **CLOCK**.

After hitting enter or pressing **OK**, you get a connection from the square output to the gate input. The envelope's output is wired to **01** in this example, so you get an envelope triggered at 8 Hz at output 1.

These are the rules for internal cables:

- Every cable must be connected to *exactly one* output.
- Every cable must be connected to *at least one* input.

That means that you can use a cable as a multiple and distribute signals to several circuits. But if a cable has no inputs or no or more than one output connects, it counts as a problem and you cannot load the patch.

Note: There are more ways to create patch cables:

- In a cell type an underscore followed by the name of the cable.
- In a cell press **enter** and choose a cable in the value dialog (or type a name for a new cable)
- Hold **Shift** while clicking into another cell (Windows: Alt key). That creates a cable between the two cells.

2.7 Controllers

Adding controllers

The fun part with DROID is attaching one or more controller modules to your master. When the project started, there was just the P2B8 controller available, which has two potentiometers - or short pots - and eight buttons. Hence the name! Now there are altogether six controllers that you can get for Droid. Learn more about the available controllers and how to connect them to the master on page [32](#).

In a nutshell, when wiring the controllers please check the following things:

- Check that the small green jumper on each controller is set to *Park* (or removed). Just on the last controller it must be at *Last*.
- The X7 must always be the first in the chain.
- The cable coming from the master must go to */N*, the cable to the next controllers is plugged into *OUT*.

Once your system is setup, it's very easy to use controllers in your patch. The first step is adding them to the rack view of the Forge. To do this double click on the background or choose *New controller* from one of the menus or use the icon *Controller* in the sidebar. The order of the controllers from left to right in the Forge must match the order of the wiring in your rack.

Notes:

- You can rearrange controllers with drag & drop. The patch will automatically be adapted so all references to the controls still work as expected. That's an easy way to adapt a foreign patch to your rack.

- When you remove a controller the Forge offers you to remap its controls to other existing controllers.
- The master, X7 and G8s cannot be moved.
- If you don't have or don't use the G8 or X7, you can hide it from the rack view. Check the *View* menu for that.

Using pots

The easiest way of using a potentiometer is by moving the cursor to a cell of an input parameter and then clicking on the pot in the rack view. This will insert something like **Potentiometer P1.2** in the cell.

Here **P1.2** is the register name for the pot and it means *controller one pot two*. If you aren't a mouse guy, you also can type **P1.2** if you like (omit the word *Potentiometer*, that will appear automatically). Or you press enter in a cell to get the value selector where you find the pots under *Controls*.

A pot always represents a value from 0.0 to 1.0 depending on the pot position. Often that range is not what you need, but with the help of the columns 2 and 3 (factor and offset) you can create any custom range. Consider using pot **P1.2** for setting and LFO speed between 1 and 10 Hz. This can be done by:

Column 1: **Potentiometer P1.2**

Column 2: **9**

Column 3: **1**

In the text representation this would be:

hz = P1.2 * 9 + 1

The math is easy: If the pot is totally at its left position, the register **P1.2** has the value 0.0. So $9 \times 0.0 = 0.0$ and thus adding one gives 1. At the right position the value of the pot is 1.0, so $9 \times 1 + 1 = 10$.

You can do much more complex things with potentiometers. For any of those please have a look at the circuit **pot** (see page [242](#)). For example you can:

- Overlay one pot with several independent functions by using **select**
- Save different values of a pot into up to 16 presets
- Create a virtual center notch, to make it easy to select the middle position *exactly*.
- Have a pot output discrete numbers, for example **0, 1, ... 8**, to select preset numbers, pattern lengths and much more
- Apply a non-linear slope to the output value

If you don't need any of these, just use pot directly without the **pot** circuit. That keeps your patch simpler.

Hints:

- If you right-click on a pot, button or other control in the rack view, you get a context menu.
- You can rearrange assignments of controls with drag & drop in the rack view.
- Double clicking on a control allows you to label it.

Using buttons

A button outputs the value **1** while pressed or **0** otherwise. Its register abbreviation is **B**, so **B3.4** is the button four on controller 3. You assign them just like pots.

The main difference is that buttons contains an LED. So if you want to make use of that, you need to *output* a value to the LED.

The button LEDs have their own registers, named **L**. So the LED in button **B3.4** is called **L3.4**. If you send a **0.0** to an LED, it will be dark. A **1.0** will make it shine at full brightness. Anything inbetween selects some intermediate brightness.

Sounds complicated, but at the end it makes sense, as you will see. And it also gives you flexibility.

Most times you don't like to *hold* the button all the time to make it do its work. You want it to *switch* between on and off with each press. This is done with the circuit **button** (see page [103](#)). And that also helps you to deal with the LED.

The following example is in Droid source syntax, but it is straight forward to setup this in the Forge. Add the circuit **Button** and the two parameter lines **button** and **led**:

```
[button]
  button = B1.1
  led = L1.1
```

Now each press at button 1 on controller 1 will *toggle* the button. **led** is an output parameter so the LED register **L1.1** will hold the current state of the button - either **0** or **1**.

You can use that as an input to some other circuit, for example for switching on and off an LFO by setting its level to 0 or 1:

```
[button]
  button = B1.1
  led = L1.1
```

```
[lfo]
  hz = 3
  level = L1.1
  sine = 01
```

There are many more ways for using buttons. Please look at page [103](#) for more examples. And also look at the circuit **buttongroup** (see page [108](#)). It can group several buttons together in a convenient way.

Hint:

- If in a circuit the LED definitions do not match the buttons, a light bulb icon will appear in the circuit

header. Click that to make the LEDs automatically match the buttons.

Switches

The S10 controller has ten switches. They have the register abbreviation **S**. The first two switches have eight positions and output the discrete numbers **0, 1, ... 7**. The small switches just have three positions: **0, 1** and **2**.

You can either use these switches directly in your patch or might want to try the circuit **switch** (see page [269](#)), for assigning something for every switch position. Create a circuit with one input for every position and just one output.

You get more details on the S10 on page [40](#).

Motor faders

The motorized faders from the M4 are always accessed via special circuits. Please refer to page [43](#) for all details about the M4.

3 Advanced patching concepts

3.1 One knob - multiple functions

Introduction

What I liked about modular synthesizers from the beginning was the principle known as “one knob one function”. In the 60’s that was certainly not yet a principle. It was the only way to build devices. Today buttons dedicated exclusively to a specific function have been almost completely rationalized away – whether it’s washing machines, cars or even doorbells of apartment blocks. Sure, the manufacturer saves money by simply installing one touchscreen instead of 50 real mechanical switches. The only thing that’s unfair is that we are being told that it’s progress that our car’s cockpit is so “clean” that we have to navigate to the third menu level to change the seat heating.

So “one knob one function” feels like pure luxury these days! And DROID is built for you to indulge in such a luxury. After all with 16 B32 controllers you can connect no less than 512 buttons to one master. So you can get quite far in reserving one button for one function.

The problem, however, (and I have to admit this at this point) is that it is a luxury. If you spend some time in creating cool DROID patches, new ideas pop up like mushrooms and in no time all pots and buttons are occupied. And not everyone has the money, the time and the patience, to order new controllers all the time.

That’s why DROID has a sophisticated system of overlaying your controls with almost as many functions as you want and switch between them, similar to menus or modes.

Overlaying pots

Let’s start with pots. Let’s assume that you have one P2B8 and want to use the upper pot to control both the attack and release of an envelope. The first step is to use the circuit **pot** (see page [242](#)). It is able to create a *virtual* pot from a real one. Let’s do this and start with controlling the attack:

```
[p2b8]
```

```
[pot]
    pot = P1.1
    output = _ATTACK

[contour]
    trigger = I1
    output = 01
    attack = _ATTACK
```

While this works, it has not really helped, yet. Still the pot has just one function. In order to map a second function on the same pot we need to do three things:

- Create a second **pot** circuit *for the same potentiometer*.
- Add a button for switching between these two functions.
- Use the **select** input in both **pot** circuits to choose which of the two functions should be active.

For our example we want to use the button **B1.1** to switch between controlling attack and release. For that we cre-

ate a **button** circuit, so that we can toggle the button. On should choose release and off attack.

We use the normal **output** of that circuit for selecting the release function. And the **inverted** output of the button is **0** when the button is active and **1** otherwise: just the opposite of **output**. We use that to select the other virtual pot – that for attack. Here is the complete patch:

```
[p2b8]
```

```
[button]
    button = B1.1
    led = L1.1
    output = _SELECT_ATTACK
    inverted = _SELECT_RELEASE
```

```
[pot]
```

```
    pot = P1.1
    select = _SELECT_ATTACK
    output = _ATTACK
```

```
[pot]
```

```
    pot = P1.1
    select = _SELECT_RELEASE
    output = _RELEASE
```

```
[contour]
```

```
    trigger = I1
    output = 01
    attack = _ATTACK
    release = _RELEASE
```

To summarize:

- For each virtual pot function that you need, create one **pot** circuit.
- Patch the outputs of these circuit to the inputs you want to control.
- Use the **select** inputs of the pots to decide which pot should be active.
- Make sure that at any time exactly one of the pot circuits is selected.

Note: As soon as you map several virtual functions to one pot, there is a difference between the *physical* position of the actual pot and the current virtual value. Nevertheless, turning the physical knob changes the virtual value. Please refer to **pot** (see page 242) for details.

Using button groups for selection

In the upper example we used a button for toggling between two states. If you want to have more than two function on a pot you need to choose a different method for selecting the “mode”. One is to use a **buttongroup** (see page 108), like the following one:

```
[buttongroup]
button1 = B1.1
button2 = B1.2
button3 = B1.3
button4 = B1.4
led1 = L1.1
led2 = L1.2
led3 = L1.3
led4 = L1.4
```

This group acts like “radio buttons”. If you press one of the four buttons, it is selected and the other three buttons are switched off. At any time, exactly one of the buttons is active.

Now we can use the **L1.1 ... L1.4** outputs of the button group for selecting four different pot functions:

```
[pot]
pot = P1.1
select = L1.1
output = _ATTACK
```

```
[pot]
pot = P1.1
select = L1.2
output = _DECAY
```

```
[pot]
pot = P1.1
select = L1.3
output = _SUSTAIN
```

```
[pot]
pot = P1.1
select = L1.4
output = _RELEASE
```

An alternative way is to use the **output** of the **buttongroup**. This outputs one of the numbers **0**, **1**, **2** and **3** depending on the selected button.

You can have the **pot** circuit get active on a specific number by using its **selectat** input in addition to **select**. If you use that, you can specify a value that **select** needs to have for the circuit to be selected (This also avoids a problem with the **led** outputs of the button group, which don't work if the button group *itself* uses select, as we will see later). Look:

```
[buttongroup]
button1 = B1.1
button2 = B1.2
button3 = B1.3
button4 = B1.4
```

```
led1 = L1.1
led2 = L1.2
led3 = L1.3
led4 = L1.4
output = _SELECT
```

```
[pot]
pot = P1.1
select = _SELECT
selectat = 0
output = _ATTACK
```

```
[pot]
pot = P1.1
select = _SELECT
selectat = 1
output = _DECAY
```

```
[pot]
pot = P1.1
select = _SELECT
selectat = 2
output = _RELEASE
```

```
[pot]
pot = P1.1
select = _SELECT
selectat = 3
output = _SUSTAIN
```

Here the first **pot** circuit is selected when **_SELECT** has the value **0**, and so on.

Selecting with switches

The S10 controller (see page 40) is perfect for selecting virtual functions. The two rotary switches have eight positions each and can directly be used for **select** in combination with **selectat**.

```

[s10]

[pot]
pot = P1.1
select = S1.1
selectat = 0
output = _ATTACK

[pot]
pot = P1.1
select = S1.1
selectat = 1
output = _DECAY

[pot]
pot = P1.1
select = S1.1
selectat = 2
output = _SUSTAIN

[pot]
pot = P1.1
select = S1.1
selectat = 3
output = _RELEASE

```

Note:

- In this example the switch positions 4 though 7 don't have any function.
- The small toggle switches of the S10 output 0, 1 or 2 and are useful for smaller selections.

Overlaying buttons

Just as pots, buttons can have multiple overlayed functions. This time you need to use the **select** input from the circuit that *controls* the buttons. The most obvious such circuit is **button**. But also **butongroup** and even

more complex circuits like **algoquencer** (see page 80), **matrixmixer** (see page 174) and **nudge** (see page 234).

Here is an incomplete sketch of a circuit that uses a butongroup with three buttons to select one of three instances of an algoquencer. That way the buttons **B1.1**, **B1.2** and **B1.3** choose between three "tracks" or "instruments":

```

[p2b8]
[b32]

[butongroup]
button1 = B1.1 # select track 1
button2 = B1.2 # select track 2
button3 = B1.3 # select track 3
led1 = L1.1
led2 = L1.2
led3 = L1.3
output = _TRACK

[algoquencer]
select = _TRACK
selectat = 0 # track 1
button1 = B2.1
button2 = B2.2
button3 = B2.3
button4 = B2.4
...
led1 = L2.1
led2 = L2.2
led3 = L2.3
led4 = L2.4
...
trigger = 01

[algoquencer]
select = _TRACK
selectat = 1 # track 2
button1 = B2.1
button2 = B2.2

```

```

button3 = B2.3
button4 = B2.4
...
led1 = L2.1
led2 = L2.2
led3 = L2.3
led4 = L2.4
...
trigger = 02

```

```

[algoquencer]
select = _TRACK
selectat = 2 # track 3
# and so on...

```

Notes:

- The three **algoquencer** circuits are mapped to the same buttons but at any time just one them uses them and displays its state at the LEDs of these buttons.
- Since the **butongroup** outputs the values **0**, **1** and **2**, the first track (aka "Track 1") is selected by **0**, not by **1**.

Important: CV inputs of **algoquencer** like **activity** are *not* handled by the **select** input, even if you assign a pot to them. These are "dump" CV inputs that just use the value that is patched there. If you want your activity pot to be switched, as well, use additional **pot** circuits and use the **select** input at these, as discussed above.

Multi level menues or selections

Selections can be nested into several levels. Let's make an example: You have a top level **butongroup** made out of the buttons **B1.1 ... B1.4** on a B32. Each button selects

one of four instruments. Each such instrument is represented by one **arpeggiator** (see page ??).

The second level consists of eight buttons on the B32 - the buttons **B1.5 ... B1.12** - shall select the allowed scale notes for the arpeggio, such as **select1**, **select3** and so on. So altogether you have $4 \times 8 = 32$ settings, but just 12 buttons.

The implementation is straightforward if you keep in mind that you must not use the **led...** outputs of a **buttongroup** for something else than the actual LEDs, *if that group uses its select input*. Remember: the **led** outputs of an unselected circuit are inactive.

In the toplevel group of buttons this is not a problem, since it is always active. It doesn't use its **select** input:

```
# Select the instrument
[buttongroup]
    button1 = B1.1
    button2 = B1.2
    button3 = B1.3
    button4 = B1.4
    led1 = L1.1
    led2 = L1.2
    led3 = L1.3
    led4 = L1.4
```

The second level groups can directly use the **L1.1 ... L1.4** registers for their selection. But here we cannot use the **led** outputs for selecting the scale notes, since they will be inactive if the instrument is not selected. Instead, we can use the **buttonoutput** outputs. They always keep their value - regardless of the current selection.

```
# Scale notes of instrument 1
[buttongroup]
    minactive = 1
```

```
maxactive = 8
select = L1.1 # first instrument
button1 = B1.5
button2 = B1.6
button3 = B1.7
button4 = B1.8
button5 = B1.9
button6 = B1.10
button7 = B1.11
button8 = B1.12
led1 = L1.5
led2 = L1.6
led3 = L1.7
led4 = L1.8
led5 = L1.9
led6 = L1.10
led7 = L1.11
led8 = L1.12
buttonoutput1 = _SEL_INST1_1
buttonoutput2 = _SEL_INST1_3
buttonoutput3 = _SEL_INST1_5
buttonoutput4 = _SEL_INST1_7
buttonoutput5 = _SEL_INST1_9
buttonoutput6 = _SEL_INST1_11
buttonoutput7 = _SEL_INST1_13
buttonoutput8 = _SEL_INST1_FILL
```

In the **arpeggio** (see page [92](#)) circuit of instrument 1 you can now wire the selection cables to the corresponding inputs:

```
# Arpeggiator 1
[arpeggio]
    select1 = _SEL_INST1_1
    select3 = _SEL_INST1_3
    select5 = _SEL_INST1_5
    select7 = _SEL_INST1_7
    select9 = _SEL_INST1_9
    select11 = _SEL_INST1_11
    select13 = _SEL_INST1_13
    selectfill1 = _SEL_INST1_FILL
```

```
selectfill2 = _SEL_INST1_FILL
selectfill3 = _SEL_INST1_FILL
selectfill4 = _SEL_INST1_FILL
selectfill5 = _SEL_INST1_FILL
... # further stuff here
```

Notes:

- This example shows how you can use one **buttongroup** with eight buttons and **maxactive = 8** as a elegant replacement for eight individual **button** circuits.
- Other use cases might prefer the **output** of the button group instead of the **buttonoutput** outputs.

Dealing with unused buttons

You might have situations where some of the buttons are *not selected at all*. With this I mean that none of the selected circuits use them. DROID doesn't touch the LEDs in these buttons and they keep their last state. This can be confusing and you probably will want to switch LEDs in unused buttons off.

You do this by using a **buttongroup** circuit where you don't map the buttons, just the LEDs, and set **maxactive = 0**. And you make sure this "dead" button group is selected in the above situation:

```
[buttongroup]
    select = _SOME_SELECT
    maxactive = 0
    led1 = L2.5
    led2 = L2.6
    led3 = L2.7
    led4 = L2.8
```

The upper example switches of the LEDs **L2.5 ... L2.8**, whenever **_SOME_SELECT** is not zero.

Overlaying switches of the S10

People keep asking how they can put multiple functions on the rotary or toggle switches of the S10. I must admit that I haven't found a good way to do this. The LED in a button can be switched as the function switches. In a pot I always can detect some movement. But how would you

deal with the fact that the current position of a mechanical switch does not match its "logical" position. OK, you toggle a switch back and forth after switching the mode, in order to show that you want to change its value. But that's not really fun to do.

So right now, the S10 is for the true believers in the "one switch one function" principle.

Overlaying faders of the M4

The motor faders in the M4 are *meant* to be overloaded with multiple functions. It's really what makes them stand out against all other input devices. Other than pots they can correctly show their current value physically. And they even can behave as switches with discrete position if needed.

Using the faders of the M4 is done by dedicated circuits. Please refer to the chapter about the M4 for details (see page 43).

3.2 Presets

Introduction

If you look carefully through the description of all circuits, you will find some that have a **preset** input. Among these are **algoquencer** (see page 80), **button** (see page 103), **butongroup** (see page 108), **calibrator** (see page 111), **faderbank** (see page 147), **fadermatrix** (see page 149), **matrixmixer** (see page 174), **motoquencer** (see page 205), **motorfader** (see page 226), **notebuttons** (see page 231), **nudge** (see page 234) and **pot** (see page 242). All these circuits have in common that they have some internal "state" that can be changed by user interaction. For example in **algoquencer** this state comprises the current trigger pattern that you've entered with the buttons.

A preset is one "memory slot" where you can load or save the circuit's state. This is done with the inputs **preset**, **loadpreset** and **savepreset**. When you load another preset, the circuit immediately switches to a different state. This does *not* mean that it does a reset of the current running state: For example the **algoquencer** does

not jump to the first step when you load a preset.

For internal reasons the total memory that a circuit can use for its state is limited. Therefore, each of the upper circuit provides a different number of presets. For example the **algoquencer** has 16 presets whereas **motoquencer** has only 4. Hereby the currently active state does *not* count as a preset, so **motoquencer** has five times the memory for storing its state: the currently active one plus the four presets. All these five states are automatically saved to your SD card whenever there is a change.

Switching presets with a button press

Switching between the presets can be done in two ways: in *triggered mode* and in *immediate mode*. Let's start with the triggered mode. Here you need to use all three mentioned inputs:

- The input **preset** tells the circuit which of the pre-

sets to load or save. The first preset has the number **0**, the second is **1** and so on.

- A trigger to **loadpreset** loads a preset into the circuit.
- A trigger to **savepreset** saves the current state of the circuit into a preset.

Typically you would use a **butongroup** (see page 108) to specify the preset number. If you have a S10 controller, it's straight forward to use one of the rotary switches for the preset number. But you can also turn a normal pot into a rotary switch by using the circuit **pot** (see page 242) and set **discrete** to the total number of different presets that you want to use.

Here is an example of switching presets in an **algoquencer** using a pot. We use the full 16 presets. Loading is done with button **B1.1** and saving with button **B1.2**. Note: the preset numbers start from **0**, so it's a perfect match for the **discrete** function:

[p2b8]

```

[pot]
pot = P1.1
discrete = 16 # output will be 0 ... 15
output = _PRESET

[algoquencer]
preset = _PRESET
loadpreset = B1.1
savepreset = B1.2
...

```

Notes:

- When you load a preset, changes to the current state get lost (if you haven't saved them before).
- The current state does *not* get lost when you restart your DROID or switch off your modular. It is saved to the SD card along with the presets.

Using long presses to avoid losing data

It's not entirely unlikely that you will press the wrong button from time to time. When that's your load or save button, you might overwrite some sequence that you've carefully crafted.

It's therefore a common trick to shield the preset triggers with *long presses*. Use a **button** (see page 103) circuit for each of the two buttons and use its **longpress** output. The **led** output is not necessary as the button has no state. Here is the upper example with the extra safety net enabled:

```

[p2b8]
[pot]
pot = P1.1

```

```

discrete = 16 # output will be 0 ... 15
output = _PRESET

```

```

[button]
button = B1.1
longpress = _LOAD_PRESET

[button]
button = B1.2
longpress = _SAVE_PRESET

```

```

[algoquencer]
preset = _PRESET
loadpreset = _LOAD_PRESET
savepreset = _SAVE_PRESET
...

```

Now the loading and saving just happens when you press and hold the respective button for at least 1.5 seconds.

Hint: If you are a more experienced DROID geek, you could try using a **burst** (see page 101) circuit to create a short blinking animation in the button whenever a preset is loaded or saved (left as an exercise).

Immediate switching of presets

The other way of switching presets is without triggers or buttons. This is even simpler to implement. Just omit the **loadpreset** and **savepreset** inputs:

[p2b8]

```

[pot]
pot = P1.1
discrete = 16 # output will be 0 ... 15
output = _PRESET

```

[algoquencer]

```

preset = _PRESET
...

```

Here are the differences to the triggered mode:

- As soon as you turn the pot (i.e. the **preset** input changes, a new preset is loaded.
- The current preset is automatically saved.

And a subtlety: because the current preset and the current state are essentially the same, you "lose" one memory slot. With immediate switching, **algoquencer** has just the four presets and no "extra" preset in the current state.

Things not stored in presets

Every now and then the question pops up why things like **activity** of the **algoquencer** are not saved in a preset. The answer is: the **activity** is not part of the internal state of the **algoquencer**. It's a CV input. Its value comes from the *outside*.

At first this might be counterintuitive if you map a pot to it (like **activity = P1.1**). But believe me: it's still a CV input. **algoquencer** cannot know that it's a pot. And if it would save that to a preset, and load it later: What should it do with the CV input? Should it be ignored in future? You see: lots of problems...

Still you might want to save the *pot's position* to a preset. And this can be done with a **pot** (see page 242) circuit, as we will see below.

Saving pots to presets

You might ask yourself: How can I get a preset for the position of a potentiometer, such as on the P2B8? Especially if I use it for controlling things like **activity** in an **algoquencer**?

The solution is very easy: Use **pot** (see page 242). It has a **preset** input. And then patch its **output** to the input that you want to control with the pot via an internal cable:

```
[pot]
pot = P1.2
preset = _PRESET
```

3.3 Tap tempo

There are a few circuits that have a **taptempo** input. Among these are **burst** (see page 101), **contour** (see page 126), **gatetool** (see page 160) and **lfo** (see page 163). Such an input is used to specify a time interval or a frequency. That's basically the same. For example an interval of 0.5 seconds corresponds to a frequency of 2 Hz. Sometimes that interval is then used as a gate length. The circuit **lfo** (see page 163) is an example of a circuit that uses this information as a frequency.

With **taptempo**, instead of specifying a number of seconds or milliseconds, you send a number of succeeding triggers. The time span between these triggers is used as the time interval.

There are two ways of using **taptempo** inputs. One way is, as the name suggests, a manual input. You can wire a button to the input and then "tap in" the time interval with a series of button presses. Here is an example with

```
output = _ACTIVITY
[algoquencer]
activity = _ACTIVITY
```

Of course you can combine that with the presets of **algoquencer** and switch the value of **activate** along with the actual sequencer pattern. Here is an example:

```
[p2b8]
[pot]
pot = P1.1
discrete = 16 # output will be 0 ... 15
output = _PRESET
```

```
[pot]
pot = P1.2
preset = _PRESET
output = _ACTIVITY
```

```
[algoquencer]
preset = _PRESET
activity = _ACTIVITY
...
```

Note: After loading a preset into a pot, its physical position does not reflect its logical value anymore (it would need a motor for that, just as the motor faders). Please look at the description of **pot** (see page 242) to learn how this works.

lfo (see page 163):

```
[lfo]
taptempo = B1.1
sine = 01
```

There are a few details that you should now when putting a tap tempo:

- Two button presses are enough to enter a tap tempo.
- If you press three times, the two intervals between the three presses are averaged so your tempo input gets more precise.
- If you press more than three times, just the last three presses are recognized.
- If you press the button and the last press was more than four seconds ago, you start a new row of

presses. So you cannot tap in an interval greater than four seconds.

- After you start your **DROID**, the **taptempo** is preset to 0.5 seconds (which corresponds to 2 Hz).

The second way of using a **taptempo** input is by patching a steady clock here. Most probably this will be your master clock. Since always the last three clock ticks ("taps") are recognized, the set interval is constantly updated to any changes in the speed of the clock. Please note:

- Speed changes in the input clock need some time to be recognized.
- When the input clock stops, the tap tempo is not set to zero or infinity, but simply keeps at the last setting.
- The **taptempo** input of the LFO does not keep the **phase** in sync. If you need that, patch the **sync** input in addition to the **taptempo** input.

4 Creating DROID patches with a text editor

4.1 General procedure

If you don't like to use the Forge, you can write patches by directly editing the text file. This is the general procedure:

1. Create a text file called **droid.ini**.
2. Copy this file to a micro SD card.
3. Insert the card into your **DROID** master.
4. Press the button on the **DROID** master.

If the **DROID** finds an error in your patch, LEDs will blink and tell you more about that error. Fix your error and try again. That's all.

If you have an **X7** expander attached to your master, the whole procedure is a lot easier. The **X7** gives you direct USB access to the SD card. The card is attached to your computer by putting the little switch on the **X7** to the left. This is like *inserting* the card into your computer. Now you can edit or copy your **droid.ini**. Afterwards simply put the switch back to its center position. That will remove the card from your computer (eject it first with your file browser). Also the patch will be immediately loaded by your master, no need to press the button.

Since the Forge operates on the same kind of text files,

you can open such a manual file with the Forge and also edit Forge-created files with a text editor. The Forge even has a simple built in editor for editing the patch or just parts of it in its text form.

Procedure in details

Here is the procedure again with some more details:

1. Use your PC, Mac or Linux box for creating a text file with the name **droid.ini**. A text file is not a MS Word file. In Windows you can create or edit a text file with Notepad or with some more convenient text editor. Note: some might want to edit **droid.ini** directly on the SD card. This is possible, of course. It's always handy, however, to have a copy of that file on your computer, just in case.
2. When you are finished, copy this file to the micro SD card your **DROID** has been shipped with or to any other micro SD card that is compatible with **DROID**. You need a micro SD card reader for this. Do not use any subdirectories on the card. Put the

file into the main directory. The card needs to be formatted with the standard FAT filesystem. If you buy a new card, it is most likely formatted that way anyway. Hint: If you like, you can create and edit your file directly on the card, of course. This saves the extra step of copying it.

3. Insert the micro SD card into the small card slot of your **DROID** master. Put it in with the metal contacts downwards. Be gentle, as always :-)
4. Press the button left of the SD card slot. Of course your **DROID** has to be powered up while you do this. The **DROID** now reads the file **droid.ini**, copies it into its internal flash memory and restarts, in order to load and activate the new patch. If everything is OK, one light will make one quick circle around the 16 LEDs and your patch is up and running. After that you can remove the card if you like. Your **DROID** does not need it anymore. Note: If you are using an **X7** expander, the memory card remains in the master module all the time. You also don't need to press the button on the master, just use the switch on the **X7**.

4.2 Basic structure of the patch file

Droid offers a long list of pre-programmed functionalities - called circuits - from which you can pick and choose for your needs. Each circuit takes input values, processes them and produces output values. It is your task to set the inputs to values you like. Such a value could be taken from a hardware input, a button, a pot, or simply be a fixed value. The outputs of the circuit can be connected to hardware outputs, LEDs or even to the inputs of other circuits in order to create more complex patches.

All this is configured in a simple text file with the name **droid.ini**, which is also called the **Droid patch**. Using a simple text file has lots of advantages:

- You can edit it with nearly every operating system.
- No special software is needed. This will probably

still work in 30 years, when you just have bought a vintage DROID on ebay for a couple of thousand bucks.

- You can easily post and share your DROID patches or patch snippets in our Discord community or on other internet boards.
- You can copy & paste parts from other one's DROID patches.
- You can add comments to your patch.

Here - again - is an example of a DROID patch:

```
[lfo]
    hz      = 0.5
    triangle = _CABLE_1
```

```
[contour]
    gate     = I1
    decay    = _CABLE_1
    sustain  = P1.1
    release  = I2
    output   = O1
```

As you can see the **droid.ini** is a list of circuit declarations. In the upper example we see two circuits: **[lfo]** and **[contour]**. Each one comes with a list of inputs and outputs which are assigned to jacks, fixed values or internal patch cables.

In the example all jack declarations are indented for better readability.

4.3 Finding a problem in your DROID patch

It is not entirely unlikely that you got something wrong in your patch, some syntax error, some invalid line, stuff like that. Humans make errors, but this is no big deal, since DROID helps you finding the reason and location of any problem in your DROID patch by two means:

1. It creates a file called **DROIDERR.TXT** on your SD card.
2. It flashes some LEDs in a certain way.

So if you experience any strange LED blinking after loading your patch, put the card back into your computer (or put the switch on your X7 to the left again) and look into the file **DROIDERR.TXT**, which should be there now. This file just contains one line, maybe like this one:

ERROR IN LINE 17: Invalid output '09'. Allowed is 01 ... 08

This tells you the exact location and reason of your problem so that you can easily fix it.

LED blink codes

As an alternative to the error file, the DROID master also shows the location and reason of the error in form of LED blink codes. There are two types of errors that you can make:

1. **General errors** concern the patch as a whole. The SD card is missing. You have misspelled the file

name. Things like that. In such a case *all* LEDs will flash in the same color. The color indicates the reason of the error. On the next page you find a table of all *global error codes*.

2. **Local errors** concern just one specific *line* in your DROID patch. In that case just some of the LEDs will flash. Again, the color shows you the reason for the error, according to the table *local error codes*. In addition, the LEDs show you the exact *line number* where your error occurs. This is done in the following way:

- The input LEDS 1 ... 8 indicate the *tens* of the line number. If the error happens to be in line 90, then LED 1 + 8 will flash. If it is in line 1 to 9, then no input LED flashes at all.

- The output LEDs 1 ... 8 indicate the *ones* and are added to that number. Again, if a 9 is needed, then 8 + 1 will flash.
- If your patch has more than 99 lines, then the error could be in line 100+. In that case one of the input LEDs will flash *white*. That LED

- indicates the hundreds of the line number.
- If the error is in some line at 900 or more, several LEDs will flash white. Just add them up. So e.g. if LED 2 and LED 8 flash white, this means 10 times 100, hence 1000.
 - The maximum line number that can be shown

that way is, if all eight LED flash white plus 99. That is $100 + 200 + \dots + 800 + 99 = 3699$. If your patch has even more lines, better look into the file **DROIDERR.TXT**. There you can see the line number of the error in clear text.

Examples for error codes

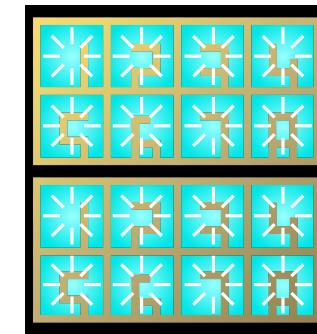
Invalid parameter value in line 81:



Invalid register in line 99:



The SD card was not found or could not be read:



Undefined parameter in line 90:



Line too long in line 144:



Too many circuits or out of memory:

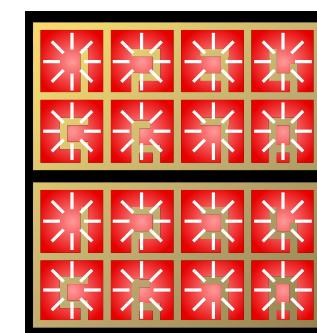


Table of error codes

All LEDs flashing at once (global error)

yellow	Patch not found: This can happen in the following situations: 1. No file with the name droid.ini is present on the memory card. 2. You DROID started without having loaded a patch ever. 3. You did a factory reset without loading a patch afterwards.
red	Too many controllers: You have declared more than the allowed number of 16 controllers.
blue	Patch is too big: The size of your droid.ini file is too big. The maximum of the size <i>without</i> spaces and comments is 64,000 bytes - which is quite a lot.
cyan	Out of memory: The circuits in your patch use too much memory. So you have too many large circuits or too many circuits in total. The memory consumption of each circuit only depends on its type. The smallest circuit is bernoulli and has a size of about 200 bytes. The largest circuits are midifileplayer with 7000 bytes and cvlooper with 18,000 bytes. Most circuits need between 400 and 800 bytes. And the total available memory is about 110,000 bytes.
magenta	Invalid firmware file: The firmware upgrade failed because the contents of droid.fw is invalid. The file is incomplete or corrupted.
white	No SD card found: No card could be found. Maybe you inserted it in the wrong way? Or your card is not supported. Or you pressed the button too early. Sometimes it helps to simple press the button again.

Note: If you get your *start animation* with just white LEDs instead of colored ones, your DAC calibration needs to be redone. See page [71](#) for details.

Just some of the LEDs flashing (local error in one line in **droid.ini**)

yellow	Unknown register: You used a non-existing register name (registers are the things like 01 , I7 and so on). Please check the list of allowed registers in this manual on page 27 .
orange	Unknown parameter name: that circuit does not support that parameter. Please check the circuit references in chapter 13 .
red	Unknown circuit: This type of circuit does not exist. Please check the exact spelling. Maybe you have an old firmware that does not support that circuit yet? On page 67 you learn how to do a firmware upgrade.
blue	Line too long: One line in your patch exceeded the maximum allowed line length of 63 characters.
green	Internal patch cable misused: One of your internal patch cables (see page 30) is not properly used: 1. No input: One patch cable is only used as output. 2. No output: One patch cable is only used as input. 3. Double output: One patch cable is used twice as an output.
magenta	1. Invalid header of circuit: DROID was expecting an opening square bracket [, but found something else. 2. Invalid parameter line: DROID was expecting something like clock = I7 , but found something completely different. Parameters always start with a letter. This is followed by an equals sign. 3. Invalid parameter value: Your parameter has an invalid value. Please checkout this manual about allowed values for parameters and their exact syntax.

4.4 Inputs, outputs and other registers

Your DROID has lots of inputs and outputs. Also its LEDs behave like outputs and buttons and pots behave like inputs. All these are called registers, because they behave like things that can store values. Each register consists of a special character followed by a number or number combination.

Most important of course are the eight CV input and output jacks **I** and **O**. With the normalizations **N1**, **N2**, ... **N8** you can specify a signal or value that should be used for **I1**, **I2**, ... **I8** when no patch cable is inserted. But we will come to that later.

When you have attached an G8 expander, you get eight more jacks called **G1** through **G8**. Each of these can either be used as an input or an output. They are simple gate inputs/outputs that just know "On" and "Off", or 0 and 1. When used as an output they output either 0 V or 5 V.

Starting with the blue-3 firmware and the new version of the G8 expander, you can add up to four G8s to your master. If you have more than one G8, you need a dot-notation for the gate names, for example **G2.7** for the gate 7 on expander 2.

The stuff on your P2B8, P4B2, B32, P10 and other controllers can also be accessed via registers. Here there is

always a dot in the name, separating two numbers, like **P1.2** or **B4.8**. The first number is always the number of your controller. The second number is the number of the element on the controller. So **B4.8** is the 8th button on the 4th controller. P10 controllers just have **P** registers, no **B** or **L** registers. Likewise the B32 has just buttons and thus no **P** registers.

Please note that each button has two registers: one with the letter **B** for the button itself. DROID will set that to **1.0** while the button is pressed (and hold) and to **0.0** otherwise. The second register is for the LED in the button and begins with **L**. This is an *output* register where you can write values to. A value of **0.0** will set the LED off, while **1.0** creates full brightness. But the LEDs also support any number in-between and will have a brightness according to that number. Negative numbers are treated like positive numbers here, so **-0.5** will produce the same brightness as **0.5**.

As long as you do not actively use the **L**-registers the LED in a button will automatically be lit while you hold it. Please look at the **button** circuit in page 103 for how to convert a push button into one that toggles its state on each press.

Overriding the LEDs of master, G8 and X7

The registers **R1** through **R56** let you override the function of the LEDs for the inputs and outputs, for the gates on the G8s and also for the LEDs of the X7 expander. This is sometimes very useful when you have a couple of unused inputs (and thus unused LEDs). Sending some internal values to one of these LEDs gives you some feedback about what your DROID is doing.

Sending a value of 0.0 to such a register makes the corresponding LED dark. Other values select a color at full brightness. Here is the table of colors (intermediate values give intermediate colors):

0.2	cyan
0.4	green
0.6	yellow
0.73	orange
0.8	red
1.0	magenta
1.1	violet
1.2	blue

Here is the complete table of all register types:

Register	Type	Description
I1 I2 I3 I4 I5 I6 I7 I8	input	The eight inputs of the DROID master
N1 N2 N3 N4 N5 N6 N7 N8	output	The normalization of these inputs. When nothing is patched into an input, the according I-register will take its value from the matching N- register instead. Any they are 0.0 if you have not set them.
O1 O2 O3 O4 O5 O6 O7 O8	output	The eight outputs of the DROID master
G1 G2 G3 G4 G5 G6 G7 G8	input/output	The eight gate jacks of the (first) G8 expander. Each can be used either as an input or as an output.
G2.1 G2.2 G2.3 G2.4 ... G2.8	input/output	The eight gate jacks of the second G8 expander. Use G3.X and G4.X for the third and fourth G8 expander.
G9 G10 G11 G12	output	The four gate jacks of the X7 expander. These are always outputs.
R1 R2 R3 R4 R5 R6 R7 R8	output	The colored LED squares in the first two rows (those for the inputs)
R9 R10 R11 R12 R13 R14 R15 R16	output	The colored LED squares in row three and four (those for the outputs)
R17 ... R48	output	The colored LED squares on the first, second, third and fourth G8 expander
R49 R50 E51 R52 R53 R54 R55 R56	output	The colored LED squares on the X7 expander
P1.1 P1.2 P2.1 P2.2 P3.1 P3.2 ...	input	The pots on your P2B8, P4B2 or P10 controllers. P3.2 is the 2 nd pot on your 3 rd controller.
B1.1 B1.2 B2.1 ... B2.1 B2.2 B2.3 ...	input	The push buttons on your P2B8, P4B2 or B32 controllers. B3.6 is the 6 th push button on your 3 rd controller.
L1.1 L1.2 L2.1 ... L2.1 L2.2 L2.3 ...	output	The LEDs in these push buttons
X1	output	Special register for displaying a value encoded in the master's 16 LEDs

4.5 Specifying numbers in your patch

Note: you always need to write the numbers in "plain" format, for example **0.01** or **12345.67** or **-5.0**. Scientific notations like **3.4^-10** are not allowed. It's also not allowed to write just **.5** instead of **0.5**.

There are two suffixes that you can attach to a number: **%** and **V**. Appending a percent sign basically divides the number by 100, so ...

```
pulsewidth = 45%
```

... is just the same as

```
pulsewidth = 0.45
```

Appending a **V** divides the number by 10, which is exactly what you need in order to convert a number to a voltage

to be output at a jack. So:

```
pitch = 2V
```

... is just the same as

```
pitch = 0.2
```

Sometimes this is easier to read. Please be just aware that the **V** is applied just to the number itself. You **could** write **1/12V**, but that is *not* $\frac{1}{12}$ V, but is $\frac{1}{12}V$, which is - when you convert the voltage back to a number - $\frac{1}{12}$, which is 0.8333. Whereas $\frac{1}{12}$ V would be 0.008333 - a hundred times smaller!

Some inputs or outputs behave like gates that only know 0 or 1, low or high, on or off. For your convenience you

can use the words **off** - which is just a short hand for **0.0**, and **on** - which stands for **1.0**, if you like. Here is an example:

```
[contour]
loop      = on
output    = 01
```

This is exactly the same as:

```
[contour]
loop      = 1.0
output    = 01
```

4.6 Attenuating and offsetting inputs

Attenuation / Amplification / Multiplication

Each *input* of a circuit (not the outputs!) has a built-in option for attenuation and offsetting. Attenuation is done by multiplying the input with a value. Well, if you "attenuate" with a number greater than 1, the name attenuation would not really be correct, since the signal in fact gets amplified and not attenuated.

Let's assume you want to control the **level** parameter of an LFO with the first pot of your first controller (see page [163](#) for details on the LFO circuit). That pot can be addressed with **P1.1**:

```
[lfo]
```

```
level = P1.1
output = 01
```

The pot has a range from 0 to 1, which corresponds to 0 V ... 10 V. That's maybe too much for your application. So let's limit the range to 5 V, which is the same as 0.5. This is done by multiplying the pot with 0.5:

```
level = P1.1 * 0.5
```

Now **level** will range from 0 V to 5 V.

The attenuation does not need to be a fixed number. Let's CV control the level of the LFO with the external input **I1**. Now we multiply that with the pot **P1.1**, which makes the latter an attenuator for the CV. How cool is that?

```
level = I1 * P1.1
```

Fixed numbers can also be negative. The following line basically *inverts* the LFO's output since its output voltage is negated:

```
level = P1.1 * -1
```

If you like, you can use a short hand for that:

```
level = -P1.1
```

But that is really just an abbreviation for **-1 * P1.1**. From that follows, that **-P1.1 * I1** is **not** possible, since this would be **-1 * P1.1 * I1**, which would be two multiplications!

Division

There is another shorthand: It is allowed to use division, if the thing you divide by is a *fixed number*. So Instead of `pitch = I1 * 0.0833333` you can write:

```
pitch = I1 / 12
```

Again, this is a short hand for `I1 * 0.0833333` and this its treated as a multiplication. For that reason you cannot write `I1 / P1.1` or anything similar, since here the DROID would really have to do a dynamic division with the current value of `P1.1`. Use the `math` circuit for such things (see page [172](#)).

Offsets / Summing

An *offset* is applied by adding a number. This must be written after the (optional) attenuation. Let's have the level of the LFO set by `P1.1` but be at least 2 V:

```
[lfo]  
level = P1.1 + 0.2
```

Now the level would range from 2 V to 12 V. Since 10 V is the maximum, we could multiply the pot with 0.8 first, which results in a range from 2 V to 10 V:

```
level = P1.1 * 0.8 + 0.2
```

Again you are not restricted to fixed numbers. You can also use any `DROID` register you like. In this example

we use `P1.1` as a coarse tune and `P1.2` as a fine tune (20 times finer) for the rate of an LFO:

```
[lfo]  
square = 01  
rate = 0.05 * P1.2 + P1.1
```

Using `+` can even be used for mixing together two input signals. The circuit `copy` just copies an input to an output, but since the offset can be used with any register you can build a simple CV mixer:

```
input = I1 + I2
```

Note: If you want to sum more than two signals, use the `mixer` circuit (see page [204](#) for details).

Subtraction

Mathematics says, that subtraction is nothing else than the addition of a negative number. So you can subtract `0.5` from `P1.1` by writing:

```
input = P1.1 + -0.5
```

Since this looks clumsy, you are allowed to write as a short hand:

```
input = P1.1 - 0.5
```

Note: you *can* also use the negation on a register:

```
input = I1 - I2
```

But note: here this is an abbreviation for `-1 * I2 + I1!` So you already have “used up” your multiplication, even if you don't see it. The general rule is: If DROID can transform your line into the form `A * B + C`, everything is good.

Summary and Further notes

- Generally the format is `A * B + C`. So you are limited to one attenuation (multiplication) and one offset (addition / subtraction)
- Each of A, B and C can be a fixed number, any of the registers or an internal patch cable (for those see page [30](#)).
- Attenuation must be written first, offset last.
- There are some abbreviations for subtraction and division. They work if the thing can be transformed into `A * B + C`.
- No other operations are allowed (no brackets, additional operations, divisions, etc.)
- If you need more complex math operations, have a look at the `math` circuit (see page [172](#)).

Are you curious *why* DROID does not allow more complex operations here? Why is it so restrictive? The reason is a matter of CPU performance! When your patch is parsed, everything is converted to `A * B + C`. If you don't use the multiplication, B is set to `1`. No offset? Then C is `0`. So when it comes to the real time computation of these values, it's just the simple `A * B + C`. No conditions to be tested, no if/then/elses or similar stuff. It's really super fast. And that's important because you want your DROID to have low latency and smooth envelopes.

4.7 Internal patch cables

One of the fun parts is the fact, that internally you can connect several circuits without using any real inputs or outputs. Instead of an output you simply put a name of your choice that begins with an *underscore*. That same name can be used at another circuit as an input. Here is an example of an internal LFO triggering an envelope:

```
[lfo]
    square = _TRIGGER

[contour]
    trigger = _TRIGGER
    attack = 0.0
    release = 0.2
    output = 01
```

This patch cable is always a multiple, so it can be used by more than one circuit:

```
[lfo]
    square = _TRIGGER

[contour]
    trigger = _TRIGGER
    attack = 0.0
    release = 0.2
    output = 01

[contour]
    trigger = _TRIGGER
    attack = 0.5
```

```
release = 0.8
output = 02
```

Note: There are two rules that are checked by the DROID. And it will show an error message in green if one of these are found to be broken (see page 23 for an explanation of the error codes).

1. Each internal patch cable must be used as an input *and* as an output (otherwise it would be useless).
2. No internal patch cable may be used *twice as an output*. This would make no sense and is in effect a short circuit.

4.8 Using outputs as inputs

There is another way of connecting circuits: You can use an *output* register as an input to another circuit. The following example creates an LFO that outputs a square wave to LED R1, in order for it to flash in the speed of the LFO. R1 is the LED designated for input 1, but we simply misuse that as a signal LED for our LFO. Then an eu-

clidean rhythm is triggered with that same signal, simply by using R1 as an input here:

```
[lfo]
    hz = 2
    square = R1
```

```
[euklid]
    clock = R1
    length = 12
    beats = 5
    output = 01
```

4.9 Using inputs as outputs

Using input registers as outputs is not allowed. And it would not make any sense. If you try so, you will get a yellow blinking error message for the according line.

Look at the following example. Here - due to a copy & paste error - the LED states are sent to the button regis-

ters. That won't work. And for that reason DROID won't allow it:

```
[buttongroup]
    button1 = B1.1
```

```
button2 = B1.2
button3 = B1.3
led1 = B1.1 # Argr. should be L1.1!
led2 = B1.2 # Argr. should be L1.2!
led3 = B1.3 # Argr. should be L1.3!
```

4.10 Parameter arrays

Some of the circuits have arrays of similar jacks, like **output1**, **output2**, **output3** and so on. Here you can al-

ways omit the digit **1** if you just want to address the first jack in the list. So **output** is just the same as **output1**.

4.11 Comments & spaces

You can use comments in your **DROID** patch by making use of **#**. Then all further text until the end of the line is being ignored: **# Here comes the envelope for the foobar voice**

```
[contour]
trigger = _TRIGGER # wired to sequencer
attack = 0.5 # another comment
release = 0.8
```

```
output = 02 # wired to foobar trigger
```

4.12 More than one patch on the memory card

Sometimes you might want to have more than one **DROID** patch on your card and switch back and forth between these without going back to your computer. This can be done if you have at least one controller with buttons, such as P2B8, P4B2 or B32.

It goes like this: Put your additional patches on the card with special filenames in the format **droidXY.ini**, where *X* is the number of the controller and *Y* the number of the button. Then for example **droid14.ini** will be loaded if you *first press and hold* the button 4 on your first controller while then pressing the load button on the master.

This way if you have one P2B8 you can choose between nine different patches. If you have a second P2B8 controller, this extends to 17 patches, because now holding button 1 on controller 2 will load **droid21.ini** and so on. A B32 gives you a total of 32 alternative patches to load and so on. And yes: if you have 10 or more controllers and some B32 amongst them, **droid124.ini** would be

loaded by button 24 on controller 1, but also by button 4 on controller 12.

Important: It is crucial that *every* of your patch files contains the appropriate **[p2b8]** or other controller declarations! Otherwise you won't be able to switch over to the other patches since button presses will not longer be registered by the **DROID** master. It will instead fall back to the normal **droid.ini** in that case.

If you load a patch that way, the states of your circuits are saved in a special file that accompanies the patch. The name of that file is **DSTA*XY*.BIN**, so for example **DSTA14.BIN** if you load the patch **droid14.ini**. All you need to know is that each patch has its separate state. So if you e.g. have an **algoquencer** in each of two patches, its patterns will separately loaded and saved.

5 Controllers

5.1 Installing the controllers

Controllers are easy to install and use. The picture on the right shows the back of the P2B8 controller, but the other controllers look similar.

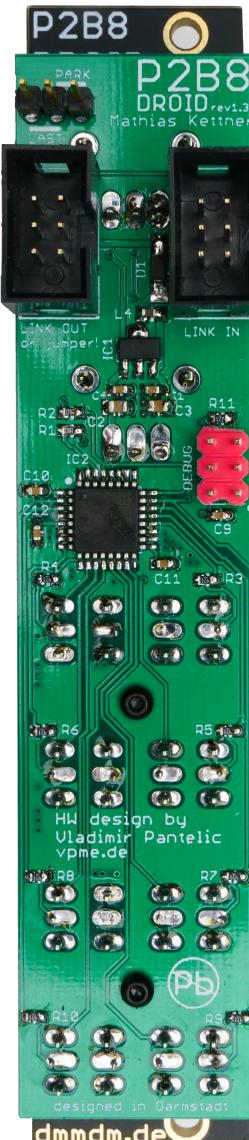
Each controller has two 6-pin connectors that are *mounted in boxes* (shrouded). They are labelled "LINK OUT" (left) and "LINK IN" (right). These connectors are for building a chain of controllers. Don't mix this up with the 6-pin header that is labelled "Debug", which doesn't have a box!

With your controller you got a 6-pin ribbon cable. Connect one end of it to the shrouded 6-pin controller connector of your master and the other end to the "LINK IN" of your first controller.

Take another 6-pin cable and wire the "LINK OUT" of your first controller to the "LINK IN" of your second controller. Continue until all controllers are chained together.

Finally: Every controller also has a three-pin header with the labels "LAST" and "PARK". When you get the module there is a small connector ("jumper") between the two pins that are labelled "LAST". This jumper is crucial for making the chain work. Here is the rule:

- On the *last* controller, the jumper must be in the position "LAST".
- On *all other* controllers, the jumper must be in the "PARK" position or removed (The park position is just for your convenience that you don't lose the jumper).

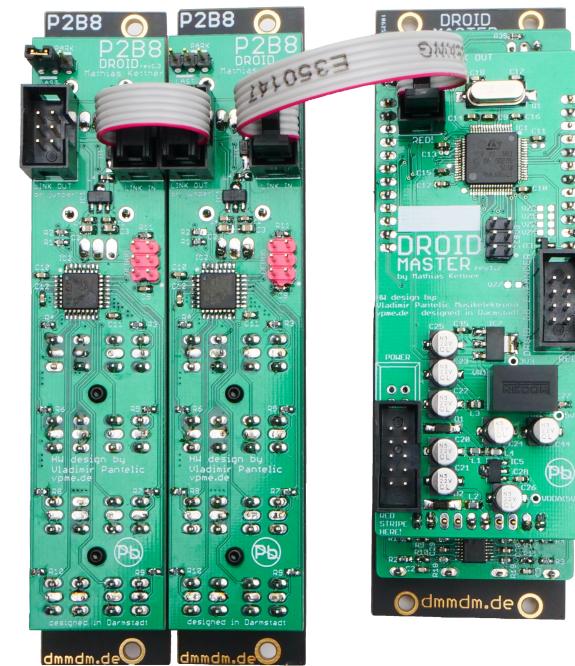


Jumper for terminating the chain

Use these connectors.

Don't use this one!

This is how a setup with two P2B8s on a master looks like:



If you switch on your system after connecting the controllers, those with LEDs should make a short power up animation. This does *not* mean that they are wired and jumpered correctly, though. To make a real test, you need to prepare a DROID patch, as you will see below.

Note: The M4 controller (see page 43) needs an additional power connector to your Eurorack system. The other controllers are powered by the master.

5.2 How to use controllers in your patch

Working with the Forge

Before you can use the controllers in your patch, you need to declare them in your patch. If you are working with the Forge, that's super easy. Double click on the top area with the modules, click the "Controller" icon on the left, or use the menu entry *Edit / New controller....* This brings up a collection of controllers:



Double click a controller to add it to your patch. Make sure that the controllers are in the same order as you have wired it to the master - from left to right. In case you have mounted your master on the right side and the controllers from right to left, you can switch how Forge displays your patch with *View / Show master on the right side*.

Now if you want to use one of the controls, bring the cur-

sor in your patch to the cell that shall "receive" the value of the pot or button and click on this control in the rack view. The Forge then inserts something like *Button B2.7* into this cell. This means *Button 7 on controller number 2*.

Working with the motor faders in the M4 is a bit more complex. Please have a look into the chapter about the M4 (see page [43](#)).

Working with a text editor

If you write your patch with a text editor, Just write one line with the content **[p2b8]**, **[p10]**, **[b32]**, **[p4b2]**, **[s10]**, **[m4]** or **[p8s8]** for each for your controllers at the top of your patch. The order of these declarations must match the order of your controllers in the chain, beginning with the one that is directly connected to the master. Here is an example with two P2B8s followed by one P10:

```
[p2b8]  
[p2b8]  
[p10]
```

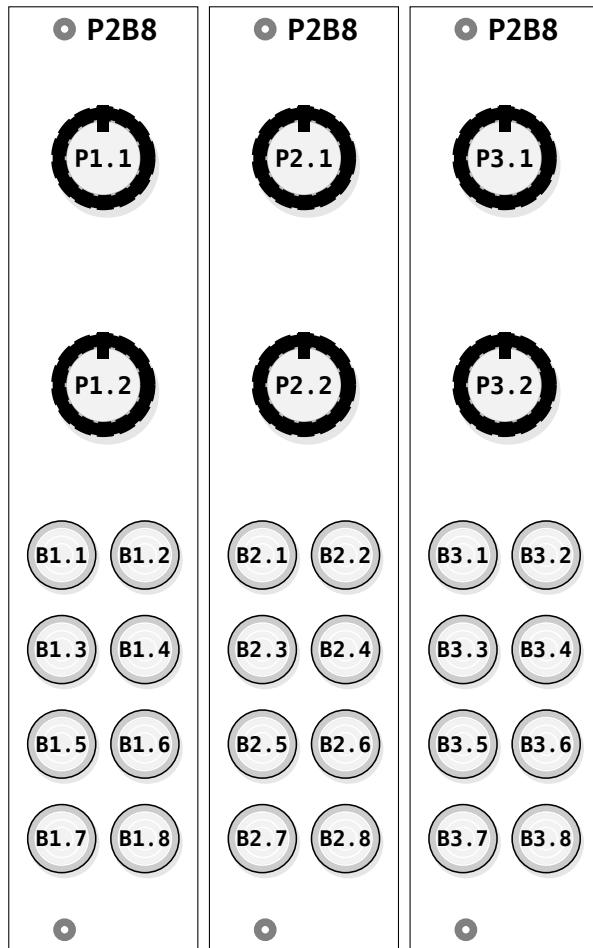
Now you can use the pots, buttons and LEDs by indicating these special registers in your patch as follows:

Px.y	potentiometers
Bx.y	buttons
Lx.y	LEDs in buttons
Sx.y	switches (S10 and P8S8)

Replace x with the number of the controller and y with the number of the pot, button, LED or switch on that controller. Examples:

- **P1.2** is the *second* pot on the *first* controller
- **B3.8** is the *eighth* button on the *third* controller
- **L3.8** is the LED in that button

Here is a schematics of the numbering of three P2B8 controllers:



Look at the following example. Here we have three controllers attached to the master: One P2B8, then one P10 and finally one more P2B8. Then we use some of the pots of the P10 for controlling the timing of an envelope circuit:

```
[p2b8]
[p10]
[p2b8]
```

```
[contour]
trigger = G1
output = 01
attack = P1.3 * 0.3
decay = P1.4
sustain = P1.5
release = P1.6
```

Details on the potentiometers

The potentiometers of the P2B8 and P10 output a number in the range 0.0 ... 1.0. This corresponds to a voltage from 0.0 V to 10.0 V. Wherever there is a CV parameter in a circuit (labelled $\wedge\vee$ in the table of inputs) you can set a pot here. An example would be an envelope generator:

```
[p10]
```

```
[contour]
gate = G1
output = 01
attack = P1.3
decay = P1.4
sustain = P1.5
release = P1.6
```

If you do not like the range of the pot you can easily change it by attenuation and offsetting as described on page 28. Let's make attack just go from 0.0 to 0.3:

```
[p10]
```

```
[contour]
```

```
gate = G1
output = 01
attack = P1.3 * 0.3
decay = P1.4
sustain = P1.5
release = P1.6
```

Of course you could use the *same* pot for more than one input. The following example use one single pot for attack, decay and release - with different scaling, however!

```
[p10]
```

```
[contour]
```

```
gate = G1
output = 01
attack = P1.3 * 0.3
decay = P1.3 * 0.5
sustain = P1.4
release = P1.3 * 0.7
```

Sometimes you want to use a potentiometer in a *bipolar* way - e.g. with a range from -1.0 to 1.0. This can be achieved by multiplication with 2 and subtracting 1:

```
[p2b8]
```

```
[copy]
input = P1.1 * 2 - 1
output = 01
```

For more complicated tasks about pots there is the circuit **pot** (see page 242). Here are some of its features:

- Make it easy to exactly dial in 0.5 by creating an artificial notch.

- Overlay the same pot with several independent virtual values.
- Easily create a bipolar pot with access to the left and right half of the values.
- Use the master's 16 LEDs for highlighting the current pot value

Details on the buttons

The buttons like on the P2B8, B32 and so on yield a value of **1.0** while pressed *and hold* and **0.0** otherwise. While this is sufficient for using them as trigger, in most cases you want the button to toggle its state between on and off each time you press it.

5.3 Troubleshooting

Here are the most common reasons why controllers don't work as expected. If you have trouble with the controllers, please try the following before you reach out to our community or us. We have a production error rate of less than 1 in 1000 modules so far. So the chances are huge that you can fix your problem yourself.

Jumpers: If your LAST/PARK jumpers are not set correctly, the controllers will powerup anyways. The LEDs will show their boot up animation. A patch might even be able to use the LEDs in the buttons. But you won't get button presses or pot positions back to your master. That's because the jumpers organize the transportation of the output data of the whole chain back to the master.

IN/OUT swapped: If you mix up the two connectors, the LEDs on the module will still light up on boot time. But no communication works. It happens to me all the time, since it's easy to get confused by the fact that left/right

Here the circuit **button** helps (see page 103). It converts a push button into an on/off switch. The following example uses **B1.1** in order to switch an LFO between unipolar and bipolar:

[p2b8]

```
[button]
button = B1.1
led    = L1.1
```

[lfo]

```
bipolar = L1.1
sine    = 01
```

Please note, how the LED **L1.1** is set by the button, so

that you have visual feedback of the current state. And since that register contains **0** or **1** depending on the button's state it can directly be used for the input **bipolar** of the LFO.

The **button** circuit can do much more interesting things, for example:

- Create buttons with three or four toggle states
- Combining more buttons into a group, similar to "radio buttons".
- Overlay one button with several independent functions
- Detect double clicks and long presses

See page 103 for all the details.

changes when you turn the module around.

Wrong declaration of controllers in your patch: The controllers need to be added in their correct order to your DROID patch. Make sure that you have setup the controllers in the FORGE in their correct order from left to right. If you mix them up, they get garbled data from the master that they cannot interpret.

Bad cables: This happened, even if it's super rare. If you are unsure and you have more than one cable, make a setup with just one controller on the master. Make a simple patch that uses that single controller. Now try your other 6-pin cables. If one cable works and another doesn't, it's an almost 100% indication that you have detected a broken cable.

M4 blinking in rainbow colors: If the four LEDs of a M4 controller (see page 43) flash in the alternating four col-

ors red, green, yellow and blue, it indicates that it does not have a communication with the master. It does this in any of the upper situations. So checkout the hints. If it slowly "pumps" in one color (starting with red, then yellow, then green), it's currently charging its super capacitors and needs some time to get ready to work (1-2 minutes at most).

Bad module: If you really got a bad module, we apologize! You just won a 1 in 1000 price for bad luck. But you still get a chance to get things to work. There is a chance that the problem just appears if the module is *the last* in the chain, or if its *not the last* in the chain. If you have a suspicious module, try both situations. There was one defective module where just the "PARK" position was broken. The solution was to put that module as last or simply remove the jumper from PARK.

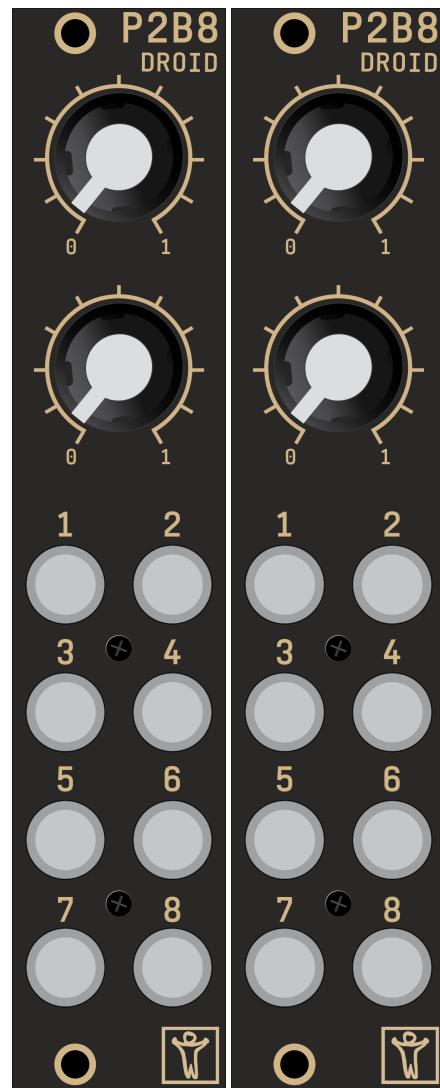
If your thing your module is defective, please contact out

community on Discord. It's still a good chance that you can fix it yourself. Sometimes people are blind. You module *really* looks broken, anyway: Please contact your dealer or us directly.

And here is last hint: If you have correctly declared your controllers in your DROID patch, the LEDs in the buttons should be lit as long as you hold the button (this is the default behaviour until you use the button in our patch).

If this works, that the communication with the master is working fine.

5.4 The P2B8 controller

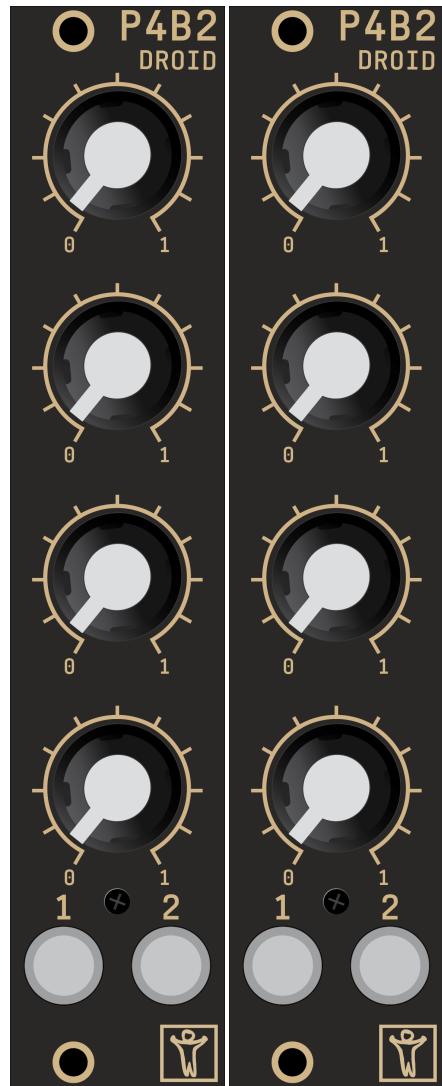


The P2B8 controller was the first available controller and is still the most popular, since it has a balanced number of pots and buttons and is very flexible. It is good choice if you have just one or two controllers.

On the first P2B8...

- the two pots are addressed with **P1.1** and **P1.2**.
- the buttons range from **B1.1** to **B1.8**.
- the LEDs in these buttons are **L1.1** to **L1.8**.

5.5 The P4B2 controller



The P4B2 controller give you four nice pots and still two buttons. Otherwise it's very similar to the P2B8. The P4B2 is a good choice if you like to work with a larger number of big pots.

On the first P4B2...

- the four pots are addressed with **P1.1** through **P1.4**.
- the two buttons are **B1.1** and **B1.2**.
- the LEDs in these buttons are **L1.1** and **L1.2**.

5.6 The P10 controller



The P10 controller has two big pots (the same as the P2B8 controller) and eight small pots. That makes a total of 10 pots, which are all behaving in the same way. They are numbered from **1** to **10**, so if your P10 would be the first in the chain, these pots are addressed in a **DROID** patch by

P1.1, P1.2, P1.3 ... P1.10.

The P10 is handy if you need to control lots of continuous values. The small pots are not as easy to operate as the big ones but they are very space efficient.

5.7 The S10 controller



The S10 controller has ten switches. They have the register abbreviation **S**. The first two are rotary switches and have eight positions. They output the discrete numbers **0, 1, ... 7**. The small switches just have three positions: **0** (down), **1** (center) and **2** (up).

In many cases the output values of the switches can be used directly for controlling something. In other situations you might want to use the **switch** circuit. It's a perfect solution for having the switch select one of a list of values. Here is an example:

```
[switch]
    offset = S1.1
    input1 = 0
    input2 = 2
    input3 = 3
    input4 = 5
    input5 = 6
    input6 = 10
    input7 = 11
    input8 = 100
    output1 = _FADERMODE
```

Here the switch 1 (**S1.1**) sets on offset to a **switch** circuit and sends one of the values **0, 2, 3, 5, 6, 10, 11** and

100 into the cable **FADERMODE**.

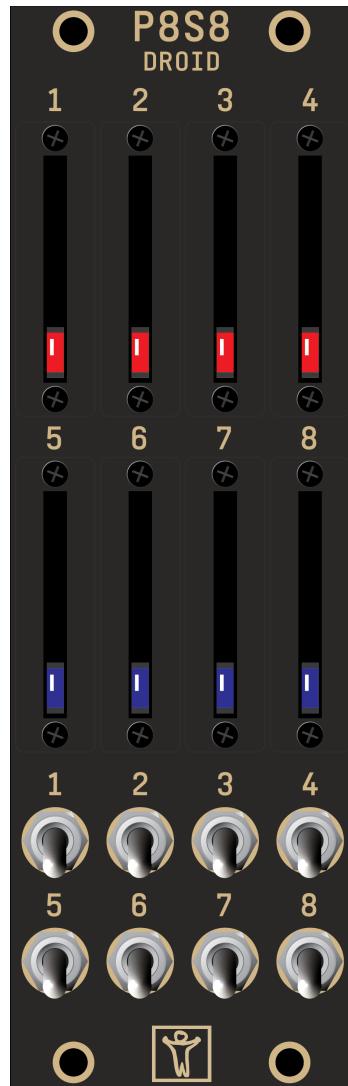
As always: inputs can be CVs. So you can also have dynamic inputs into the switch circuit. Here we use one of the small three-way switches to select one of three waveforms of an LFO:

```
[lfo]
    hz = 3
    sine = _SINE
    saw = _SAW
    square = _SQUARE
```

```
[switch]
    offset = S1.3
    input1 = _SINE
    input2 = _SAW
    input3 = _SQUARE
    output1 = 01
```

The switches are programmed in a way that if you move them fast, intermediate values will not be seen by the Droid circuits. So for example if you move one of the small switches directly from down (**0**) to up (**2**), the intermediate middle position with the value **1** will not get "visible", not even for a short time.

5.8 The P8S8 controller



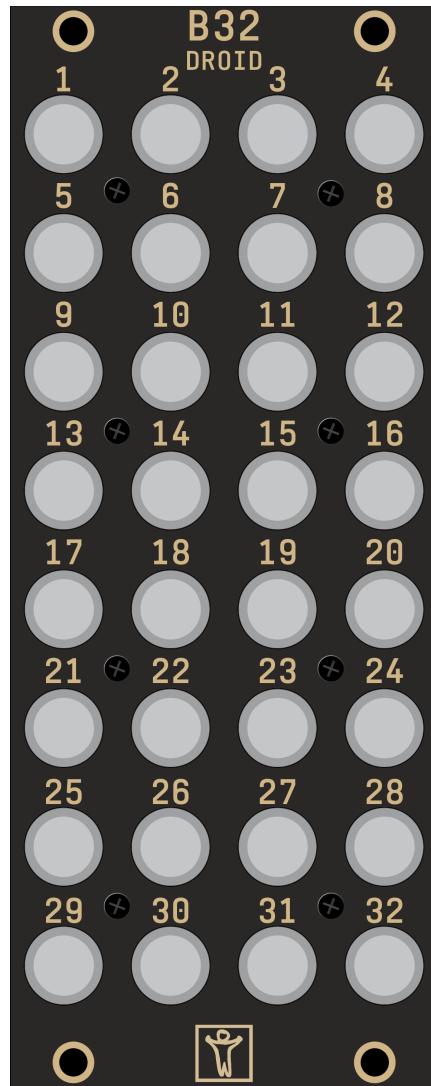
The P8S8 controller is for those who love those little sliders. The P8S8 has eight Alpha sliders with a range of 20 mm. They behave like the normal pots and are addressed with **P1.1** through **P1.8**. The bottom position is 0, at the top position their value is 1.

As a speciality the faders contain LEDs that can be controlled and used for any purpose. Use the registers **L1.1**

through **L1.8** for these. As long as you don't use the LED registers, the brightness of the LEDs reflect the current fader positions.

At the bottom the P8S8 has eight toggle switches - just the same as in the S10 (see page 40). These switches have three positions: **0** (down), **1** (center) and **2** (up). You access them with the registers **S1.1** through **S1.8**.

5.9 The B32 controller



You can never have too many buttons! And the B32 gives you not less than 32 of them. The B32 is a perfect companion for the M4 motor fader controller as the M4 provides lots of virtual “pots” and the B32 is handy for switching between these.

Of course the B32 is also a good play ground for trigger sequencers based on the [algoquencer](#) (see page 80).

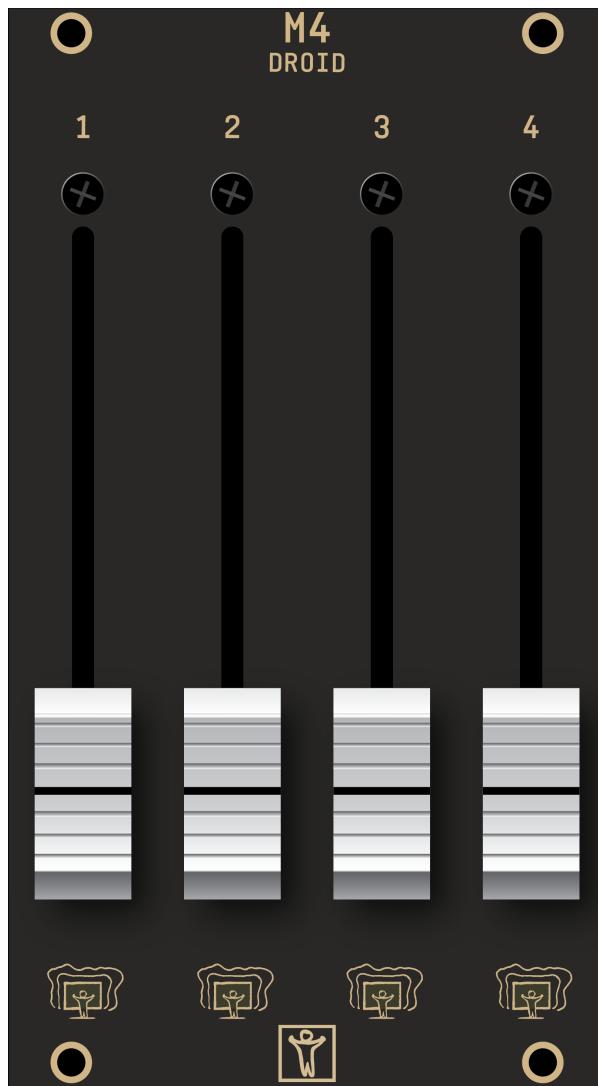
The buttons are numbered **B1.1** through **1.32** (as labelled on the face plate) and the LEDs accordingly **L1.1** through

L1.32.

They LEDs have one restriction: They just support four brightness levels: off, low, medium and full. This is a design decision for the sake of fast data transfer and low latency.

You will notice that the B32 is super fast in detecting button presses. You can slide with one finger through a column of eight buttons as fast as you like, but you will never make the B32 be too slow to detect one of the presses.

5.10 The M4 motor fader controller



Quick start

Here is how you get started with your M4 as fast as possible:

1. Wire the M4 to your master just as the P2B8 or any other controller. If the M4 is your last controller, set the green jumper to "Last", just as usual.
2. Connect the M4 to the bus power of your Eurorack case. It is the only **DROID** controller that needs its own power connection.
3. Declare the M4s in your patch with [**m4**].
4. Use the circuits **motorfader** (see page [226](#)), **faderbank** (see page [147](#)), **fadermatrix** (see page [149](#)) and **motoquencer** (see page [205](#)) for using the M4 in your patches.

Note: When you switch on the power, your M4 unit needs some time for charging their internal power system. That can last 60 - 90 seconds. While they are charging, here LEDs show a colored animation and go from red through yellow to green and finally off.

Installing the M4

You install the M4 just as all the other controllers (for more about controllers read about the P2B8 on page [37](#)): Connect the **IN** connector to your master with the 6-pin ribbon cable that came with your module. Make sure that you always use the *shrouded* pin headers (there is an additional 3×2 connector at the bottom which is just for debugging the hardware).

If the M4 is the last controller in your chain, set left jumper to *Last*. If other controllers follow, connect the next one to the **OUT** connector and remove the jumper or set it to **PARK**.

The M4 also needs a connection to the power of your Eurorack modular case. It will not take the power from the master (as the other controllers do). The reason is obvious: motor faders need a decent amount of power.

There are two more jumpers, labelled with **+150mA** and **+100mA**. These jumpers configure the power management. Read below for details and then decide which position you want to use. If you are unsure, put both jumpers into the right position (**+0mA**). In that setting each M4 needs up to 350 mA from your 12 V rail.

After you switch on your rack you will see an LED animation on the M4. It starts with red, then gets yellow, then green and finally the LEDs go off. This animation shows you that the power management of the M4 is charging its gigantic capacitors in order to provide the full strength to the motors later. During this charging phase the M4 will not respond to anything that happens in your patch.

Similar - when you turn off your rack - the M4 needs to discharge the capacitors for safety reasons. It does this by running all motors at full speed down and also doing an LED animation in white and blue. Just before the end the LEDs just glim red, because the green and blue part of the LEDs need a higher voltage and go off first.

Before unmounting the M4, switch off the rack and wait until this animation has stopped completely. Then it is safe to remove and put away the M4.

Using the faders in your patches

The traditional way of using motor faders is that you have several *presets*. Every preset holds a certain fader position. With some other control, e.g. a button, you can switch between presets and the new setting of the fader becomes active immediately. This is the classical application for mixing desks, where you can use presets for different mixes that you have prepared for different musical situations. You find general information about presets on page 19.

There is a second even more interesting application, however: You can assign multiple *overlaid functions* to one fader. For example one single fader could control attack, decay, sustain and release of an envelope. So just in order to save rack space and money you use one input device for controlling several parameters. In this application switching between the different functions does *not* alter any value. It just gives you access to control another parameter. And - as opposed to encoders - the motor faders act as a display for showing you the current values of the parameters.

The DROID motor faders are designed to do both applications: presets, overlaid functions and even both at the same time, because it make absolutely sense.

A speciality of the M4 - however - are its capabilities for *force feedback*. With the help of the motors it can simulate artifical *notches* or dents and thus convert a fader into a linear switch with a specific number of fixed positions. You can really feel these notches and that way easily switch between clock divisions, notes of a musical scale and whatever else you like - without the need of any display. It can also simulate something similar to a pitch bend wheel, where the fader always wants to move back into the center.

The most basic and elementary way to use faders in your patch is using the **motorfader** (see page 226) circuit. When you are creating patches with banks of many faders, please also have a look at **faderbank** (see page 147) and **fadermatrix** (see page 149). Those circuits manage a collection of faders with a single circuit and make your patches simpler.

In addition there is the **motoquencer** (see page 205) circuit which is a building block for simple and complex performance sequencers based on motor faders and the experimental specialised **firefacecontrol** (see page 154), which turns an RME Fireface audio interface into a motorized mixing desk.

As a starting point for further reading I suggest starting with the circuit **motorfader** (see page 226).

The touch plates

Below each fader the M4 has one touch plate with an integrated RGB LED. The touch plates are usable as buttons in your patch. Whenever a finger is touching the plate, the respective button register **B outputs 1**, otherwise **0**. In addition, the circuit **motoquencer** (see page 205) makes implicit use of the touch plates (and maybe some future circuits, too).

Unfortunately, however, touch plates don't have two definite metal contacts like in the buttons of the P2B8, B4B2 and B32, but work by measuring the time an internal capacitor needs to load. If you lay your finger on a touch plate, this time increases as some of the current is deviated into your finger and thus the loading time increases. Which means some inherent fuzziness and the touch plates need some preconditions in order to work reliably. If you experiance your touch plates not to react properly to your finger, check the following:

- The wetter your fingers are the better the plates work.
- They also work better, if your power supply provides a ground connection to the 120 V/240 V network.
- As a last resort touching some jacks of your modular with one hand while using the touch plates with the other hand will almost always work.

"Real" buttons would have been a better solution, but alas - there is simply not enough space behind the face plate for them. The motorized faders don't come in smaller sizes and we already have worked hard in making touch plates and LEDs possible. Consider the touch plates as a bonus add-on. If you don't like them, use the normal buttons in your controllers. Also, with the the **motoquencer** (see page 205), you can use the faders as an alternative for settings gates.

The LEDs

The LED below the touch plates can be accessed with an **L** register - just like in the P2B8. In addition, there is a **R** register that controls the color of the LED, similar to those on the master. If you just use the **R** registers, the LED will light in full brightness. If you just use the **L** register, the LED lights white in the brightness specified by the value you feed into that register. Using both **R** and **L** at the same time gives you control over brightness and color.

Registers

Here is the summary of all M4 registers, assuming that **[m4]** is your first declaration in your patch:

B1.1 ... B1.4	Touch plates
L1.1 ... L1.4	LED brightness
R1.1 ... R1.4	LED color
P1.1 ... P1.4	Current real physical fader values

The motor faders

The **DROID M4** has four industrial class motorized faders with 60 mm action range from ALPS. They are a combination of normal linear potentiometer with an electrical motor that can move that potentiometer. The motor is not a step motor but runs continuously. The M4 software determines the current position of the fader by reading out the value of the potentiometer and controls the motor to move to the desired position.

The motor control is done via pulse width modulation (PWM), whose frequency is way beyond the audible range.

Adapting the fader power

Using the circuit **droid** (see page 141) you can adapt the motor power of the faders. There are two settings. One is for the normal movement power (and hence speed). The other one is for tuning the power of the haptic feedback when you work with notches. Try mapping both parameters to pots and you can test their influence:

```
[droid]
m4faderspeed = P1.1
m4notchpower = P1.2
```

The power management

Motor faders are nice but need lots of power. As a matter of fact, one fader could use up to 800 mA from your 12 V rail when the motor is running at full power - if you would run it directly from the Eurorack power supply. So even a single M4 would need 3.2 A for full operation. That's a lot more than a typical power supply provides. And it's just one module! That's probably the main reason why we haven't seen flying faders in Eurorack sooner.

We have solved the issue in the M4 by means of modern supercapacitors (supercaps). Those little miracles can store up to 100 times more energy per volume than than electrolytic capacitors and can accept and deliver charge much faster than batteries. They also tolerate many more charge and discharge cycles than rechargeable batteries. The four supercaps of the M4 can deliver 3.2 A for the faders with ease - of course with the limitation of doing it just for a short time. That's not an issue in a normal usage pattern of the faders, since they move super fast and just for fractions of seconds.

When you power up your M4, you will notice that it takes some time to become operational. That is because it needs to load the supercaps before the show can begin. That time is somewhere in the range of 60 to 90 seconds. The current loading state is indicated by an LED travelling from left to right again and again. The colors starts red, goes yellow and gets green just before the module is powered up.

Note: when you work with the faders and let them jump back and forth very fast very often, it can be the case that the supercaps run out of power. In that case the fader motors will go off for a couple of seconds, the supercaps recharge and the powerup LED animation is visible (with green LEDs).

The M4 has an intelligent charging mechanism that manages the power of the supercaps and makes sure that there is enough power for fader movements while not exceeding a *limit* of current that is drawn from your Eurorack +12 V power rail. With two jumpers on the back of the module you can set the maximum charging current of the M4:

- The minimum charging limit of the M4 is 350 mA.
- With the left jumper you can raise that by 150 mA.
- With the right jumper you can raise that by 100 mA.

That way you can choose between 350 mA, 450 mA, 500 mA and 600 mA. The more power you allow the M4, the faster it charges up and the more fader movements per second it can do.

If you allow the M4 to draw too much current, your Eurorack power supply can overload. That might lead to various problems:

- It could overheat.
- It could blow its fuse.
- It could trigger its short circuit detection and switch off itself.
- The voltage of the 12 V rail could drop too much.

Please make sure that you use the M4 in a way that is within the specification of your power supply.

The good new for last: once the M4 is charged up and when you use the fader in a reasonable way, the power consumption of the M4 is much lower than the maximum limit. This is an important difference from modules like those with vacuum tubes that need their heating power all the time.

Discharging

When you switch power off, the M4 still has lots of energy stored in its supercaps. For safety reasons, it will discharge the supercaps as fast as possible as soon as it detects main power off. Discharge is done by constantly moving all fader motors downwards and lighting the LEDs in which with the maximum brightness - with one blue LED wandering from left to right.

At some point in time the voltage is not sufficient to drive the motors anymore. The LEDs are still animated. Later they will get red and slowly fade out.

Do not unmount the M4 from the rack until all LEDs are off!

This is important to avoid short circuits by accidentally connecting the supercaps with metal of the case or the like.

Software update for the M4

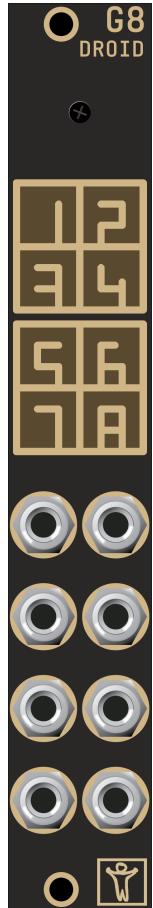
Because the M4 is much more complex than the other controllers, it has a more complex software that might need firmware updates from time to time.

The procedure is exactly the same as for the X7 (see page [57](#) with the following additional notes:

- The firmware file on the SD card must have the name **m4.fw**.
- In the master's maintenance menu the upgrade of the M4 is on position **6** (not 8 as the X7). And its color is yellow (not green).
- The M4 that you want to upgrade must be **the only module that is attached to the master!** The jumper on the lower edge of its back must be set to "Last".

6 The G8 expander

6.1 Introduction



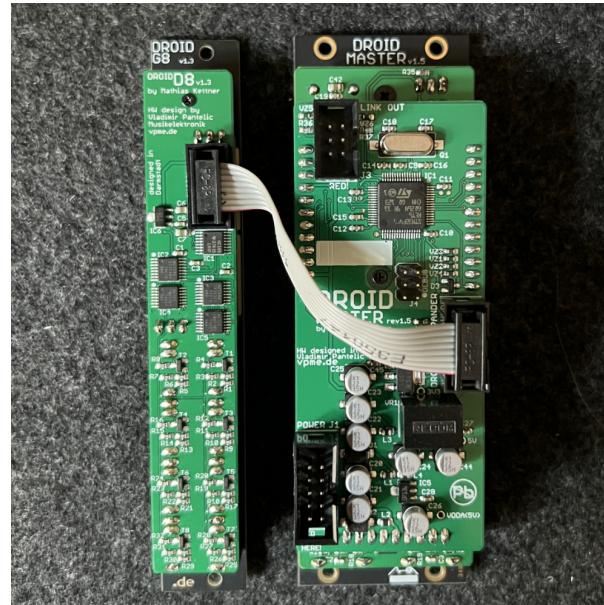
The G8 expander gives you eight additional jacks, each of which can be used as a gate or trigger input or output. They are ideal for working with clocks, gates and triggers, but can be used for simple CV modulations, as well.

There are two hardware versions of the G8. Version 2 was introduced 2023 and allows you to chain up to four G8 expanders to one master. For that purpose it has two connectors on the back: one to be connected to the master, one for the next G8.

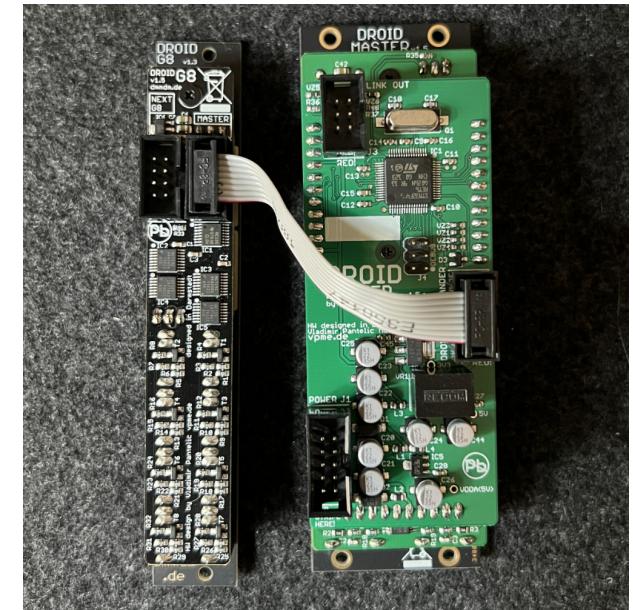
The original G8 version 1 has only one connector. There is no need to be sad if your G8 is version 1, since it still can work in a chain with more G8s if it is the last one. So if you want a second G8, simply get a version 2 one and use the old one as the second G8.

6.2 Installation

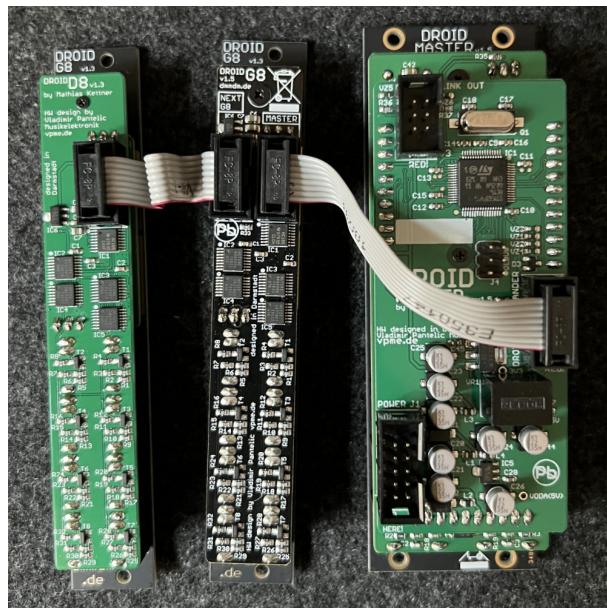
If you have just one G8 version 1, simply use the 8 pin ribbon cable that has been shipped with your G8 and connect the G8 to the 8 pin port of the master as shown in the following picture. Put the red stripe down in both modules.



The G8 version 2 has two connectors. Here use the right one labelled "Master":



To create a chain, wire the master to the “Master” input of the first G8, which must be version 2. Then wire the other connector of this G8 to the “Master” input of the second G8 and so on. No termination jumper is needed. The last G8 in the chain can either be version 1 or version 2.



6.3 Using the G8 in patches

You can access the jacks of the first G8 with the registers **G1**, **G2** ... **G8**. If you work with more than one G8, you need to use a dot notation and write the number of the G8 in the chain before the dot. So the gate 5 on expander 3 would be **G3.5**. You are also allowed to use that for the first G8 or if you just have one G8 (e.g. **G1.5**).

- Each jack can either be used as input or as output.
- When used as input it will read a value of 1 (= 10 V) at an input voltage of approx **0.75 V** or above and 0 otherwise (also for negative voltages).
- When used as an output they output **5 V** when you send a value 0.1 or higher, and 0 V otherwise.

Why do the gates not output 10 V? Well, while this would be more logical, but it was actually impossible to do in hardware easily since the G8 needs a very special chip that is able to switch between input and output via software. This chip does not support 10 V. 99.9% of all Eurorack modules will happily accept 5 V as a valid trigger. Some analog envelopes with vintage circuitry might need higher voltages. If you encounter such a module, you can use one of the outputs of the **DROID** master, which out-

put 10 V.

The G8 also has eight multicolored LEDs. These indicate inputs in blue lights and outputs in red, when high. You can override the default function of LEDs in order to display something or your own liking. Use the registers **R17** ... **R48** for that purpose.

There is nothing special to do in your **droid.ini** for setting up the G8 expanders. They don't need to be declared like the controllers. Using the **G** registers enables the expanders automatically. If you load a patch with **G** registers but don't have a G8, nothing dangerous happens and the rest of the patch will work normally.

7 The X7 expander

7.1 Quick start



You already know what the X7 is all about? Want to start immediately? Here is a super short quick start guide for experienced DROID users:

1. Wire the X7 to your master just like a controller. It must be the first in the chain.
2. Use the MIDI functionality via the circuits [midin](#) (see page 183), [midout](#) (page 190) and [midithrough](#) (page 199).
3. Access the four gates via [G9](#), [G10](#), [G11](#) and [G12](#)
4. Connect the USB cable and set the switch *left* for USB access to the SD card. Set it back to the middle position for disconnecting USB and loading the patch.

7.2 General overview

Features and applications

Welcome to the X7 expander. The X7 gives you USB and MIDI connectivity for your DROID and also four gate outputs with modular levels.

You can process incoming and generate outgoing MIDI streams, both via classical DIN cables and via USB. Both in and out directions support polyphony with eight or even more voices.

For size reasons the X7 uses 3.5 mm TRS jacks for MIDI instead of the classical DIN jacks. But it comes with two DIN ↔ TRS adapters, so you are free to use either form factor.

As a bonus feature, the X7 provides super fast loading of DROID patches via USB - without any need for putting the SD card in and out anymore.

Here are some examples of what you can do with the X7:

- Attach an external keyboard to your modular.
- Use an external hardware sequencer for playing melodies and beats in your modular.
- Use an external MIDI controller to control your DROID patch.
- Do the same with a MIDI controller app on your tablet or phone (via USB).
- Use your modular for playing polyphonic music and beats on your hardware synths or software synth plugins in your DAW, tablet or phone.
- Connect two DROIDS (both with X7) and exchange values and triggers via CCs and notes.

- Use the four additional gate outputs on the X7 for sending clocks, gates and triggers and free your valuable CV outputs for other things.
- Access the SD card in your master just like a USB thumb drive for direct access to it via your PC, Mac, phone or tablet.
- Alternatively load new patches to your master via MIDI sysex from your PC - *and get your new patch ideas up and running in less than a second.*

The switch

At the top the X7 has a **switch** with three positions. This switch selects the current function of the USB port:

left	Activate USB access to the SD card
middle	Don't use the USB port
right	Activate MIDI via USB

Beware: in the left position the master will not work as usual and does not run your patch. See below for details.

The jacks

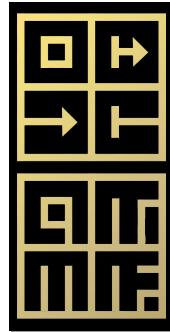
The X7 has the following jacks:

- One USB-C port for MIDI via USB and for access to the master's SD card from your PC
- One 3.5 mm stereo jack (also called TRS, which stands for "tip ring sleeve") for MIDI input, with *autosensing* for MIDI TRS type A and B
- One 3.5 mm stereo jack for MIDI output

- Four gate outputs for gate and trigger signals at modular level

This sums up to a total of seven ports, hence the name X7 (the original idea of naming it “U1M2G4” was soon abandoned, since that was too clumsy and also wouldn’t fit on the face plate).

The LEDs



Similar to the master, the face plate has multicolor LEDs indicating what’s going on at the seven ports:

- The top left LED shows the current state of the SD card in the master.
- The top right LED shows what’s going on on the USB MIDI connection.
- The LEDs in the second row show incoming and outgoing MIDI data at the TRS ports.
- The four LEDs labelled 9, 10, 11 and 12 show the current state of the four gate outputs.

7.3 Installation

The installation of the X7 is very easy. These are the rules:

1. Wire the X7 to the shrouded 6-pin header on the top right of the master, just like P2B8, P10 or other controllers.
2. There is no jumper. You don’t need one here.
3. Always install it **as the first module** in the chain!

4. Make sure that the switch is in the middle position when you start.
5. You can only attach **one X7** to your master.

Just like all the controllers, the X7 has an *input* connector, which is at the top *right* side if you look from the back. On the *left* side is the *output* connector. Connect the master with the shipped 6 pin ribbon cable to the *input* connector. If you have any controllers, like P2B8, P10 and so on, wire the first of these to the *output* connector of the X7.

That’s all. the X7 is powered from the master so there is no dedicated power cable.

Note: You don’t need to change anything in your DROID patches for now. Even if the X7 is connected to the master like a controller, it does *not* need to be declared. And it also does *not* count when it comes to the numbering of **P1.1** and so on.

7.4 USB access to your SD card

The X7 can give you direct access to the SD card of the master via USB. Start with the switch in its middle position. And make sure the micro SD card is in its slot on the master. The top left LED of the X7 always shows you dim white light whenever a SD card is present.



Now connect the USB-C port on the X7 with your PC, Mac, Linux, phone or tablet (I’ll just use “PC” for the rest of this manual) and set the *switch on the X7 to the left*. This enters “USB stick mode”.

Note: For a USB-C ↔ USB-C cable to work, your X7 must at least have hardware revision “Rev 1.5.1”. The revision is printed on the back of the module top right. Also you need at least the firmware “orange-912” on your X7 (see below for firmware upgrades). If your X7 has “Rev 1.3” or “Rev 1.2” or you have “orange-911” or earlier, the X7 needs a USB-A ↔ USB-C cable. For that reason such a cable is shipped together with the X7.

After a few seconds, your PC should detect a new storage device with the exact contents of the micro SD card. Since X7 is a “class compliant” mass storage device you don’t need any driver on your PC.

If you work with the Forge, you should see the *Save to SD* icon become active and you can use that to write your patch to the SD card. Much faster is using MIDI Sysex, however.

If you don’t like the Forge, you can edit **droid.ini** directly on the card or copy a patch from your PC to the card, just as you are used to when you are working with your SD card reader. The USB-Stick mode is also helpful

for getting the **ERRORS.TXT** or **STATES1.TXT** file from your SD card, even if you work with the Forge.

When you are finished, *eject* the volume / disk on your PC. After that set the switch back to its middle position. This will remove the USB connection and also automatically launch the new DROID patch. So you don't need to press the button on the master.

A few notes:

- If your patch has an error (blinking LEDs and stuff, see page [23](#)) put the switch back to the left, wait for the SD card window to popup and look for the file **DROIDERR.TXT**. Open it and you will see the exact reason for the error.
- The access to the SD card via the X7 is slightly slower than using an SD card reader on your PC since it takes the extra miles via the X7
- If you need to re-format the card for some reason, better do this in the micro SD card reader that was shipped with your master. It's much faster that way.
- If you are working with Mac and experience that the access is slow, check out disabling Spotlight on the card. A script for that can be found on page [72](#).

7.5 MIDI

MIDI features overview

One key feature of the X7 is working with MIDI. The combination of the DROID master with the X7 probably forms the most flexible, comprehensive and powerful MIDI converter in Eurorack land. Here are some of the key features:

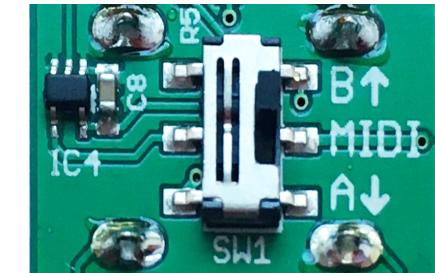
- Support for both MIDI → CV and CV → MIDI at the same time.
- Unlimited polyphony (number of simultaneous notes) except that you run out of jacks.
- The MIDI streams of USB and TRS can be used independently in parallel, so you have two input and two output streams.
- Flexible "MIDI through" routing while splicing in and out events
- Comprehensive support and access to the vast majority of MIDI features such as CCs, clocks, the running state, pitch bend, all types of pedals and much more.
- Automatic pitch stabilization detection in the CV/gate → MIDI conversion, thus working precisely with Eurorack sequencers and quantizers.
- Super fast DROID patch upload via USB-MIDI SysEx.

And of course you benefit from DROID's own flexibility when it comes to quantization, LFOs, chord generators, switches and all that stuff.

MIDI over DIN

For space reasons, the X7 uses 3.5 mm stereo jacks (TRS) for MIDI. But we ship two TRS to DIN adapters with the X7. Use these for connecting classical DIN MIDI devices.

Note: When you use one of the shipped adapters for the MIDI output via DIN, make sure that the switch at the back of the X7 is set to position B (up).



MIDI over USB

The X7 supports MIDI over USB. Hereby it acts as a *USB device*. This does *not* mean any limitation of being an input or output device. It can be both. Even at the same time. But the actual limitation is that the X7 cannot provide power to your MIDI devices and cannot be a USB host.

That means that MIDI devices that are USB devices themselves cannot be connected to the X7 via USB, even if you have a matching cable. Connect your MIDI keyboards and controllers with the TRS jack if USB doesn't work for you here.

But the USB port is perfectly suitable for connecting the X7 to your PC, Mac, tablet or phone. The MIDI implementation is "class compliant". That means that you do not need any driver software. Simply connect the X7 with the shipped (or any other) USB-C cable to your PC and set the switch to the right. You should now see a new MIDI device, which can be selected as input or as output depending on what you are going to do.

Note: As of now the USB-MIDI standard has a concept of up to 16 virtual MIDI "cables". The X7 receives data on all cables and always sends on cable 0. Future software updates might make this more flexible, if there is demand.

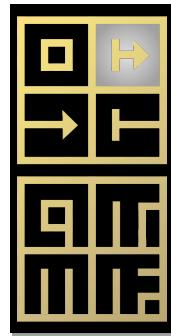
By the way: MIDI over USB is not restricted to the standard MIDI data rate of 31250 bits per second.

The LEDs

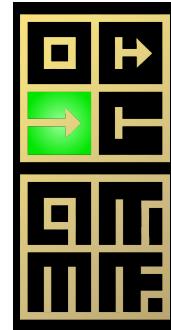
When working with MIDI, watch the corresponding LEDs. Here is what the colors mean:

black	no data transmitted
dim white	steady activity
green	note on
red	note off
blue	some other MIDI event

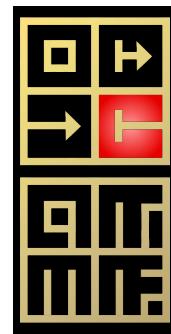
The top right LED shows the status of USB-MIDI:



The third LED shows MIDI data via *incoming* TRS:



The fourth LED shows MIDI data via *outgoing* TRS:



MIDI to CV (MIDI input)

The most common application for MIDI and modular synthesizers is converting MIDI note events to CV/gate signals. When you press a key on a MIDI keyboard or when a MIDI sequencer starts playing a note, a MIDI “note on” message is being sent over the wire. Likewise at the end of the note a “note off” message is sent.

A typical MIDI to CV module receives these messages and feeds at least two jacks: one with the *pitch* of the currently played note in form of the typical 1 volt per octave

scheme. And one *gate* output which is high (e.g. at 5 V) while the key is being held.

Of course there is much more, like clock signals, controllers and so on. This X7 can give you access to the vast majority of MIDI features.

The hardware connection is done either with the 3.5 mm TRS jack or via USB (or both at the same time). The X7 comes with two identical TRS → DIN adapters, so you can use the much more wide spread classical MIDI cables with DIN plugs.

Even if you don't use our adapters but use the 3.5 mm jacks directly, you don't need to care about MIDI “A and B”. The X7 does autosensing at its input. Either way will work. Just make sure you use *stereo* cables. Normal modular patch cables don't work.

The basic operation is super simple. All is done with the circuit `midiiin` (see page [183](#)). This example converts MIDI to a pitch CV at output **01** and a gate at output **02**:

```
[midiiin]
  pitch = 01
  gate = 02
```

The source is the TRS jack. But you can easily select MIDI via USB instead with the **usb** parameter:

```
[midiiin]
  usb = 1
  pitch = 01
  gate = 02
```

Per default, `midiiin` processes notes from all 16 MIDI channels. You can select one specific channel with the **channel** jack:

```
[midiin]
    channel = 5
    pitch = 01
    gate = 02
```

Note: You can use up to 32 **midiin** circuits in your patch. So you could add one circuit for each MIDI channel that you want to process.

For polyphonic patches with more voices simply specify more pairs of gate and CV. This example supports three simultaneous notes:

```
[midiin]
    pitch1 = 01
    pitch2 = 02
    pitch3 = 03
    gate1 = 05
    gate2 = 06
    gate3 = 07
```

If you have a G8 expander (see page [47](#)), you can directly control eight analog voices:

```
[midiin]
    pitch1 = 01
    pitch2 = 02
    pitch3 = 03
    pitch4 = 04
    pitch5 = 05
    pitch6 = 06
    pitch7 = 07
    pitch8 = 08
    gate1 = G1
    gate2 = G2
    gate3 = G3
    gate4 = G4
    gate5 = G5
    gate6 = G6
```

```
gate7 = G7
gate8 = G8
```

Notes have velocities, also there are MIDI *controllers* like the volume, the modulation wheel or more. These can directly be accessed via output parameters:

```
[midiin]
    pitch = 01
    gate = 02
    volume = 03
    modwheel = 04
    cncnumber1 = 17 # get CC number 17
    cc1 = 05      # output that on 05
```

Also you get simple access to various MIDI clocks and the start and stop status:

```
[midiin]
    clock = G1
    start = G2
    stop = G3
    running = G4 # alternative to start/stop
```

The MIDI notes needn't be used for playing voices. The following example uses the note for selecting a root note for a **minifonion** (see page [200](#)):

```
[midiin]
    pitch = _PITCH
[minifonion]
    root = _PITCH * 120
```

You even can use MIDI keys (maybe from controller pads) as buttons.

```
[midiin]
    note1 = 24 # MIDI note number of C-0
    notegate1 = _KEY_C
```

```
[button]
    button = _KEY_C
    onvalue = 0.8
    offvalue = 0.2
    output = 01
```

This was just a quick overview and there are much more inputs and outputs available. Please have a look at page [183](#) for more details on **midiin**.

CV to MIDI (MIDI output)

While MIDI to CV interfaces still are the vast majority of MIDI modules, the other way round becomes more and more interesting. With more and more complex quantizers, sequencers and other fascinating and inspiring CV modules people want to integrate existing hardware or software synths into their modular systems for playing melodies and beats that are generated by these modules.

For that task you need a CV to MIDI converter. That converts pitch and gate information that are present in form of CVs, into a stream of MIDI events and sends these over DIN or USB to the sound modules.

Such CV to MIDI converters are still rare in Euroland and many of the existing modules have severe restrictions or instabilities. One crucial problem is that most sequencers do not output gate and pitch information exactly synchronously. Another is that you need to have high quality jitter free AD converters for precisely catching your pitch CVs.

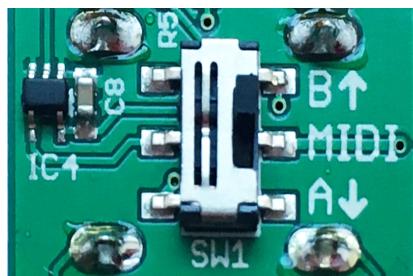
The X7 aims to be the most precise, comprehensive and

flexible CV → MIDI converter available and we are confident that it indeed is. It supports an unlimited number of voices (even if your master just has eight CV inputs, more voices can be created internally with all your **sequencer**, **algoquencer**, **chords**, **arpeggio**, **minifonion** and other circuits). Also it gives you access to almost every conceivable MIDI feature. And it benefits from the master's super precise and stable AD converters.

So let's get started with the hardware. Just as with MIDI IN, you can choose between USB and TRS. But here there is a difference. The problem arises from the fact that the mapping of the MIDI DIN plug to 3.5 mm stereo jacks has been - well - fucked up by the hardware vendors. Some have chosen the tip of the plug to be the TX signal, others have found the ring to be more suitable. So two incompatible "standards" haven arisen, which were later called MIDI "type A" and MIDI "type B".

While at the input there is an autosensing, at the output side this is not possible. So this time you need to get it right. For that reason on the back side of the X7 there is a small switch where you can select either type A or type B for your TRS output. If you are unsure which one is the correct one for your specific device, simply try both.

Note: For our shipped adapters set the switch in position B!



Using the CV → MIDI feature of the X7 is easy. Use the

circuit **midfout** (see page 190) for that purpose. Here is an example for a monophonic patch with just one voice. The pitch input is read from **I1**, the gate from **I2**:

```
[midfout]
pitch = I1
gate = I2
```

Per default, X7 sends on MIDI channel 1 on TRS. You can change both with the parameters **usb** and **channel**:

```
[midfout]
usb = 1
channel = 7
pitch = I1
gate = I2
```

To create a polyphonic patch simply add more pitch/gate pairs:

```
[midfout]
pitch1 = I1
pitch2 = I2
pitch3 = I3
gate1 = I5
gate2 = I6
gate3 = I7
```

Of course you can use internally generated or shaped pitch information, as well. In this example the pitch input from **I1** is quantized to C minor before sending it to MIDI (see page 200 for details on the **minifonion** circuit):

```
[minifonion]
input = I1
degree = 7
```

```
output = _PITCH
```

```
[midfout]
pitch = _PITCH
gate = I2
```

You can even create a MIDI to MIDI quantizer - without any further eurorack module:

```
[midfin]
pitch = _INPITCH
gate = _GATE
```

```
[minifonion]
input = _INPITCH
degree = 7
output = _OUTPITCH
```

```
[midfout]
pitch = _OUTPITCH
gate = _GATE
```

Of course you can also access all the CCs and other controllers, such as velocity, aftertouch, and polyphonic key pressure. Also you can send your modular clock and reset signals via MIDI. Please see page 190 for all details on the **midfout** circuit.

And by the way: as always, all parameters are CV controllable and can be changed on the fly - even things like **channel** and **usb**.

I think you can guess the flexibility of this approach!

7.6 MIDI through

The X7 can forward MIDI data, that are incoming via TRS or USB, to one of its two outputs (TRS / USB), while still

being able to “feed in” additional events into the same output (using `midout` (see page 190)) or processing the events (using `midin` (see page 183)).

Use the `midithrough` (see page 199) circuit for forwarding data from an input to an output. Here is an example:

```
[midithrough]
fromusb = 1
tousb = 0 # means TRS jack for output
```

This will forward MIDI events from the USB port to the TRS output. Note: All `midin` and `midout` circuits still work, so the output stream on the TRS jack will both contain the original events from MIDI-USB and the events you create with your `midout` circuits.

`midithrough` cannot do any filter or processing on the fly. But if it would become an issue, we might add useful feature here in future.

7.7 Four gate outputs

The X7 has four gate outputs. These are easy to use and also not very thrilling. But useful. Each of these can output modular level triggers or gates of 5 V.

For using the gates, refer to them as `G9`, `G10`, `G11` and `G12`. Why not starting at `G1`? Well, the gates `G1` ... `G8` are reserved for the first G8 expander (see page 47), even you don't use one. Note: the gates on the X7 are only outputs, whereas the G8 can also use them as inputs.

Of course you can use the gates in combination with MIDI. Here is an example for outputting three different MIDI clocks as well as a reset signal at the gates:

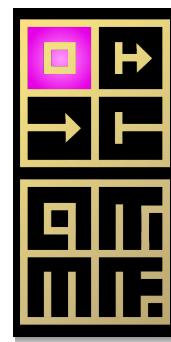
```
[midin]
clock = G9 # 16th notes
clock8 = G10 # 8th notes
clock4 = G11 # quarter notes
start = G12 # trigger at MIDI start message
```

7.8 Eight multi color LEDs

Just as with the master and the G8, you can override the functions of the eight LEDs on the X7 with your own choice of colors. Use the registers `R49` through `R56` for that purpose.

Here is an example for changing the LED color with a pot:

```
[p2b8]
[copy]
input = P1.1
output = R49
```



7.9 Fast patch upload via Sysex

MIDI defines a type of event that is called “Sysex”, which is an abbreviation for “MIDI System Exclusive Message”.

These are portions of data bytes that just have a meaning to certain types of devices and are not standardized by MIDI. These messages can mean *anything* to a device. In fact one of the original ideas was to load “patches” to and from a hardware synth.

And exactly that original application is implemented by the X7: You can upload `DROID` patches to your master via MIDI sysex. Why would you do that, if you could simply use “USB stick mode”? Well, there are a couple of advantages:

- The upload via sysex is really super fast.
- Your `DROID` does not stop playing music for more than a fraction of a second.
- You don't need to touch the switch nor the button of the master. So it's a complete *remote control*.
- You don't need to do this cumbersome “eject” of the USB drive.

If you use the Forge, using Sysex works just out of the box. Put the X7 switch to the right. Let it there. At any time you can upload your current patch just by clicking the *Activate!* icon in the toolbar!

If you don't use the Forge, it's a bit more complicated to setup, since you need a software for sending patches via Sysex. But if anything goes wrong you can always fall back to USB stick mode.

Patch upload via sysex on Linux

The best way to setup the patch upload via sysex depends on which operating system you use. Let's start with Linux, just because it's the easiest. On any decent regular Linux installation there usually is a tool called `amidi`. It's part of the sound driver (ALSA), so it's usually already installed. `amidi` can send any MIDI commands in-

cluding sysex.

Now in the Firmware ZIP-file that you find for download on your shop, you find the directory **utilities/sysex/linux** and in there the script **droidpatch**. Copy that script to **/usr/local/bin** and make sure it is executable.

Now you can upload a patch file by calling **droidpatch** with the name of your patch file. It needn't be called **droid.ini**:

```
user:~ $ droidpatch mypatch.ini
```

Of course the switch on the X7 needs be on the right (MIDI). That's it.

Patch upload via sysex on Mac

Now let's look at the Mac. It's basically the same procedure as on Linux just with one change. Mac does not have **amidi**. Instead you need another tool for doing MIDI on the command line. I recommend to use **sendmidi**. This has several advantages over more complex software suites:

- It is small.
- It is free.
- It is command line based and thus good for automating things.

You can get **sendmidi** here: <https://github.com/gbevin/SendMIDI/releases>. Choose your operating system and download and unpack it. Basically there is no installation necessary since this tool really just consists of one single file, which is called **sendmidi**. I suggest that you copy that file to **/usr/local/bin**, so that it is always ready for you to use.

Just as with Linux, in the Firmware ZIP-file you find the directory **utilities/sysex/mac** and in there the script **droidpatch**. Copy that script to **/usr/local/bin** and make sure it is executable. Put the X7 switch to the right and you can send patches with the new command **droidpatch**:

```
user:~ $ droidpatch mypatch.ini
```

One side note: **sendmidi** on Mac sometimes has a problem that every 256th byte is lost. The problem seems to lie deep in the API of Mac itself. If you run into that problem, you can try to enter a space into your patch file at the right position. Or you might consider using the Droid Forge instead of the command line.

Patch upload via sysex on Windows

Just as with Mac, the first step is to install **sendmidi**. You can get it here: <https://github.com/gbevin/SendMIDI/releases>. There is no real "installation". Just take the program **sendmidi.exe** and copy that to the directory where you keep your DROID patches. If you have none, it's a good time to create one now.

Open a terminal window, go to the directory with **cd** and try it out by simply calling that program. It should output a version number:

```
C:\Users\dmmdm\patches> sendmidi  
sendmidi v1.0.15  
https://github.com/gbevin/SendMIDI
```

Usage: sendmidi [commands] [programfile]...

Now connect your X7 with USB to your computer. And put the X7's switch to the right. Then check if **sendmidi** detects the X7, by adding the word **list**:

```
C:\Users\dmmdm\patches> sendmidi list  
Microsoft GS Wavetable Synth  
DROID X7 MIDI
```

Here it is! Now for every subsequent call to **sendmidi add dev x7** in order to select the X7 as output devices.

Now let's try the MIDI connection by sending a note event. This small tool is really cool. In fact you can send all sorts of MIDI events. You can even create sequences with lots of notes events and pauses in between. It's kind of really low level MIDI sequencing. So let's play a C2 at full velocity (value 127):

```
C:\patches> sendmidi dev x7 on c2 127
```

If everything goes well, you should see the LED 2 on the X7 shortly flash green:



If this works, you know that the USB-MIDI connection is working and **sendmidi** is also ready. The next step is to convert your **DROID** patches into MIDI sysex files. To do this you just need to add a sequence of five specific bytes at the beginning, then add the patch and one final special byte at the end.

With the X7 software releases there are the files **sysexhead.txt** and **sysextail.txt** in the subdirectory **utilities/sysex/windows**. These need to be glued to the beginning and the tail of the patch in order to form a MIDI sysex file. I recommand that you copy them to your patch directory.

Note: For this all to work it is very important that your patch files don't contain non-ascii characters. So don't use German umlauts or any other special character that's not part of the English language (you would do that just in comments anyway).

On the command line you can use the command **copy** for gluing together the head, the patch and the tail. Use a plus sign between the file names like this:

```
C:patches> copy sysexhead.txt + yourpatch.ini
+ sysextail.txt yourpatch.syx
```

Write this in one line. This will convert **yourpatch.ini** into a new file called **yourpatch.syx**. That file can easily be sent via **sendmidi**:

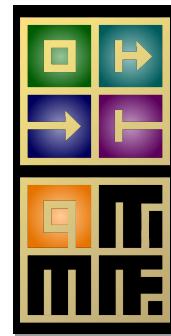
```
C:patches> sendmidi dev x7 syf yourpatch.syx
```

That's all! Your master should now load the patch, show a very short restart animation and your patch is up and running.

7.10 Software update for the X7

Other than the simple expanders like the P2B8 or the P10, the X7 has a rather sophisticated software. Some bugs might be found. And new feature ideas will be implemented. So The X7 has a software update procedure.

When you start the X7, it shows its current sofware version in the 2x2 LED field of the gates. The first released version is called orange-9 and is indicated by the G9 LED shining orange:



In order to make things as easy as possible for you, the software update for the X7 is done by the master. You don't need to change anything in your cabling for that. Leave the X7 attached as the first expander on the master.

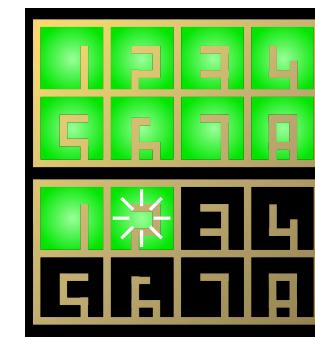
Here are the steps for an X7 firmware upgrade:

1. Copy the firmware file for the X7 (from Discord or from our Download page) to the SD card in the master.
2. Rename it to exactly **x7.fw**
3. Bring the master into the maintenance mode (see page [70](#) for details). Long things short: this is done by a very long button press.
4. Your maintenance menu should show a *green* menu item at position 8 (if not, the SD card or the file **x7.fw** on it is missing):

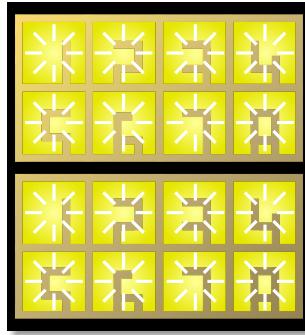


5. Now press the button a couple of times until the blinking cursor is at position 8.
6. Press the button longer in order to start the update procedure.

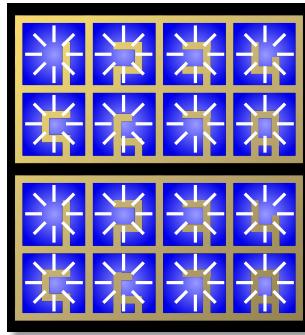
If everything goes well, you see a kind of progress bar running through all 16 master LEDs, while the X7 does the same kind of animation with its 8 LEDs.



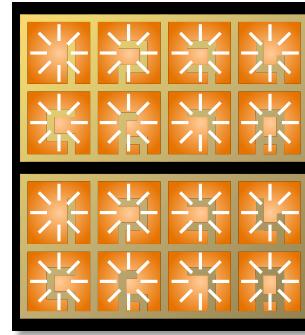
In case of an error, all 16 LEDs blink in one color. If all LEDs blink **yellow**, the firmware file is missing (which is strange, because it was there at the beginning):



All blinking blue means an invalid size of the firmware file:



And orange means that the file could not be read from the SD card:



After the upgrade, you need to leave the maintenance menu on your master. Do this by navigating the blinking cursor to the white LED 1 and press the button a bit longer:



7.11 Some technical details

Are you interested in the technical issues of the X7? Here are some details.

The X7 uses the same micro controller (MCU) as the DROID master: The STM32F446RET6. It is running at 180 MHz and has a 32-bit hardware floating point unit. It's a very powerful processor and hard to get these days (chip crisis). But it's worth it for short latencies and high data rates.

The communication between the master and the X7 is running at a much higher bit rate than is used for the controller communication. It's using 1 MBit/sec, whereas the controller bus is running just at about 50 Kbit/sec. This is the reason why the X7 needs to be attached as first module directly to the master. This higher bitrate allows for transferring MIDI data with low latency - while the controllers are still being processed at the same speed as without the X7.

When you switch to "USB stick mode" (switch to the left), the bit rate is even increased to 2 MBit/sec in order to make the access to your micro SD card as fast as possible.

The auto sensing of the MIDI TRS input is done with a bridge rectifier, four diodes, so the polarity of the input is ignored.

8 The R2M/R2C controller bridge

8.1 Introduction

The R2M/R2C is a pair of two 2 HP modules that allow you to connect a chain of controllers to your master through a standard 3.5 mm stereo cable (sometimes also called aux-cable). The usual idea is that you put all your Droid controllers into a skiff case and mount your master, X7 and G8 into another case, together with all your fancy Eurorack sound modules.

While you could do this with the typical 6-pin ribbon connector (e.g. the 80 cm version that we offer), using the R2M/R2C combination has some serious advantages:

The connection cable can be almost arbitrary long (20 m have been tested and works perfectly). Since the connection is done on the front of the modules, you can quickly disconnect your skiff for the purpose of travelling to a gig. You can use a standard 3.5 mm stereo TRS cable for the connection. These modules are not just passive connectors but contain special driver ICs that transform the electronic voltage levels, which run in the 6-pin ribbon, to something more stable and reliable that is fit for longer distances in a more hostile environment.

The controllers do not receive their power from the master but from the R2C module, which has a power connector and a voltage regulator for that purpose. Each chain of the R2C module provides the same power to its controller chain as the master does (it contains the identical voltage regulator). That means that you can connect up to 32 controllers (!) to one R2C.

Another nice thing: The R2M/R2C combination allows for two of these master / controller connections in parallel. That means that you can have two masters being

attached to their individual controller chains. That does not mean, that each of the masters can access each of the controllers at the same time, however. Both master / controller connections work completely separately.

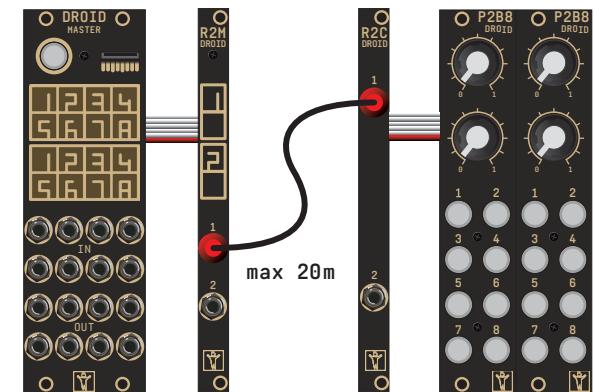
8.2 Setup with one master

First let's assume that you have just one master. On the back of the R2M (M stands for "master") you will find two 6-pin shrouded connectors. These are labelled 1 and 2. Connect connector number 1 with the 6-pin ribbon cable to that output of the master that is usually used for the Droid controllers.

Mount the R2M next to your master. Mount the R2C (C stands for "controller") into your skiff and use the shipped 10-pin power cable for powering it with Eurorack power (red stripe down). Otherwise the controllers won't work. The R2C has two 6-pin connectors on the back, as well. Connect the first controller of your chain to the connector labelled 1.

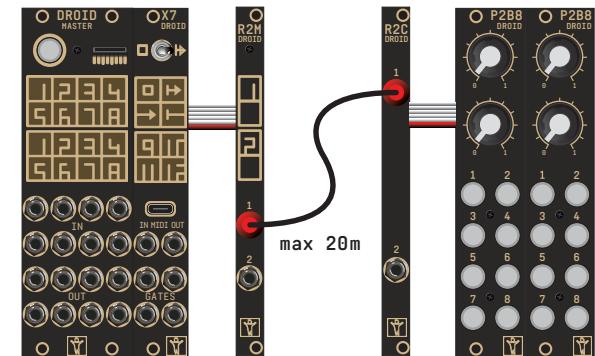
Now plug one of the shipped 3.5 mm stereo aux cables to jack 1 of the R2M to jack 1 of the R2C. Or use your own 3.5 mm stereo cable for that purpose.

You don't need any changes in your Droid patch.



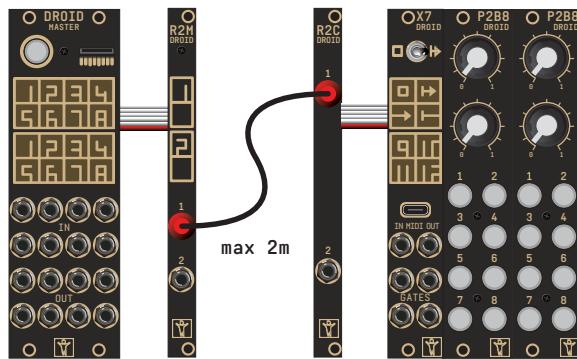
8.3 X7 connected to the master

If you have an X7, connect the R2M to the X7, so that the order is master / X7 / R2M. Mount the X7 next to the master. Connect the R2M to the controller output of the X7.



8.4 X7 in the skiff

You can move the X7 to the “other side” of the connection by connecting the R2M directly to the master and using the X7 as the first module after the R2C. If you do this, the maximum distance that you can bridge is smaller, but 2 m should always be possible. This should be sufficient for almost any case.



8.5 Controllers before the R2M/C bridge

You can put the R2M/C bridge at any position in your controller chain that you like. So it's possible to have some controllers directly connected to the master. Simply wire

the last of these to the R2M.



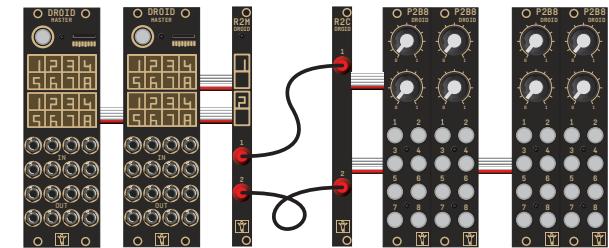
8.6 More than one bridge

If you have lots of controllers and put them in two skiffs, you can even use two R2M/C bridges and put a second bridge somewhere later in the chain of controllers.



8.7 Setup with two masters

As states above, the R2C/M is dual channel. You can create a second master / controller bridge with the same pair of R2 modules. Connect the second master to connector 2 of the R2M and its controllers to connector 2 of the R2C. Note: both master / controller chains are separated and cannot interact with each other.



9 Droid under the hood

9.1 How the module's state is saved

If you ask people what's the number one annoyance when using a module, most will answer this: When a module is losing its state when you power cycle your modular. That's also the number one reason for people running their system the whole night through.

Therefore the **DROID** - of course - will save it's state always automatically. But what do I mean with "state" in the first place? It's very simple: If you have defined a **button**, **DROID** remembers whether it is currently *on* or *off*. If it is *on now*, so will it be after a power cycle of your system or a restart of the module (the same holds for *off*, of course).

Other circuits have states as well, for example the **algoquencer** (state of the step buttons, the accents, the pattern length), the **matrixmixer** (state of all matrix buttons), the **calibrator** (state of the calibration adaption) and so on.

Only the result of manual interaction is saved, not for example the contents of the **cvelooper** or the current phase of an **lfo**.

All these states are saved to the micro SD card into a file with the name **DROIDSTA.BIN**. This file is created with a fixed size of 128 KB when your **DROID** starts. All manual changes to your circuits are saved there after a short delay of about 1.5 seconds. Also when you press the button for loading a new patch, the states are saved immediately, even if the last change was less than 1.5 seconds ago.

This has the following implications:

- When no memory card is in the **DROID**, no states will be saved. But you can always put one there even if the module is already running for some time. It will be detected automatically and all states will be saved after a second or two.
- When you move the SD card from one **DROID** to another, the current circuit states will also be moved.
- If you want to erase all your settings, you can do this by starting the **DROID** without an SD card and inserting it later. The settings file will only be loaded right at the beginning. If it's not present, all circuits start with their default settings.

The format of the file is binary and looks chaotic. You cannot open or edit it with any software. But the format is very efficient, so the ongoing saving of states doesn't have any impact on the precise timing or performance of the **DROID**.

Note: If you forget to have the SD card inserted when you power up your **DROID**, it will run with default states. Inserting the SD card afterwards will **not** load the saved settings but the other way round! It will save the current states on the card. This way you **lose your original settings**. So if you have forgotten to start with the card, **power off the module, then insert the card, then power it on again**. That way you won't lose your settings.

You might ask what happens if you change a patch? The state of the circuits of the previous patch was saved to the SD card. How can that saved state be loaded into a new patch that might have a different structure?

The rule is this: Droid numbers all circuits *of the same type*, starting from 1 - according to their appearance in the patch. So there is button 1, button 2, etc. And there is buttongroup 1, buttongroup 2 and so on. When you press a button, **DROID** writes to the SD card something like "This is the new state of button 2."

When that state is loaded later into a new patch, the mapping of the loaded states to the circuits uses that same numbering. So the saved state of button 2 is loaded as start state for the second button in the new patch.

From this follows that:

- If your new patch has less buttons than your previous one, some of the saved states are ignored, since the matching buttons don't exist anymore.
- If your new patch has more buttons than your previous one, the exceeding buttons start in the default state.
- If you change the order of the circuits in your patch, circuits will get the "wrong" states when you first start it.

Note: There is only one state file on the SD card. If you swap patches back and forth, you will always mix up your state if the patches have different structures. You might want to get a separate SD card for every patch, if swapping and not losing your state is crucial.

Sometimes you don't want a circuit to save its state. You want a fresh start every time you start your **DROID**. Or you missused a circuit that's meant for manual operation (e.g. **nudge** (see page [234](#)) for some automatic changes

that happen very frequent and you don't want to flood your SD card with new useless states.

All circuits that save states have an input **dontsave**. Set this to **1** to prevent the state from being saved (and loaded):

```
[nudge]
dontsave = 1 # prevent loading/saving
...
```

9.2 The order of the circuits

You might ask yourself what role the *order* of the circuits plays in your patch file. Well - in most cases it doesn't matter at all, in some cases, however, it might cause very subtle timing differences in the range of a couple of hundred μs . In order to understand this, we need to have a closer look at how the DROID works:

The basic working process of your DROID is a simple *loop* that is repeating over and over again - at a speed of approximately 180 μs per cycle, which means that it is running at approximately 5.5 kHz! In each cycle of the loop the following things happen:

- The current values of all inputs, gates, buttons and pots are read in and stored in the **I**, **G**, **B** and **P** registers.
- Each circuit creates a new value for each of its outputs. That might include writing new values into **0**, **G**, **L** or **R** registers.
- The contents of the **0** and **G** registers are converted

into voltages for their respective output jacks. The contents of the **L** and **R** registers are translated into brightness and color of the according LEDs.

Now let's look at two circuits that are internally wired:

```
[bernoulli]
input      = G1
distribution = P1.1
output1    = _TRIGGER
```

```
[contour]
trigger    = _TRIGGER
output     = 01
```

Here an external trigger at **G1** (on the G8 expander) is being used to trigger an envelope randomly, which is then sent to **01**. Here - because of the order of the circuits - the envelope will start *in the same loop cycle* in which the trigger is seen at **G1**.

Now let's change the order:

```
[contour]
trigger    = _TRIGGER
output     = 01
```

```
[bernoulli]
input      = G1
distribution = P1.1
output1    = _TRIGGER
```

Now it is different. In the cycle in that the trigger is detected at **G1**, the envelope has already been processed. It gets its trigger through the internal wire **_TRIGGER** not before the next cycle. This introduces a short delay of up to 160 μs . This is not very long, but it can easily be avoided.

Note: However, when your patch contains quite a lot of circuits, the loop time gets longer. Even then, it is likely to stay below 500 μs .

9.3 Displaying the value of a register

In the section about finding errors in your patches we already talked about the *status dump file* (see page 65). That shows you the exact value of every single input, output, potentiometer and other register.

But there is another way of showing a current value from within your patch, and that *live*. This can be useful, for example, if you want to spare a potentiometers and use a fixed value instead but first need to find out which value fits best. Maybe you need a simple envelope with a fixed

non-zero attack value. You could try out different values by changing your patch over and over again. But that's quite annoying.

Here the experimental **X1** register helps. It's an output

register. When you send a value there, all the LEDs of the front panel will show that value in a way similar to the line-error-encoding of the patch parser. Here is an example:

```
[p2b8]
```

```
[contour]
```

```
attack = P1.1  
release = P1.2  
trigger = B1.1  
output = 01
```

```
[copy]
```

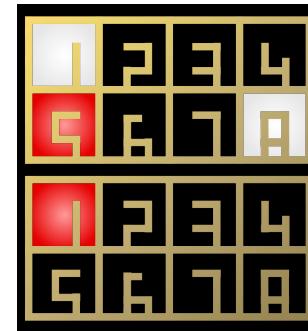
```
input = P1.1  
output = X1
```

Now turn the knob **P1.1** for setting some nice attack value. As soon as you remove that from its zero-position, all LEDs will move around in red and white and show the current value of **P1.1** with three digits. Input LEDs are lit white and red. White digits account for 0.1 and red digits for 0.01. The red digits at the outputs account for 0.001. Here are some examples:

The value 0.148:



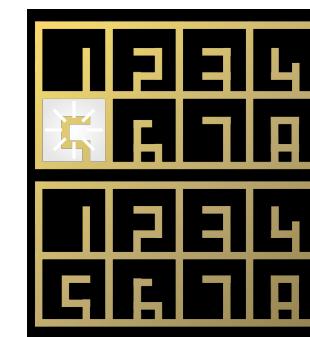
The digit 9 will be displayed as 8 + 1. So here is 0.951:



A zero digit means of course that no LED is lit in the according section. Here is 0.950:



But what if digits in the input section collided? E.g. 0.550 would need the LED of input 5 to be red and white at the same time. Well, then it will blink between white and red:



The upper scheme just works for numbers in the range 0.0 ... 1.0. But there are different color schemes for the non-white LED that enable showing other ranges:

- Numbers in the range -1.0 ... 0.0 (excluding zero) are shown with blue LEDs.
- Integer numbers in the range 2 ... 1000 are shown in orange color, with factor of 1000 applied.
- Integer numbers in the range -1000 ... -2 are shown

in cyan color, with factor of -1000 applied.

Example: this is the pattern for the number 148:



Once you have found a nice value, simply replace **P1.1** with that fixed value and your pot is free for something else!

Note: When you send 0 to the **X1** register, it will be inactive and the LEDs behave like normal and show the actual values of your inputs and outputs.

9.4 Displaying current values

There is an easy method for getting the current value of all registers! Simply *double press* the master's button - just similar to a mouse double click. If you do this, all LEDs will flash white once. And on the SD card a file with the name **STATES1.TXT** is being created. This file will not only show you the current value of all registers but also the values of all internal patch cable (see page [30](#)).

When you do this again, a **STATES2.TXT** and so on is created. When **STATES99.TXT** is reached, it starts over again from **STATES1.TXT**. When you create the first dump file after the **DROID** has started, all old files from the previous run are automatically deleted.

Here is what such a file looks like:

DROID status

Firmware version: blue-1
Running since: 34.576 sec
Free RAM: 110579 Bytes (97.857%)
Size of patch: 1333 Bytes (2.082%)

Inputs:

I1: 0.3201	I2: 0.8210	I3: 0.0000	I4: 0.0000
I5: 0.0000	I6: 0.0000	I7: 0.0000	I8: 0.0000

Normalizations:

N1: 0.0000	N2: 0.0000	N3: 0.0000	N4: 0.0000
N5: 0.0000	N6: 0.0000	N7: 0.0000	N8: 0.0000

Outputs:

O1: 1.0000	O2: 0.2000	O3: 0.3333	O4: 0.0000
O5: 0.0000	O6: 0.0000	O7: 0.0000	O8: 0.0000

G8#1 Gates:

G1.1: 1	G1.2: 0	G1.3: 0	G1.4: 1
G1.5: 0	G1.6: 0	G1.7: 0	G1.8: 0

RGB-LEDs:

R1: 0.000	R2: 0.000	R3: 0.000	R4: 0.000
R5: 0.000	R6: 0.000	R7: 0.000	R8: 0.000
R9: 0.000	R10: 0.000	R11: 0.000	R12: 0.000
R13: 0.000	R14: 0.000	R15: 0.000	R16: 0.000

Controller 1 [p2b8]:

B1.1: 0	B1.2: 0	B1.3: 0	B1.4: 0
B1.5: 0	B1.6: 0	B1.7: 0	B1.8: 1
L1.1: 0.000	L1.2: 0.000	L1.3: 0.000	L1.4: 0.000
L1.5: 0.000	L1.6: 0.000	L1.7: 0.000	L1.8: 0.000
P1.1: 0.77631	P1.2: 1.00000		

Internal patch cables:

_CLOCK:	1.00000
_PITCH:	0.23430
_RELEASE:	0.30000

9.5 Controller latency

As stated above, you can attach up to 16 controllers to one **DROID** master. These controllers are connected via a ribbon cable with six wires. Four of these wires comprise a power supply for the controllers with 5 V (except for the *M4 - Motor Fader Unit*, which has its own power supply). The remaining two wires form a digital serial connection between the modules. The master sends data to the first controller, the first controller to the second and so on until the last controller sends all collected data back to the

master.

This serial line sends approximately 7200 bytes per second. Every controller needs a different number of bytes per update and for the P2B8 it's 11 bytes. So if you have just one P2B8, you get $\frac{7200}{11} = 654$ updates per second. That's roughly one update per 1.5 ms - which is pretty fast. That means that a button press is registered by the master after 1.5 ms plus some internal computation time.

If you have the maximum of 16 controllers (which would be 80 HP of controllers), things slow down a bit, of course, since now every controller gets just $\frac{1}{16}$ of the data in the serial connection. In that case a button press would need about 25 ms to be registered. This is still way fast enough for the typical switching tasks that you typically do with the **DROID**. However, playing live drums with the buttons would not be very tight (I wouldn't suggest that anyway).

10 Firmware upgrade

10.1 What version do you have?

DROID is an active project, new features are being added, bugs are being fixed. Also new controller modules require changes in the software of the master module. All these things are reasons why, from time to time, we release a new firmware (software) version for the DROID master.

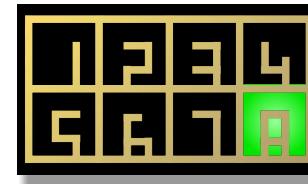
If you want to use the new features or have the bugs fixed, you can update your firmware. You find the newest release always on our [download page](#) and also in our [Discord community](#).

Unless most other software, DROID uses a combination of a color and a number in order to name a software version. For example the version this manual is written for is called **blue-3**.

Note: Some of the expanders and controllers also have firmwares that you can update. Please see page [57](#) for the X7 and page [46](#) for the M4.

When your master starts you can see your current version in a short LED animation. Look at the first two rows of LEDs (which normally show the inputs) and their numbers from 1 to 8. One or more of them will light up in a color. Read these as a number and add the color and you have the firmware version. The other two lines show a rainbow animation and are not important.

This is how the version green-8 is being shown:



If two numbers light up, don't add them but read them as a number, for example this is blue-13 (not 4!):

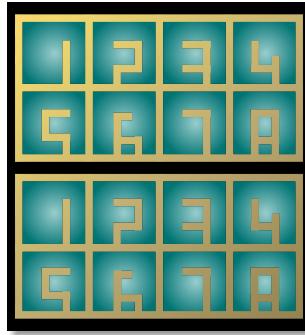


10.2 Normal update procedure

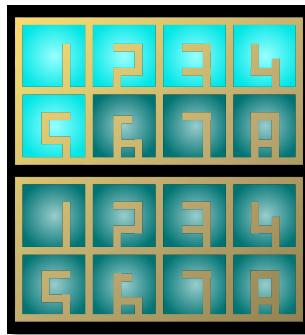
Here is how you upgrade the firmware of your DROID:

1. Download the most current firmware file from the DROID's homepage at <https://shop.dermannmitdermaschine.de/droid>.
2. Copy that file to your micro SD card **and rename it to droid.fw**.
3. Insert that micro SD card into your DROID and press the button, or power your DROID on while the SD card is inserted.

Now if everything is well, the 16 LEDs show a dark cyan color:



Now your DROID reads the contents of the file **droid.fw** and burns it into the internal flash memory. While this is going on the LEDs change their color one by one into bright cyan:



If everything goes well then at the end all LEDs flash a couple of times and the DROID starts into normal mode. Here are some things that could possibly go wrong:

Missing firmware file

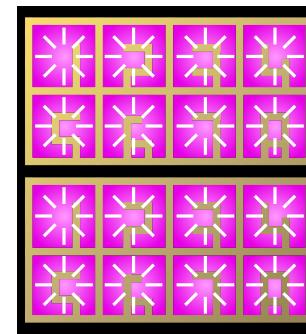
If you have not copied the file **droid.fw** or misspelled it or it cannot be found for some other reason like a defunct SD card then simply nothing happens. The DROID starts

like usual.

Invalid firmware file

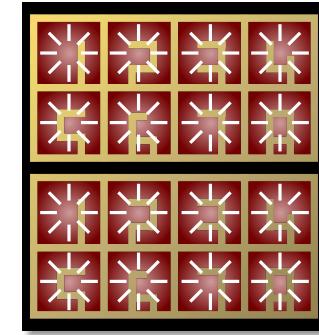
A magenta blink code means that your firmware file **droid.fw** is somehow not valid. It has the wrong size. This usually has one of two reasons:

- You copied to wrong file to **droid.fw**
- You try to update to a **blue** version on a DROID that currently has a **green** version. If you want to switch to blue, you need one extra step. Please see on the next page in the section *Upgrade from green to blue* for details.



Fail to program

If there is some error when programming the file into your DROID's memory, all LEDs blink dark red. Retry downloading and upgrading the firmware again!



Firmware already up-to-date

If the firmware in the file **droid.fw** already has been flashed successfully in a previous update, nothing happens. The DROID automatically detects this and skips the update. So it is save to leave the SD card with **droid.fw** in the SD card slot.

10.3 Upgrade from green to blue

After the firmware version **green-8** there is a bigger change. So the next version is not green-9 but **blue-1**. The main difference is that **blue** firmwares are larger and allow for more cool circuits and other stuff in your **DROID**.

In order to make that possible we needed to change the firmware format. For that reason - if your **DROID** has a green firmware installed - you need to update your **bootloader** first. The bootloader is that part of the software that does the actual firmware upgrade. If your master came already shipped with a blue firmware, everything is fine and you can stop reading here.

With the bootloader from the green firmware you will get all LEDs flashing magenta if you want to update to **blue-2** (or any other blue firmware). So in this case you need to do the following steps:

1. Update to **green-8**. This is important since only this firmware has a menu entry for updating the bootloader.
2. Use the maintenance menu to update the bootloader. After which you are on green-8.
3. Update to **blue-2** or any other blue firmware just as described on the pages before.

Here is how step 2 works in detail. Do the following steps for this:

First make sure that you have the firmware file of **green-8** on your SD card. This is probably the case anyway if you just updated to green-8. Now press the button long in order to enter the maintenance menu (see page [70](#) for details).

If everything goes well, LED 7 must show a new **blue** menu entry:



If the blue menu entry does **not** appear, it's for one of the following reasons:

- The file **droid.fw** does not match the firmware that is currently running (update your firmware

first)

- Your bootloader is already up-to-date (identical with the one in **droid.fw**).
- The file **droid.fw** is missing on the card.
- The file **droid.fw** is damaged.
- The file **droid.fw** cannot be read from the card (try reformatting the card with a FAT filesystem in that case).
- The SD card is not readable.
- No SD card is present.

Now use short button presses in order to move the blinking cursor to LED 7. There press the button long. This will start the update. A blue LED will run one cycle around, the **DROID** will restart and you are done. This whole thing should last just a few seconds.

IMPORTANT: Do not switch off your DROID until the procedure is finished!!! Doing so will make it completely unusable. It has to be reprogrammed in our labs if that happens.

If you enter the maintenance menu again, the menu item 7 should have disappeared, since your bootloader is now up-to-date.

If you need any help, please post a question on our [Discord community](#).

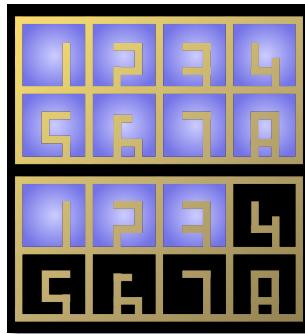
11 Calibration, factory reset and other maintenance stuff

11.1 The maintenance mode

The DROID has a special mode for various maintenance tasks. This mode is a bit “hidden” so that you do not enter it accidentally. You enter the maintenance mode by holding the button on the master for a couple of seconds. After 1.5 seconds of holding the button, an animation of light blue LEDs going from O8 over to I1 starts:



When the blue LEDs reach I1, continue holding the button. DROID restarts. Still hold the button. Now the animation of the blue LEDs starts in the opposite direction:



When the end is reached - this time at O8 - and you now release the button, the DROID enters the maintenance mode. If you let go the button before this you go back into normal operation.

In maintenance mode you will see a white “cursor” blinking at the LED for I1. Cell I3 is red, Cell I4 is magenta:



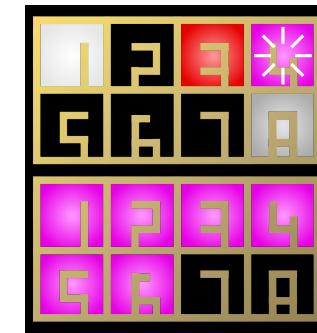
The four positions I1 ... I4 represent four different menu options:

1. **WHITE (I1)**: leave the maintenance mode and restart the DROID.
2. black: currently unused.
3. **RED (I3)**: reset the DROID to factory mode (but keep calibration).
4. **MAGENTA (I4)**: start the procedure of calibrating the voltage of the eight outputs.

A short press of the button moves the cursor to the next cell. Pressing three times brings you to cell 4:



A long press of the button selects the item the cursor is currently at. It starts an animation on the LEDs of O1 ... O8 in the same color as the selected item (in this case calibration mode):



When the animation reaches O8, the item is being selected.

11.2 Factory reset

The factory reset can help in situations where - due to some software problem, maybe in a beta or testing version - the **DROID** is stuck and does not want to run again. The problem might be triggered by the current saved states of the circuits or by the currently loaded patch.

You do a factory reset in the maintenance menu by selecting position I3 (red).



All circuit states are erased. Also the current patch is erased from the internal flash memory of the master.

Note: If the patch is still on the SD card, it will immediately be reloaded after the reset, so if you want to avoid this, put either a different patch on the card or remove the card while doing the factory reset.

The calibration of the voltages of the outputs is *not* lost, when you make a factory reset!

11.3 Calibration of the outputs

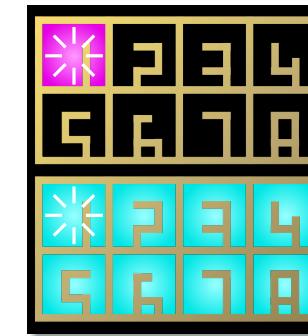
The **DROID** comes with 8 high precision DA converters (DACs) that produce highly accurate voltages for the output jacks. These need to be calibrated in order to match their designed precision. Calibration of the DACs is done in our labs before we ship the units to you.

There is a super tiny chance that your calibration gets lost: When you switch off your rack just in that fraction of a second when you load a new patch by pressing the button and at the same time deleting the calibration backup file on your SD card! However unlikely: if your **DROID** does not start with its usual rainbow animation but with a white LED animation, your DACs are not calibrated and not very precise anymore. In that case do as described here.

Otherwise you probably never will need to calibrate your outputs. If you want to do so anyway, please make sure that your **DROID** has warmed up before you start. That gives the best precision. Calibration is easy and you just need a patch cable. As a preparation unplug all jacks before you start.



Now enter maintenance mode and select cell number 4 (magenta):



After entering the calibration mode, the top 8 LEDs are black and the bottom 8 LEDs are cyan - with the exception of input 1 blinking magenta and output 1 blinking cyan.

Now use a patch cable and connect input 1 to output 1. **DROID** now tries out different output voltages and measures them by means of the precision ADC of input 1. This information is being used for the exact calibration. The result of the calibration is saved to the **DROID**'s internal flash memory.

As soon as channel 1 is calibrated the LED O1 changes to green. The cursor moves to the next channel:



Now proceed to the second pair of jacks and connect input 2 to output 2. Do this until all eight channels are green. **DROID** will then automatically end calibration and start normal operation.

If one of the channels will not go green in spite of having a proper connection between the relevant input and output you might have a hardware problem. Please contact us.

Hint: If you like you can use eight patch cables and patch

all eight connections at once. Then you just have to wait for a couple of seconds until everything is calibrated.

By the way: If you are looking onto your SD card, you will find a file with the name **DROIDCAL.BIN**. This is a backup of your DAC calibration. Don't touch it. Just leave it there. If you delete it, it will automatically reappear anyway. If your **DROID** loses its calibration for some reason (currently there is none I can think of...), starting the **DROID** with a card with this file will automatically restore the DAC calibration.

11.4 Using your own SD card

Formatting a micro SD card

DROID comes shipped with a micro SD card ready to use, but you can use your own card if you like. Usually when you buy a card it should work out of the box. If not, you might need to reformat it. The following filesystem types

are supported:

- FAT 12
- FAT 16

- FAT 32

Exfat is *not* supported. Also the cluster size (sector size) needs to be 512 Bytes.

Speed up cards on Mac

The Apple Mac automatically creates several files and directories on every storage device it finds, in order to support spotlight search and a trash bin. Both of which is not needed for your **DROID** and *substantially* slows down the card access when you use it with the X7.

The card that comes with your master has been prepared by us in a way that avoids these special Mac features - if your master came shipped with at least version blue-1. If you create your own card, or if yours came shipped with

an older firmware version, you can prepare it yourself.

This can be done by the following commands that you need to enter on the terminal while the card is inserted into your Mac. Hereby we assume that the name of your card is **Untitled**. If not, please adapt the commands to your name:

```
mdutil -i off /Volumes/Untitled
cd /Volumes/Untitled
```

```
rm -rf .{,_}{{fsevents,Spotlight-V*,Trashes}
mkdir .fsevents
touch .fsevents/no_log .metadata_never_index .Trashes
cd -
```

Please double check what you are typing. Especially the **rm** command is very dangerous if you are not in the right directory or have mistyped one of the dots or curly brackets!

12 Hardware

Master

Doepfer A-100 compatible "Eurorack" module with 8 HP

- STM32F446 Micro controller running at 180 MHz
- 8 CV input jacks with a voltage range from -10 V to +10 V, driven by highly accurate low jitter 16 bit AD converters
- 8 CV output jacks with a voltage range from -10 V to +10 V, driven by highly accurate low jitter 16 bit DA converters
- 16 full color LEDs
- MicroSD card reader
- Button for reloading the MicroSD card
- Expansion port for up to four G8 expanders
- Expansion port for up to 16 controllers

Power consumption:

- +12 V rail: 154 mA
- 12 V rail: 15 mA

G8 Expander

Eurorack compatible expander for the DROID master, with 4 HP

- 8 tristate gate/trigger-jacks that can each be used either as an input or an output
- 8 full color LEDs

Power consumption:

- +12 V rail: 41 mA
- 12 V rail: 0 mA

X7 Expander

Expander with USB, MIDI TRS in/out, four gates, with 4 HP

- STM32F446 Micro controller running at 180 MHz
- USB-C connector supporting USB 2.0 device mode
- Four gate outputs with 0 V or 5 V
- Switch for USB mode with three positions: SD / off / MIDI
- 8 full color LEDs
- Port for connection to the master
- Expansion port for connection to the controllers

Power consumption:

- +12 V rail: 94 mA
- 12 V rail: 0 mA

P2B8 Controller

Eurorack compatible expander for the DROID master, with 5 HP

- STM32F030 Micro controller running at 48 MHz
- 2 potentiometers
- 8 buttons with LEDs

Power consumption:

- +12 V rail: 12 mA
- 12 V rail: 0 mA

P4B2 Controller

Eurorack compatible expander for the DROID master, with 5 HP

- STM32F030 Micro controller running at 48 MHz
- 4 potentiometers
- 2 buttons with LEDs

Power consumption:

- +12 V rail: 11 mA
- 12 V rail: 0 mA

B32 Controller

Eurorack compatible expander for the DROID master, with 10 HP

- STM32F030 Micro controller running at 48 MHz
- 32 buttons with LEDs

Power consumption:

- +12 V rail: 24 mA
- 12 V rail: 0 mA

P10 Controller

Eurorack compatible expander for the DROID master, with 5 HP

- STM32F030 Micro controller running at 48 MHz
- 2 large potentiometers
- 8 small potentiometers

Power consumption:

- +12 V rail: 10 mA
- 12 V rail: 0 mA

S10 Controller

Eurorack compatible expander for the DROID master, with 5 HP

- STM32F030 Micro controller running at 48 MHz
- 2 switches with 8 positions each
- 8 small switches with 3 positions each

Power consumption:

+12 V rail: 10 mA
-12 V rail: 0 mA

M4 Controller

Eurorack compatible expander for the DROID master, with 14 HP

- STM32F030 Micro controller running at 48 MHz

- 4 alps motorized faders with a fader way of 60 mm
- 4 RGB multicolor LEDs
- 4 touch sensitive plates

Power consumption:

+12 V rail: 350 mA - 600 mA (configurable)
-12 V rail: 0 mA

13 Reference of all circuits

This is a reference of all circuits that are supported by firmware version blue-3 of DROID. The description of each circuit is made of two parts: a general introduction with some examples and a table of all input and output jacks that the circuit offers.

Just like real synth modules the input and output jacks of DROID's circuits have different characteristics, which are denoted by one of seven symbols in the reference:

	Jacks with the symbol work with <i>continuous</i> CVs in the full voltage range from -10 V to +10 V.
	This symbol denotes jacks that work on a precise "one volt per octave" base. Such outputs can be patched to the V/Oct inputs of VCOs. Inputs with this symbol expect pitch information e.g. from sequencers or musical quantizers.
	This jack has a range from 0.0 to 1.0 . Input values greater than 1.0 are truncated to 1.0 , values below zero are set to 0.0 . This input can be seen as a fraction or percentage. When you use fixed values you can write percentages, for example 55% instead of 0.55 . Since potentiometers yield values in exactly that range you can directly assign one to such a CV. If you control that CV with an external voltage, the range is 0 V ... 10 V.
	This jack is very similar to that of type , but its neutral value is in the middle position – at 0.5 or 50% or 5 V. An example is the jack distribution of the algoquencer circuit: At the middle position beats are distributed evenly in the bar. Left or right of the center the beats are more oriented to the first or second half of the bar, respectively. If you assign a pot, the center position of the pot is the neutral position. Values out of the range 0.0 ... 1.0 are truncated into that range. Hint: The input notch of the pot circuit at page 242 helps you exactly centering a pot at 0.5. The range for external voltages is 0 V ... 10 V.
	This jack operates with integer numbers such as 1, 2, 3 and so on. An example is the length input of the euklid circuit. For some jacks 0 can be allowed as well. One example is the inputoffset jack of the switch circuit. Any non-integer number will be rounded to the nearest integer. So a value of 0.6 will be interpreted as 1. Wiring an external input directly to such a jack does not make much sense, since the range 0 V ... 10 V just maps to 0 ... 1. For a 2 you would need 20 V. So you need to add some scaling, for example somejack = I1 * 10 , which converts an external 2 V to the number 2.
	This denotes a <i>stepped</i> voltage. This is one that only appears in discrete steps. An example of a stepped output CV is the pitch output of the sequencer circuit.
	Jacks with this symbol just know 0 and 1 or on and off . These are things like a gate from an envelope, where the length of the input counts. Some circuits also have switch inputs or settings of that type that enable features like "looping on". Also all inputs that are meant to be wired to buttons like B1..1 are of that type, since buttons output exactly such gate signals. Output jacks of that type always either send 0.0 (0 V) or 1.0 (10 V). Using G1 ... G8 for these is also fine, but they output 5 V instead of 10 V. When you wire an external input to such a jack, it will see a 1 at a voltage of at least 1 V and 0 otherwise.
	These jacks are trigger inputs or outputs. A trigger input just is interested about points in time where the voltage changes from 0 to some positive value above roughly 1 V. The <i>duration</i> of the time where the voltage is not zero is not interesting here. A typical use are clock or reset inputs. When the DROID outputs a trigger, is it sends a signal of 10 V for a duration of 10 ms. Using G1 ... G8 from the G8 expander for these is just fine, but the output voltage will be 5 V in that case. For external input voltages use any regular clock/trigger/gate signal from your system.

The column **Default** shows the value a parameter has if you don't patch anything into it. Here the special symbol denotes a certain "intelligent" behaviour when this jack is not used. Please read the description for details.

Memory consumption

Nothing in the world is for free. And also using circuits has a price: memory. Every circuit you use need its share of RAM. Your DROID has about 110.000 bytes of RAM free to be used by circuits. Every circuit needs a certain amount of RAM - plus some extra bytes for every used parameter.

The following table shows the RAM usage of each of the circuits:

adc	56	firefacecontrol	1080	once	24
algoquencer	872	flipflop	40	polytool	240
arpeggio	112	fold	32	pot	120
bernoulli	32	fourstatebutton	40	quantizer	40
burst	40	gatetool	56	queue	312
button	96	lfo	216	random	32
butongroup	432	logic	56	recorder	1712
calibrator	216	math	64	sample	40
chord	112	matrixmixer	168	select	24
clocktool	96	midifileplayer	6376	sequencer	168
compare	32	midiin	528	slew	48
contour	112	midiout	656	spring	56
copy	24	midithrough	208	superjust	64
crossfader	40	minifonion	88	switch	96
cvlooper	17336	mixer	40	switchedpot	88
dac	56	motoquencer	1112	timing	56
delay	1672	motorfader	104	togglebutton	48
droid	64	noop	16	transient	56
euklid	48	notchedpot	40	triggerdelay	248
explin	32	notebuttons	128		
faderbank	616	nudge	136		
fadermatrix	640	octave	32		

In addition each used input or output parameter need some memory, depending on its type:

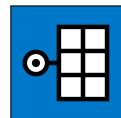
- Normal inputs need 12 bytes.
- Trigger inputs need 16 bytes.
- Tap tempo inputs need 30 bytes.
- Normal outputs need 4 bytes.
- Trigger outputs need 8 bytes.

In addition each internal patch cable and each unique constant (like **1.5** or **-12**) needs 8 bytes. Note: If you are using the *Droid Forge* for creating your patches, you don't need to do

any computations yourself. The Forge will always show you the exact memory consumption of your patch.

13.1 adc - AD Converter with 12 bits

This circuit converts an input value into a binary representation of up to 12 bits. Consider the following example:



```
[adc]
  input = I1
  bit1 = 01
  bit2 = 02
  bit3 = 03
```

In this example three bits are being used. Three bits can represent a number from 0 to 7. These are mapped to the input range from 0 to 1 (or 0 V to 10 V) in the following way:

input	bit1	bit2	bit3	bit value
$-\infty \dots 0.125$	0	0	0	0
$0.125 \dots 0.250$	0	0	1	1
$0.250 \dots 0.375$	0	1	0	2
$0.375 \dots 0.500$	0	1	1	3
$0.500 \dots 0.625$	1	0	0	4
$0.625 \dots 0.750$	1	0	1	5
$0.750 \dots 0.875$	1	1	0	6
$0.875 \dots \infty$	1	1	1	7

Values lower than 0 are treated as 0. Values higher than 1 are treated as one.

In other words: this circuit will convert an analog input value into three different gate outputs.

The expected range of the input value is from 0 to 1 per default, but you can change that with the parameters **minimum** and **maximum**. For example you could have just the range of 0.1 to 0.5 mapped to the three bits:

```
[adc]
  input = I1
  minimum = 0.1 # 1V
  maximum = 0.5 # 4V
  bit1 = 01
  bit2 = 02
  bit3 = 03
```

Now the table looks like this:

input	bit1	bit2	bit3	bit value
$-\infty \dots 0.15$	0	0	0	0
$0.15 \dots 0.20$	0	0	1	1
$0.20 \dots 0.25$	0	1	0	2
$0.25 \dots 0.30$	0	1	1	3
$0.30 \dots 0.35$	1	0	0	4
$0.35 \dots 0.40$	1	0	1	5
$0.40 \dots 0.45$	1	1	0	6
$0.45 \dots \infty$	1	1	1	7

If you use more of the **bit**-outputs you get more resolution. For example if you use **bit1**...**bit8**, the total range will be divided into 256 equal pieces. Since bit 1 is the most significant bit, adding more and more bits will not change the way bit 1 is behaving.

The applications of this circuit are various and often surprising. For example using different LFO wave forms as inputs (other than square) and you will get slower and faster gate patterns.

Please also have a look at the circuit **dac** (see page 136, which does the exact opposite!

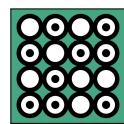
Input	Type	Default	Description
input		0.0	Input signal to convert to binary representation.
minimum		0.0	The lowest assumed input value. This value and all lower values will be converted to the bit sequence 000000000000 .
maximum		1.0	The highest assumed input value. This value and all higher values will be converted to the bit sequence 111111111111 .

Output	Type	Description
bit1 ... bit12		The 12 bit outputs. bit1 is the MSB - the most significant bit. The LSB (least significant bit) is the highest output that you actually patch. If you do not need the full resolution of 12 bits, simply just use the first couple of outputs.

13.2 algoquencer - Algorithmic sequencer

The Algoquencer is a versatile performance sequencer, that implements a completely new approach: It combines a classical trigger sequencer with a turing machine and other algorithms in order to create a very hands on pattern generator for live improvisation. Its main tasks are:

- trigger sequencer for drum voices
- pitch sequencer
- melody generator
- generator of repeating random CVs



It can also be used as a simple random number generator - may it be totally chaotic random numbers or self similar patterns like those generated by the so called "Turing Machine".

There are lots of interesting high-level parameters that you can easily map to pots on your controllers - such as *Activity*, *Variation*, *Déjà-vu* and many more. With a turn of a knob you can instantly increase or decrease the density or complexity of your patterns in various ways.

Here are some of the features:

- Up to 16 step buttons
- change the pattern length on the fly
- manually editable accents for each step
- ratchets and drum rolls
- fills
- deterministic and chaotic randomization
- simple muting
- fractal sequencing

If you use the Algoquencer for drumming, each **algoquencer** circuit plays just one voice - e.g. a snare drum. For orchestrating a whole drum kit simply use

more Algoquencers with possibly different parameters. It totally makes sense to use some of the pots and buttons with all drum instruments - e.g. a pot for *Déjà-vu* - and others on a per-instrument base, like *Activity*.

Here are some examples of how to use the Algoquencer circuit.

Pseudo random voltages / Turing machine

Without any inputs other than **clock** the algorithmic sequencer creates a sequence of random numbers that *repeat* over and over every 16 steps. This is much like the "Turing Machine". The voltage range of the **pitch** output defaults to 0 V ... 3 V:

```
[algoquencer]  
    clock = G1  
    pitch = 01
```

You can change the length to any other value up to 64 by using the **length** parameter:

```
[algoquencer]  
    clock = G1  
    pitch = 01  
    length = 12
```

If you do not like the default output voltage range you can adjust that with the inputs **pitchlow** and **pitchhigh**:

```
[algoquencer]  
    clock = G1  
    pitchlow = 1V
```

```
pitchhigh = 4V  
pitch = 01
```

dejavu controls the randomness - or to be more precise how random values are picked. It has a default of **1.0**. This means that once a random decision has been made for a certain step of the pattern it will be that way for ever. The same random pattern will repeat again and again. Making **dejavu** smaller will convert *some* of the decisions to be random while others still repeat unchanged over and over again.

You want to change the entire pattern? You can choose another one by setting **pattern** to an arbitrary integer number:

```
[algoquencer]  
    clock = G1  
    pitch = 01  
    length = 12  
    pattern = 5
```

Another way to change the pattern is to send a trigger to **nextpattern**, for example with a button:

```
[algoquencer]  
    clock = G1  
    pitch = 01  
    length = 12  
    dejavu = 1  
    nextpattern = B1.1
```

Do you like slowly evolving patterns (which is a feature from the "Turing Machine")? The **morphs** parameter - which is usually **0.0** - will introduce random changes to the repeating pattern in a very controlled way:

- Changes (aka morphs) are introduced each time the pattern starts (again) - never in-between
- The exact *number* of changes is controlled with the **morphs** parameter and is *not* random.
- The steps where these changes happen and the changes itself *are* random.

morphs takes a number between **0.0** and **1.0**. At **0.0** no morphs happen. At **1.0** every step will be morphed - thus completely changing the pattern every time it would repeat. Here is a table of how exactly the parameter affects the number of morphs per 64 steps. It is done in a way that is very suitable for mapping it directly to a pot and gives a very fine resolution at the left half of the pot:

morphs	morphs per 100 steps
0.0	no morphs
0.1	1
0.2	4
0.3	9
0.4	16
0.5	25
0.6	36
0.7	49
0.8	64
0.9	81
1.0	100

As you can see the smallest number of morphs - if you set **morphs** just a little above 0 - is one per 64 steps.

Note: If you are curious whether morphs are happening you can wire the output **morphled** to some LED. It will

flash whenever morphs happen.

Dejavu or morphs?

Did you get the difference between **dejavu** and **morphs**? Here once again:

- dejavu** controls, whether to use just complete random values (**dejavu = 0**) or repeating pseudo-random sequences (**dejavu = 1**).
- morphs** comes into play, when **dejavu** is > 0 and modifies the pseudo-random sequences from time to time a bit so they won't get boring.

True random voltages

If you do not want the random pitches to repeat you can set the **dejavu** parameter to 0. This transforms the **algoquencer** into a simple random number generator:

```
[algoquencer]
  clock = G1
  pitch = 01
  dejavu = 0
```

It can be very interesting to map **dejavu** to one of the pots of your controllers. That way you can change on-the-fly between structured melodies and complete randomness - or anything between!

Using the Algoquencer as drum sequencer

This is how you setup the Algoquencer for use as a drum sequencer. Like in the previous examples you need a clock signal. Also using a reset input helps you to sync

your drums with some external stuff. A trigger here resets the pattern to the first step:

```
[algoquencer]
  clock = G1
  reset = G2
```

A trigger into **clock** will move to the next step of the pattern. One into **reset** resets back to the first step.

Algoquencer supports up to 16 buttons (aka *step buttons*) for manually setting up a trigger pattern. If you assign less than 16 buttons then your patterns will be shorter. You probably want to assign these to buttons of your controllers, e.g.

```
button1 = B1.1
button2 = B1.2
button3 = B1.3
button4 = B1.4
```

In order for the LEDs in these buttons to work you also need to assign the **led...** outputs:

```
led1 = L1.1
led2 = L1.2
led3 = L1.3
led4 = L1.4
```

Please make sure that there is no "hole" in your definitions. You cannot use **button8** if you not also use **button1** through **button7**.

Note: You can use Algoquencer even without step buttons. This is like having an empty pattern, but **activity** will still work and create artificial beats if it is not zero.

Last but not least wire the output **trigger** to the trigger input of some drum voice.

```
trigger = 01
```

For a simple “normal” trigger sequencer this is enough. I’d suggest you setup this small example first and once it is up and running you investigate further features of Algoquencer. Here is the example once again complete for usage while we assume that you have an P2B8 controller:

```
[p2b8]
```

```
[algoquencer]
  clock    = I1
  reset    = I2
  button1 = B1.1
  button2 = B1.2
  button3 = B1.3
  button4 = B1.4
  led1     = L1.1
  led2     = L1.2
  led3     = L1.3
  led4     = L1.4
  trigger  = 01
```

Accents

Algoquencer supports setting or not setting an accent for each of the steps. For this there is a “second page” of the buttons where you can edit these accents. In order to access that accent page you need to wire the input **accentbutton** to one of your buttons (e.g. **B1.5**). Also wire the output **accent** to some external output jack and patch that to the accent input of your drum voice:

```
accentbutton = B1.5
accent      = 03
```

Now while you hold the accent button the step buttons will switch over to showing the accents instead of the normal beats. And you can set and remove accents now.

Note: if you do not want to be forced to *hold* the button while editing accents you can convert it into toggle button using the **[button]** circuit:

```
[button]
  button    = B1.5
  led       = L1.5
  output    = _ACCENTS
```

```
[algoquencer]
  # ... the other stuff
  accentbutton = _ACCENTS
  accent      = 03
```

Alternate steps

The Algoquencer just supports 16 steps, but there is a great way to extend your pattern to 32 or more steps. The concept for this is a bit unusual, but all the more musical and hands on. It goes like this:

There is an *alternate page* of another 16 buttons. These are like a third layer of buttons (if you account the accents for the second layer). Just like with the accents you define a button for bringing up that layer, for example:

```
alternatebutton = B1.7
```

While you hold that button you edit the alternate page instead of the normal steps.

Now: every active step in the alternate page will *flip* the according step in the normal page *for every second bar*.

That way you can have a variation of the pattern every second bar but you just edit the *differences* to the normal pattern. So adding or removing one beat every second bar can be done by activating exactly one step in the alternate page.

You are not limited to a pattern of two bars. By setting **alternatebars** to another value you can change the frequency of the alternate bar:

```
alternatebutton = B1.7
alternatebars = 4
```

Now bars 1 - 3 are played normally and every forth bar the alternate page is applied. That basically forms a pattern of 64 steps.

Pattern length and bars

As you have at most 16 buttons one pattern can have a length of at most 16 steps. The length of the pattern can be set in various ways:

- If you wire at least one **button1** then the length defaults to the number of wired buttons.
- This can be overridden by setting **length** to any value (e.g. **length = 7**).
- If you use the **lengthbutton** then you can interactively change the pattern length during your performance. This will always override the **length** input.

Add the button for changing the length is easy:

```
lengthbutton = B1.6
```

One *bar* usually has the same number of steps as your pattern. But if you set **repeats = 2**, one bar will consist

of two times the pattern (and thus lasts twice as long). Bars are useful when you use **fills** or **branches**.

Playing fills

Fills are additional beats the Algoquencer adds at the end of certain bars in order to play a musically interesting fill. In order to use this first wire **fills** to some CV or most likely to a pot:

```
fills = P1.1
```

Now if you crank up that pot clockwise then more and more beats will be added - with a tendency to the end of the bar. In music - however - playing a fill each bar is not very interesting. By setting **fillorder** to 1, 2 or 3 (or even a higher number) will make the fills assume a cycle of 2, 4 or 8 or move bars. Please see below for details.

Activity and random

Four inputs are key features of Algoquencer, since they extend it from a plain old trigger sequencer to an algorithmic drummer. These are **variation**, **activity**, **dejavu** and **morphs**. The latter two already have been discussed when using Algoquencer as random generator. They have the same effect here.

The default value of **variation** is 0.0. That means that Algoquencer will exactly play the pattern as you have dialled it in with your step buttons. If you increase that value (a pot is handy for doing this, of course) then randomly some of the beats will move to other steps. Setting **various** to 1.0 will completely alter your pattern. The number of beats will stay the same!

activity will change exactly that: the number of triggered beats in one bar. The default value is 0.5 - which is the center position if assigned to a pot. Here the number of played beats is exactly the same as you have set in your pattern. Turn it left to remove (randomly) some of the beats. Turn it right to add some. At 0.0 no beats are triggered, at 1.0 there is a beat for every clock cycle.

The **activity** also has an effect when you create random voltages. Here the voltage only changes when a "beat" happens at that step, even if you are not using the **trigger** output.

Further nifty parameters

There are some more interesting parameters like **rolls**, **offbeats**, **distribution** and **branches**. Please look at the table of inputs for more details.

Presets

The algoquencer supports up to 16 presets. Each preset comprises all settings that can be interactively changed, i.e. the activated steps, accents, alternate steps, the manually changed length, the state of the mute button and also the current random seed (which was modified by **nextpattern**, **prevpattern** or **reroll**).

There are three ways of switching between presets. The first way is easy to implement. Simply send the number of the current preset to the input **preset**. It has to be a number from 0 to 15. You can for example use a pot if you multiply it with 15:

```
[algoquencer]  
  preset = P1.1 * 15  
  ...
```

Now any change you make will immediately be saved to that current preset. If you change the preset number by turning the pot, another preset will immediately be loaded and activated.

The second - more sophisticated - way is to use triggers for loading and saving. These could be buttons, e.g.:

```
[algoquencer]  
  preset = P1.1 * 15  
  loadpreset = B1.1  
  savepreset = B1.2  
  ...
```

Now turning the knob does not load or save any preset. The input **preset** is just evaluated when you press **B1.1** or **B1.2**:

- A trigger to **savepreset** will save the current settings into the preset that is selected with the **preset** input.
- A trigger to **loadpreset** will copy the contents of the preset selected by **preset** into the current settings.

Note: In the second mode you effectively have 17 presets, since the "current settings" could also be considered to be a preset. The advantage of this mode is that playing around with the settings of the algoquencer does not immediately effect any of the presets.

Hint: In order to avoid saving or loading presets by mistake, have a look at the **button** (see page 103) circuit and the **longpress** output. It sends a trigger when a button is pressed and held for a certain time.

The **third** way is a combination of the first two ways. Here you work with triggers, as well. But these triggers at the same time hold the number of the preset to load or to save. This makes situations easier where you have one

button per preset

```
[mixer]
  input1 = B1.1 * 1
  input2 = B1.2 * 2
  input2 = B1.3 * 3
  output = _LOAD_PRESET

[mixer]
  input1 = B1.4 * 1
  input2 = B1.5 * 2
  input2 = B1.6 * 3
  output = _SAVE_PRESET
```

```
[algoquencer]
  loadpreset = _LOAD_PRESET
  savepreset = _SAVE_PRESET
```

This means that if the trigger CV has the value **2** when it is non-zero, it load preset number 2. This mode is automatically active, if you don't patch the **preset** input.

There is one drawback of this method: you cannot easily access preset number **0** that way, since the CV **0** is not sufficient for triggering the input. The trick is sending a value larger than **0.1** (which is the threshold for boolean "true" values) and less than **0.5** (which would be rounded to 1). So for example send a trigger with the value **0.3** to load or save preset number 0.

Sharing buttons between multiple algoquencers

The buttons on your controllers are a valuable ressources and not to be wasted lightheartedly. And especially the **algoquencer** uses quite a lot of buttons. But the good news is: you can share most of these buttons with other instances of **algoquencer**, to create a multi-track sequencer with just one set of buttons. You can even share the buttons with completely other circuits.

The key to this is the **select** input. If you patch it, all buttons and LEDs will just be used by this instance of **algoquencer** as long as **select** gets a high gate signal. Here is an example (which is just a sketch and not complete):

```
[algoquencer]
  select = _SELECT_1
  button1 = B1.1
  button2 = B1.2
  ...
  led1    = L1.1
  led2    = L1.2
  ...
```

```
[algoquencer]
  select = _SELECT_2
  button1 = B1.1
  button2 = B1.2
  ...
  led1    = L1.1
  led2    = L1.2
  ...
```

Now you need to make sure that at any given time either **_SELECT_1** or **_SELECT_2** is active. The easiest way is with a **butongroup**, because here you can add more and more tracks if you like. Let's assume that for switching between tracks you use the buttons **B2.7** (track 1) and **B2.8** (track 2). This would look like this:

```
[butongroup]
  button1 = B2.7 # select track 1
  button2 = B2.8 # select track 2
  led1 = L2.7
  led2 = L2.8
```

```
[algoquencer]
  select = L2.7 # becomes 1 if B2.7 is selected
  button1 = B1.1
```

```
button2 = B1.2
...
led1   = L1.1
led2   = L1.2
...
```

```
[algoquencer]
  select = L2.8 # becomes 1 if B2.8 is selected
  button1 = B1.1
  button2 = B1.2
  ...
  led1   = L1.1
  led2   = L1.2
  ...
```

Please note: the buttons **mutebutton** and **unmutebutton** and their according LEDs are **not** handled by the **select** jack. The idea is that they always get their own dedicated buttons. This allows you to quickly mute or unmute several tracks at once.

Input	Type	Default	Description
clock			Clock input. This is mandatory. For each clock pulse the sequencer is advanced by one step.
reset			Reset input. A trigger here switches back to step 1.
button1 ... button16			1 st ... 16 th step button. Assign these buttons to buttons on your controllers.
length	1°2°3		Sets the length of the pattern. Note: if you use lengthbutton , this input is ignored as soon as the length button has been used for the first time. If you have assigned at least one button, the default value of length is the number of buttons you have assigned. Otherwise it defaults to 16 . The maximum length is 64. Any larger number will be truncated to 64.
pattern	1°2°3		Selects a pattern of pseudo random values. If you set dejavu to 1, all "random" decision are deterministic and repeat again and again. If you do not like these choices, you can choose a different pattern, just by setting this input to any integer number you like. The default pattern is 0. If you patch a pot here, simply multiply it by the number of different patterns you want to select, e.g. pattern = P1.1 * 10 . This will allow you to select one of the pattern 0, 1, ... 10. Note: If you use pattern , the trigger inputs nextpattern , prevpattern and reroll are ignored.
nextpattern			Switches forward to the next pseudo random pattern.
prevpattern			Switches back to the previous pseudo random pattern.
reroll			Select one of the pseudo random patterns completely by random.
clearpage			A trigger here unselects all step buttons in the currently active page (normal, alternate, accent).
pitchlow		0.0	This set a lower voltage boundary for the pitch output for notes that are randomized.
pitchhigh		0.3	This set an upper voltage boundary for the pitch output for notes that are randomized.
pitchresolution	1°2°3	0	If this is non-zero, it make the pitch output adopt that number of possible discrete values. E.g. if you set it to 2 , only the values set by pitchlow and pitchhigh are possible. A value of 3 will allow an additional value in the middle, and so on.
gatelen		0.1	The gate length in input clock cycles. A value of 0.5 (5 V) thus means half a clock cycle. A steady input clock is needed for this to work. Please note that if the gate length is ≥ 1.0 , two succeeding notes will get a steady gate, which essentially means legato. When playing rolls, i.e. more than one beat per step, the gate length is divided by the number of rolls. That way the gates get shorter and even at a gatelen close to 1.0 the gates are still audible and do not merge together.

Input	Type	Default	Description
lengthbutton			Map this to a button like B1.1 . While you press and hold this button the sequencer switches to <i>change length mode</i> . While in this mode a press of one of the step buttons will change the length of the pattern. Also while in this mode the LEDs of the step buttons will show the current length. If you do not like to hold the button but switch it on and off, you can create a toggle button with [button] and send its output here.
repeats	1◦2◦3	1	Usually one bar has the length of one pattern. Setting this to 2 will consider one bar as a run of two times through the pattern. So if you have 8 buttons and bars = 2, one bar will be 16 steps, where the 1 st and 9 th step are set by button1 , 2 nd and 10 th by button2 and so on. Why should that be useful? Well - the difference shows up when you use fills , or branches or work with the <i>alternate</i> pattern. These three algorithms work based on <i>bars</i> . And repeats = 2 makes one bar have 16 steps, even if you just have eight buttons.
alternaterepeats	1◦2◦3		If you are use using repeats and alternatebars / alternatebutton at the same time, with this input you can specify a different value for repeats when it comes to selecting the alternate button page. Assume you have eight buttons and repeats = 2 and alternatebars = 2. Then Algoquencer will play two times your 8-step pattern normally and two times alternated (since two times the 8 steps form one bar). This results in a form of A A B B. If you want your form rather to be A B A B, set alternaterepeats = 1. This way, when it comes to alteration, the length of one bar is just normal length (8 steps here).
branches	1◦2◦3	0	Enables the branching feature (sometimes also called fractal sequencing). When branches = 1, then every second bar will be using other random values - giving a sequence of the bars  . With branches = 2 you get a sequence of the form  A value of 3 creates an even longer sequence that repeats itself after eight bars:  Note: this only takes effect when you set dejavu > 0. The largest effect is when it is set to 1. And the you need to use either variation or set activity to a value greater than 0.5. Because otherwise Algoquencer will strictly play the gates that you've set with your buttons and then every bar will be the same, of course.
mutebutton			Wire this to a button like B1.2 . When you press the button once, all triggers are muted. Pressing again unmutes them. So this behaves like a toggle [button] in itself. You probably want to wire mutaled to the LED in that button, e.g. L1.2 . It show the mute state. The mute button works together with the unmute button (see below). Note: even if you use the select jack in order to overlay your buttons with several algoquencers, the mutebutton will always be active. The idea is to always have direct access to this button.

Input	Type	Default	Description
unmutebutton			A trigger to this jack resets the mute button exactly at the beginning of the next bar. While waiting for that to happen, the output unmuted will blink. Wire this to the LED in the button. Note: even if you use the select jack in order to overlay your buttons with several algoquencers, the mutebutton will always be active. The idea is to always have direct access to this button.
accentbutton			While this input is high you are in <i>accent editing mode</i> . This is very similar to the mode where you set the length. But now for each step you edit whether this step is outputting an accent when triggered. You might want to use a toggle button for this function, so you can operate without holding down the button all the time.
alternatebutton			If this input is high, you are in <i>alternate editing mode</i> . Every Algoquencer has an alternate set of steps. Each step that is currently activated <i>toggles</i> the state of the normal step, but only for each even bar. This allows to introduce variations of the pattern that occur every second bar.
alternatebars	1 ⋅ 2 ⋅ 3	2	With this input you can change the influence of the alternatebutton . Per default the pattern alternation is done every second bar. You can change this to any number you like with this input. Values less than 1 will be considered as one - which means that every bar is alternated.
accentlow		0.0	This value is output at accent when a note without an accent is being triggered or when no note is triggered at all.
accenhigh		1.0	This value is output at accent while a note with an accent is triggered. The value will be kept for the full time of the clock cycle.
activity	0 ⋅ 0.5 ⋅ 1		<p>This is the most important parameter and you will probably wire it to a pot like P1.1. The activity controls, how "busy" the sequencer is playing, or in other words how often a step gets an active gate (and thus a changing output pitch).</p> <p>Let's first assume that variation is set to 0.0 (which is the default). Then at a value of 0.5 (or pot at 12'oclock) Algoquencer will exactly play that pattern that you have set with the step buttons. Turning the knob CCW will remove more and more beats from the pattern until it is completely silent at a value of 0.0 (or pot fully CCW). But if you turn up the knob above the middle position then more and more <i>additional</i> beats will be placed into your pattern in a random way until - at 1.0 - a trigger will happen at <i>every</i> beat.</p> <p>Note: If you do not use step buttons, this parameter behaves slightly different: A value of 0.5 then means an activity of 50%, which means that exactly the half of the steps will get an event. This is different from a situation where you <i>have</i> defined buttons but all are deselected. In that case 0.5 means that exactly the number of beats of your pattern are being played, which is zero in that case.</p>

Input	Type	Default	Description
variation		0.0	<p>The <i>variation</i> controls how strictly Algoquencer will stick to the pattern that you have set with your step buttons. You probably want to wire this to a knob. A value of 0.0 (or the knob fully CCW) will allow no variations. Your pattern will be played exactly as it is. If the <i>activity</i> goes beyond 0.5, additional beats will be placed, of course. And these are random.</p> <p>If you increase the variation, more and more beats of your pattern are being replaced with other beats - while keeping the total number of beats the same. If you set variation to 1.0 (or the pot fully CW) then your pattern is completely ignored except for the actual number of beats it contains.</p>
dejavu		1.0	<p>The dejavu parameter controls what <i>random</i> should mean. If dejavu = 0.0, then all random decisions are completely chaotic - and every time a decision is taken the dice are being rolled again.</p> <p>At dejavu = 1.0 on the other hand - once a random decision has been taken for a certain step in a certain bar, it will stay always the same from now on. This will lead to repeating exactly the pattern bars over and over again. We sometimes call this random to be "deterministic".</p> <p>Any position in between will choose some of the steps as chaotic random and some of the steps as deterministic.</p>
morphs		0.0	<p>This parameter will introduce changes in formerly taken random decisions from time to time. If you set it above zero, at every start of a bar <i>some</i> of the deterministic random decisions will be remade. Setting morphs = 1 will essentially disable dejavu, since all decisions are redone every bar anyway then.</p> <p>If you know the Turing Machine: In principle that has the same idea, but Algoquencer has a few improvements:</p> <ul style="list-style-type: none"> • The number of random changes is exactly controlled by the setting. At each specific setting of morphs the same number of changes will be done at each bar. • Changes only appear at the beginning of each bar. If you use branches, they will appear whenever you sequence is over. • Small settings will introduce just one morph each 64th step.
offbeats		0.5	<p>Whenever random beats are being placed then this setting controls whether <i>downbeats</i> or <i>offbeats</i> should be preferred. At a setting of 0.5 there will be no difference. If you increase the value then more and more offbeats will appear. Offbeats are steps with an <i>even</i> number, like 2, 4, 6 and so on. Value smaller than 0.5 will prefer downbeats.</p> <p>Offbeats sound more "complex" and downbeats more simple or "down to earth".</p>
distribution		0.5	This is very similar to offbeats , but this time you decide whether beats should be placed rather in the first half of the bar or in the second half.

Input	Type	Default	Description								
fills	0 ↪ 1	0.0	When this parameter is set above 0.0 , additional beats will be placed in order to make the beat more “active”. This happens at musically useful times controlled by fillorder (see below). The additional beats within the bar are placed in a way that prefers the end of the bar. If there are already too many beats in the bar then the fill will <i>remove</i> or change some instead.								
fillorder	1 ⋅ 2 ⋅ 3	0	This integer number controls how fills are being placed:								
			<table border="1"> <tr> <td>0</td><td>every bar</td></tr> <tr> <td>1</td><td>every second bar</td></tr> <tr> <td>2</td><td>small fill in bar 2, big fill in bar 4</td></tr> <tr> <td>3</td><td>tiny fill in bar 2 and 6, medium fill in bar 4, big fill in bar 8</td></tr> </table>	0	every bar	1	every second bar	2	small fill in bar 2, big fill in bar 4	3	tiny fill in bar 2 and 6, medium fill in bar 4, big fill in bar 8
0	every bar										
1	every second bar										
2	small fill in bar 2, big fill in bar 4										
3	tiny fill in bar 2 and 6, medium fill in bar 4, big fill in bar 8										
rolls	0 ↪ 1	0.0	This parameter controls if drum rolls (or ratchets as you might call it) are being created. At 0.0 no rolls are being created. At 1.0 every beat will be converted into a roll. Rolls always happen before the actual beat, they lead to it. If you are using this feature for snare rolls you might want to use the output rollvelocity for controlling the snare volume.								
rollcount	1 ⋅ 2 ⋅ 3	1	Number of additional beats for playing the roll. Setting rollcount = 0 would disable rolls. All these beats are distributed in the clock tick before the beat the roll is leading to. The first beat of the roll is exactly one tick before that beat - or more if you increase rollsteps .								
rollsteps	1 ⋅ 2 ⋅ 3	1	Length of the roll in clock ticks (steps). The total number of additional beats is thus rollcount × rollsteps								
rollstartvelo	~~~	0.5	Rolls can be played with an increasing velocity. This first beat starts with the velocity set with this parameter. Then every beat gets a bit louder until the last beat is played with velocity 1.0 . The velocity for rolls is output at the jack rollvelocity .								
pitch1 ... pitch16	~~~	↳	You can use these inputs, if you want the pitches of the pitch output play a certain melody. That way the Algoquencer behaves like a normal melody sequencer - but all the algorithmic parameters will be applied. For example variation will also be applied to these notes. Note: If length is larger than 16, these pitch inputs will be cycled through, so step 17 uses pitch1 , step 18 uses pitch2 and so on.								
select	_____	↳	The select input allows you to overlay buttons and LEDs with multiple functions. If you use this input, the circuit will process the buttons and LEDs just as if select has a positive gate signal (usually you will select this to 1). Otherwise it won't touch them so they can be used by another circuit. Note: even if the circuit is currently not selected, it will nevertheless work and process all its other inputs and its outputs (those that do not deal with buttons or LEDs) in a normal way.								

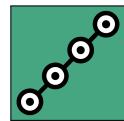
Input	Type	Default	Description
selectat	1 o 2 o 3		This input makes the select input more flexible. Here you specify at which value select should select this circuit. E.g. if selectat is 0 , the circuit will be active if select is <i>exactly 0</i> instead of a positive gate signal. In some cases this is more convenient.
preset	1 o 2 o 3		This is the preset number to save or to load. Note: the first preset has the number 0 , not 1 ! For the whole story on presets please refer to page 19 . This circuit has 16 presets, so this number ranges from 0 to 15 .
loadpreset			A trigger here loads a preset. As a speciality you can use the trigger for selecting a preset at the same time.
savepreset			A trigger here saves a preset.
clear			A trigger here loads the default start state into the circuit. The presets are not affected, unless you use direct preset switching with the preset input and without triggers. And that case the current preset is also cleared.
clearall			A trigger here loads the default start state into the circuit and into all of its presets.
dontsave		0	If you set this to 1 , the state of the circuit will not saved to the SD card and not loaded from the SD card when the Droid starts.

Output	Type	Description
trigger		Here comes the trigger output. Patch this to the trigger input of your drum or synth voice.
gate		The gate output is alternative to the trigger and has a variable length. It is useful when Algoquencer is used for creating melodies. Patch the gate input of an envelope or something similar here.
pitch		Outputs the (pseudo-)random voltage (unquantized) at each step with an active gate. This honors all the settings that control the randomness and variation, like dejavu , variation , fills and branches .
accent		Whenever a beat with an accent is being played, the value set by accenthigh is sent here, otherwise accentlow . If you are wiring this to one of the jacks of the G8 expander then that will output just 0V and 5V of course.
led1 ... led16		1 st ... 16 th LEDs of the step buttons. Assign these to the LEDs in the step buttons.
barled1 ... barled4		Patch these output to some LEDs in order to show you the current bar in the sequence.
rollvelocity		If you enable rolls, then the velocity of the roll beats will be output here. For normal beats this will always be 1.0 .
startofbar		At the beginning of every bar a trigger is output here.
mutedled		Wire this to the LED in your mute button. It will then be lit while the voice is muted.
unmutedled		Wire this to the LED in your unmute button (if used). It will blink while the unmute is waiting for the start of the next bar.

Output	Type	Description
morphled		This output will get a trigger every time a morph happens. It is intended to be wired to an LED.
fillsled		This output will get a trigger every time a fill beat is being played. Wire this to some LED if you like.
branch	1◦2◦3	This output will output the current branch number, e.g. 1, 2, 3 and so on. If you do not use branches then it is always 1.
lengthoutput	1◦2◦3	Outputs the currently selected length. This is useful if you are using the lengthbutton for interactively changing the length of the pattern and want to share that setting with other circuits.

13.3 arpeggio - Arpeggiator - pattern based melody generator

This circuit creates melodic patterns based on simple rules and many interesting configuration settings, which can lead to very simple but also most complex patterns.



Introduction

In order to better understand, how the arpeggiator works, let's compare four different ways for constructing melodies:

Sequencer	manually composed melodies
Random generator	completely chaotic sequences
Turing machine, Algoquencer	pseudo-random melodies, which repeat themselves
Arpeggiator	melodies constructed from rules

The rules for the arpeggiator can be as simple as *on each clock tick play the next note in the C minor scale*. Additional parameters are for example the pitch range, i.e. the start and the end note.

The arpeggiator shares root, scale and interval selection with **chord** (see page 115) and **minifonion** (see page 200). If you own a Sinfonion: the arpeggiator in the DROID is working a bit differently and is more about general principles than about preprogrammed patterns. That makes it more flexible and powerful.

The simplest possible example

As always, we start with the simplest possible example. And it is simple, indeed, since each of the many param-

ters has a useful default value. The only input the arpeggiator *always* needs is a clock input. The word "clock" is probably a bit misleading since it doesn't *need* to be a steady clock signal. It can be any rhythmic pattern you like. Each clock tick advances the melody to the next note and a new pitch CV will be presented at **output**, which is, of course, in the typical 1V/oct scheme.

```
[arpeggio]  
  clock = I1  
  output = 01
```

Patch **I1** to an external clock and **01** to the 1V/oct of some synth voice. The easiest way is to use the same clock also for triggering the voice's envelope.

Now you will hear a C major scale (lydian) being played step by step in a range from 0 V to 2 V. This makes 15 notes, since the scale consists of the seven notes C, D, E, F♯, G, A and B and is repeated over two octaves, but the C is here three times: at the beginning, in the middle and at the end:



When it reaches the end it immediately starts over again. So the second "bar" is really just 7 eights here!

Root, scale and interval selection

You probably don't like lydian C major. Changing that is easy with the inputs **root** and **degree**. Please have a look

at the **minifonion** circuit (see page 200) for an explanation of these parameters. Let's go for a D minor (natural) scale as an example:

```
[arpeggio]  
  clock = I1  
  output = 01  
  root = 2  
  degree = 7
```

Now we get:



Patterns

This "go through the scale" mode is just one of several possible patterns. The pattern is selected with the **pattern** input. And the default value of **0** produces the result we just have seen. Let's look at pattern **1**. This goes two steps forward and one step backward in the scale:

```
[arpeggio]  
  clock = I1  
  output = 01  
  root = 2  
  degree = 7  
  pattern = 1
```

Since pattern 1 repeats its structure every three notes it's best to display it in a metric that is divisible by three:



Pattern 2 is similar, but makes one double step forward instead of two single steps:



Pattern 3 goes a double step forward, a double step backward and a single step forward:



Pattern 4 is even more sophisticated. It goes a double step forward, a single step forward, a double step *backward* and again a single step forward:



Pattern 5 is a bit different since for each note it flips a coin for deciding whether to go one step up or down.

And Pattern 6 simply randomly chooses one of the possible notes. So strictly spoken this has nothing to do with "arpeggiation", but it's fun, so what?

Note: it's not entirely impossible that future versions of the arpeggiator introduce new patterns. So better do not yet rely on these numbers to be fixed forever.

The range

Per default the pattern is played in a range of two octaves. But that can be set easily with two parameters. **pitch** defines the lowest possible pitch of a note. The arpeggiator will choose the start note such that it is in the scale and just at or above this pitch.

And **range** defines the voltage range the pattern is being played upwards until it starts again. So if **range** is 2 V, you get a range of two octaves. A range of 0 will deform the pattern into one single note.

For interactive playing, mapping **pitch** and **range** to pots is fun:

[p2b8]

```
[arpeggio]
  clock = I1
  output = 01
  pitch = P1.1
  range = P1.2
```

Changing the playing direction

So far all pattern where going more or less upwards. From lower notes to higher notes. This can be changed by setting **direction** to 1. Now the arpeggiator starts with the highest allowed note and reverses the pattern for going downwards. Why not map this setting to a nice toggle button?

[p2b8]

```
[button]
  button = B1.1
```

```
led = L1.1
output = _DIRECTION
```

[arpeggio]

```
  clock = I1
  output = 01
  pitch = P1.1
  range = P1.2
  direction = _DIRECTION
```

Another setting that influences the direction is the **pingpong** parameter. This is a binary (gate) input, too. If it is set to 1 the direction of the pattern changes into the opposite once the end of the range has been reached. Check this example...

[arpeggio]

```
  clock = I1
  output = 01
  pingpong = 1
  pitch = 0
  range = 7/12V
```

... will create the following melody:



Why is that? Well - $\frac{7}{12}$ V is the same as 7 semitones, which is in turn one fifth. Since no root and degree are defined we are back at C major lydian. The pattern is 0 (default) - hence the simple note-by-note scale. And **pingpong = 1** makes the pattern going down again after having reached the upper limit.

Octaves up and down

The nice thing about all these parameter is that you can combine them all. They interact with each other and most combinations do useful things (well, when using the "random" pattern, the direction and pingpong are without effect, of course). And there is one more fun setting: **octaves**. This can be **0** (default) or **1** or **2**.

When octaves is **1**, each note is directly followed by the same note one octave above. That octave note is ignoring the **range**-parameter. It is always in addition to the selected range. Here is an example:

```
[arpeggio]
  clock = I1
  output = 01
  range = 1V
  octaves = 1
```

And here is the pattern this creates:



Set **octaves = 2** and you get the same but the octaves go *down* instead:



Dropping

The **drop** input lets you select different schemes of leaving out notes from the original line of scale notes. For example **drop = 1** will leave out every second note. Here is an example:

```
[arpeggio]
  clock = I1
  output = 01
  drop = 1
```

This will create the following melody:



If you have a closer look, you will see that in the upper octave other notes are being played than in the lower octave. This can sound very interesting!

Dropping can, of course, be combined with other patterns as well. Let's see the line for pattern 1:



There are more dropping-schemes. Please have a look into the table of input parameters down below.

Note selection

The most important thing comes last. For didactical reasons! What *really* makes this arpeggiator so musically versatile is its interval selection. This is the same as for

the **minifonion** (see page 200) and the chord generator (page 115).

The point is that you are not restricted to the seven notes of a scale. For this there are seven inputs **select1**, **select3**, ... **select13** that select the notes of the current scale and another five inputs **selectfill1** ... **selectfill5** that select the notes not in the current scale. These 12 inputs are binary inputs that expect either **0** or one **1**. Each of them selects one of the seven intervals of the scale for being part of the chord. Here is a table of all these inputs and the notes they would select in a C major or C minor scale:

Input	interval	step	C ^{maj}	C ^{min}
select1	root	I	C	C
select3	3rd	III	E	E \flat
select5	5th	V	G	G
select7	7th	VII	B	B \flat
select9	9th = 2nd	II	D	D
select11	11th = 4th	IV	F	F
select13	13th = 6th	VI	A	A \flat

Let's make a simple example: The arpeggio of a C major triad over two octaves going up and down again:

```
[arpeggio]
  clock = I1
  select1 = 1
  select3 = 1
  select5 = 1
  output = 01
  pingpong = 1
```

And here is the result:



One typical way to select these notes is with seven toggle buttons. Much like the Sinfonion. Assign the output of each of the seven buttons to one of these functions:

```
[p2b8]

[button]
  button = B1.1
  led = L1.1

[button]
  button = B1.2
  led = L1.2

[button]
  button = B1.3
```

```
led = L1.3

[button]
  button = B1.4
  led = L1.4

[button]
  button = B1.5
  led = L1.5

[button]
  button = B1.6
  led = L1.6

[button]
  button = B1.7
```

```
led = L1.7

[arpeggio]
  clock = I1
  select1 = L1.1
  select3 = L1.2
  select5 = L1.3
  select7 = L1.4
  select9 = L1.5
  select11 = L1.6
  select13 = L1.7
  output = 01
```

Now you can switch on and off scale notes for being part of the patterns. Have fun!

Input	Type	Default	Description
pitch	$\text{♪ } \frac{1V}{Oct}$	0V	Sets the base pitch of the arpeggio. The first note of the pattern will be the nearest selected note just above that pitch.
range	$\text{♪ } \frac{1V}{Oct}$	2V	Selects the range between the lowest and highest note of the arpeggio. A range of 0 means that there is just one single note possible and the arpeggio will stick to that note. A value of 1 V (or 0.1) means that the arpeggio will run over one octave. The maximum allowed range is 0.8 (8 octaves). Higher values will be capped to that.
clock			This input is vital: each trigger here make the arpeggio move forward by one step and adapt the pitch output. Without a clock the arpeggio will do nothing but stick to the same note all the time.
reset			Resets the arpeggio to the first step of the current pattern.

Input	Type	Default	Description
pattern	1◦2◦3	0	Selects one of a list of arpeggio pattern. The following patterns are available:
		0	step forward through the allowed notes →
		1	two steps forward, one step backward →→←
		2	double step forward, one step backward ⇒←
		3	double step forward, double step backward, single step forward ⇒←→
		4	double step forward, single step forward, double step backward, single step forward ⇒→←→
		5	random single step forward or backward ↔
		6	random jump to any allowed (other) note ⇕
direction		0	Sets the general direction in which the pattern moves. 0 means upwards and 1 means downwards.
pingpong		0	If set to 1, the pattern will reverse its direction once it has reached the end of the range. Otherwise it restarts from the beginning. So enabling pingpong is a bit like a triangle wave, whereas otherwise it's more like a sawtooth.
butterfly		0	If set to 1, every second note in the range of selected notes will be mirrored. So for example you have selected the notes 1 - 10, the new order will be 1, 10, 2, 9, 3, 8, 4, 7, 5, 6
drop	1◦2◦3	0	Selects a scheme of skipping some of the allowed scale notes. Four different values are allowed:
		0	Do not skip any notes ①②③④⑤⑥
		1	Skip every second selected note ①②③④⑤⑥
		2	Skip every third selected note ①②③④⑤⑥
		3	Skip the 2 nd and 3 rd note of each group of three ①②③④⑤⑥
octaves		0	When this is set to 1 or 2, each note will be followed by the same note one octave up (for 1) or down (for 2) respectively. These additional octave notes are in addition to the selected range.
		0	Don't play octaves
		1	Each note is followed by the same note one octave up
		2	Each note is followed by the same note one octave down

Input	Type	Default	Description																										
startnote	1◦2◦3	0	When startnote is set to non-zero, it will force the pattern to begin with a certain scale note regardless of the current note selection. 1 will select the first note of the scale (root), 2 the second and so on until 7 , which selects the 7 th as start note. This force start note replaces the note that would originally have been played.																										
root	1◦2◦3	0	Set the root note here. 0 means C, 1 means C♯, 2 means D and so on. If you multiply the value of an input like I1 with 120, then you can use a 1V/Oct input for selecting the root note via a sequencer, MIDI keyboard or the like. Also then you are compatible with the ROOT CV input of the Sinfonion. <table border="1"> <tr><td>0</td><td>C</td></tr> <tr><td>1</td><td>C♯</td></tr> <tr><td>2</td><td>D</td></tr> <tr><td>3</td><td>D♯</td></tr> <tr><td>4</td><td>E</td></tr> <tr><td>5</td><td>F</td></tr> <tr><td>6</td><td>F♯</td></tr> <tr><td>7</td><td>G</td></tr> <tr><td>8</td><td>G♯</td></tr> <tr><td>9</td><td>A</td></tr> <tr><td>10</td><td>A♯</td></tr> <tr><td>11</td><td>B</td></tr> <tr><td>12</td><td>C</td></tr> </table>	0	C	1	C♯	2	D	3	D♯	4	E	5	F	6	F♯	7	G	8	G♯	9	A	10	A♯	11	B	12	C
0	C																												
1	C♯																												
2	D																												
3	D♯																												
4	E																												
5	F																												
6	F♯																												
7	G																												
8	G♯																												
9	A																												
10	A♯																												
11	B																												
12	C																												

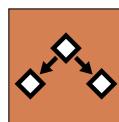
Input	Type	Default	Description																								
degree	1 o 2 o 3	0	Set the musical scale. This is a number from 0 to 11 . At 12 this repeats over again. Please refer to the introduction for the list of scales. If you multiply an input like I1 with 120 , this will internally scale to one scale per semitone and you are compatible with the DEGREE CV input of the Sinfonion.																								
			<table border="1"> <tr><td>0</td><td>lyd - Lydian major scale (it has a $\sharp 4$)</td></tr> <tr><td>1</td><td>maj - Normal major scale (ionian)</td></tr> <tr><td>2</td><td>X7 - Mixolydian (dominant seven chords)</td></tr> <tr><td>3</td><td>sus - mixolydian with 3rd/4th swapped</td></tr> <tr><td>4</td><td>alt - Altered scale</td></tr> <tr><td>5</td><td>hm⁵ - Harmonic minor scale from the 5th</td></tr> <tr><td>6</td><td>dor - Dorian minor (minor with $\sharp 13$)</td></tr> <tr><td>7</td><td>min - Natural minor (aeolian)</td></tr> <tr><td>8</td><td>hm - Harmonic minor ($\flat 6$ but $\sharp 7$)</td></tr> <tr><td>9</td><td>phr - Phrygian minor scale (with $\flat 9$)</td></tr> <tr><td>10</td><td>dim - Diminished scale (whole/half tone)</td></tr> <tr><td>11</td><td>aug - Augmented scale (just whole tones)</td></tr> </table>	0	lyd - Lydian major scale (it has a $\sharp 4$)	1	maj - Normal major scale (ionian)	2	X7 - Mixolydian (dominant seven chords)	3	sus - mixolydian with 3 rd /4 th swapped	4	alt - Altered scale	5	hm ⁵ - Harmonic minor scale from the 5 th	6	dor - Dorian minor (minor with $\sharp 13$)	7	min - Natural minor (aeolian)	8	hm - Harmonic minor ($\flat 6$ but $\sharp 7$)	9	phr - Phrygian minor scale (with $\flat 9$)	10	dim - Diminished scale (whole/half tone)	11	aug - Augmented scale (just whole tones)
0	lyd - Lydian major scale (it has a $\sharp 4$)																										
1	maj - Normal major scale (ionian)																										
2	X7 - Mixolydian (dominant seven chords)																										
3	sus - mixolydian with 3 rd /4 th swapped																										
4	alt - Altered scale																										
5	hm ⁵ - Harmonic minor scale from the 5 th																										
6	dor - Dorian minor (minor with $\sharp 13$)																										
7	min - Natural minor (aeolian)																										
8	hm - Harmonic minor ($\flat 6$ but $\sharp 7$)																										
9	phr - Phrygian minor scale (with $\flat 9$)																										
10	dim - Diminished scale (whole/half tone)																										
11	aug - Augmented scale (just whole tones)																										
select1			<p>Gate input for selecting the <i>root note</i> as being an allowed interval. When you want to create a playing interface for live operation you can patch the output of a toggle button (made with the circuit [button]) here.</p> <p>Note: When all select and selectfill inputs are 0, automatically all seven scale notes are selected, i.e. select1 ... select13 will be set to one.</p>																								
select3			Gate input for selecting the 3 rd .																								
select5			Gate input for selecting the 5 th .																								
select7			Gate input for selecting the 7 th .																								
select9			Gate input for selecting the 9 th (which is the same as the 2 nd).																								
select11			Gate input for selecting the 11 th (which is the same as the 4 th).																								
select13			Gate input for selecting the 13 th (which is the same as the 6 th).																								

Input	Type	Default	Description
<code>selectfill1</code>		off	Selects the alternative 9 th (i.e. the 9 th that is <i>not</i> in the scale).
<code>selectfill2</code>		off	Selects the alternative 3 rd (i.e. the 3 rd that is <i>not</i> in the scale).
<code>selectfill3</code>		off	Selects the alternative 4 th or 5 th . In most cases this is the diminished 5 th .
<code>selectfill4</code>		off	Selects the alternative 13 th (i.e. the 1 st 3 that is <i>not</i> in the scale).
<code>selectfill5</code>		off	Selects the alternative 7 th (i.e. the 7 th that is <i>not</i> in the scale).
<code>tuningmode</code>		off	While this is 1 , the circuit will output the value set by <code>tuningpitch</code> instead of the actual pitch. This is meant to be a help for tuning your VCOs.
<code>tuningpitch</code>	$\frac{1V}{Oct}$	0V	This pitch CV will be output while the tuning mode is active.
<code>transpose</code>	$\frac{1V}{Oct}$	0V	This value is being added to the output pitch when not in tuning mode. It can be used for musical transposition or adding a vibrato.

Output	Type	Description
<code>output</code>	$\frac{1V}{Oct}$	This is what it's all about: here comes the pitch CV for the current arpeggio note.

13.4 bernoulli - Random gate distributor

This circuit implements a “bernoulli gate”. For each gate or trigger received at **input** there is made a random decision of whether to forward that gate to **output1** or **output2**. The probability for each of the outputs can be shifted with the parameter **distribution**. It determines the probability of a gate signal to go to **output1**.



Example:

```
[bernoulli]
  input      = G1
  distribution = P1.1
  output1    = G2
  output2    = G4
```

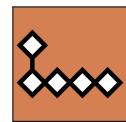
Note: each time a positive trigger edge is seen at **input** a new random decision is made for which output to use. From now on that chosen output gets an exact copy of the input signal – even if it is not a simple trigger signal but something more complex like an envelope. The other output will send 0 V.

Input	Type	Default	Description
input		0	Send gate or trigger signals here.
distribution	0 (0.5) 1	0.5	This controls the probability of a gate to be forwarded to output1 . A value of 0.5 means 50%.

Output	Type	Description
output1		Gates from input are forwarded here if the random decision was in favour of output 1.
output2		Gates from input are forwarded here if the random decision was in favour of output 2.

13.5 burst - Generate burst of pulses

This circuit produces - when triggered - a number of pulses. It can be used for solving various musical or technical tasks. Look at this example:



```
[burst]
trigger = I1
hz      = 10
count   = 5
output  = 01
```

When a trigger arrives at **I1**, the output **01** will send five triggers in a row, with a distance of 0.1 seconds (thus 10 Hz). The gate length is fixed to half of the cycle (thus here 0.05 seconds). This means that the pulse width is 50% - or in other words - the faster the burst the shorter the outgoing triggers.

Note: When a new trigger arrives while the current burst is still ongoing, it will not be finished but restarted from the beginning immediately.

If you want the bursts to be synchronized to a musical clock, you can use the **taptempo** input (here **I2**):

```
[burst]
taptempo = I2
count     = 4
trigger   = I1
output    = 01
```

Similar to the circuit **lfo** (see page 163), there is a third input for selecting the speed: **rate**. This works on a 1V/Oct base, so here is an example for outputting the bursts at half of the clock speed (-1 V pitches down one octave, which is the same as half of the speed):

```
[burst]
taptempo = I2
rate      = -1V
count     = 4
trigger   = I1
output    = 01
```

burst can also be used for very fast switching through things like presets in external gear. Here you might want fast updates. Simply set a very high frequency. Burst makes sure that the actual output rate is limited to the maximum the DROID hardware can do, so not one single burst can get lost. Also you might want to use the **skip** input, which skips a certain number of ticks before starting. This can be used to send out a reset signal to some input and *after that* sending a couple of **skip forward** triggers to some other input:

```
[burst]
hz = 5000
skip = 5
count = 3
trigger = I1
output = 01
```

Another very simple yet useful application of **burst** is converting a gate signal into a short trigger. That way you can for example convert a *running* state from MIDI into a reset trigger. Since **count** defaults to **1**, you don't need any parameters except the input and output:

```
[burst]
trigger = _MIDI_RUNNING
output = _RESET
```

In this example the trigger is emitted when the running state goes from 0 to 1.

Simple clocked trigger delay

Another application of **burst** is a clocked trigger delay. Consider the following patch:

```
[burst]
taptempo = I1
trigger = I2
skip = 7
output = 01
```

A trigger at **I2** will be delayed by 7 clock cycles.

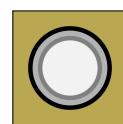
Note: This simple trigger delay has no memory of more than one trigger. Any ongoing trigger currently being delayed is overridden and forgotten as soon as the next trigger arrives. If that is what you want, fine. If you are looking for a more complex trigger delay, you find one in the circuit **triggerdelay** (see page 279) circuit.

Input	Type	Default	Description
rate		0.0	Frequency control: The default frequency of the burst rate is 1 Hz (one trigger per second or 60 BPM if you like). Each volt doubles the frequency. So an input of 1 V (a number of 0.1) speeds up to two triggers per second (120 BPM), 2 V (0.2) creates triggers at 4 Hz (240 BPM) and so on. On the other hand negative voltages reduce the speed, so -1 V (-0.1) will give 0.5 Hz (30 BPM) and so on.
taptempo			Feed a reference clock here and the burst will run at the speed of that clock - albeit optionally modified by rate . Please see page 21 for details on using taptempo inputs.
hz		1.0	Set the frequency in Hz directly by setting a number here. This is exclusive to taptempo , but will work in combination with rate .
trigger			Send a trigger here in order to start the bursts
reset			Send a trigger here to immediately stop any ongoing burst.
count	1 0 2 0 3	1	Number of triggers to send in one burst.
skip	1 0 2 0 3	0	Number of time slots to wait before starting with the burst.

Output	Type	Description
output		The triggers are output here.

13.6 button - Does all sorts of useful things with buttons

This is a utility circuit for efficiently working with the buttons of your controllers. It can implement toggle buttons (that do on/off) or even have three or four states.



It can detect long presses and double clicks

and also helps you to overload one button with several switchable functions. Note: If you just need a plain momentary button without any of these or other nifty features, you can use the register **B1.1**, **B1.2**, etc. directly and do not need this circuit.

Note: don't forget to declare your controllers at the top of your patch with lines like **[p2b8]** or **[b32]**. In the below examples I've omitted these declarations for sake of simplicity.

Toggle buttons

The most common use of **button** is to implement a toggle button. That's a button that changes from on to off and back at each press of the button. The current state of the button will persist on your SD card so you don't lose your state if you switch off your rack.

Typically you will wire the **button** input to one of your controller's buttons like **B1.1** and **led** to the LED in that button (**L1.1**). LED will then always visualise the current state of the button. As a side effect the LED register **L1.1** will store the button state as a value **0** or **1** and hence can be used by some other circuit as an input.

Here is a typical example. The button is being used for enabling the loop in a CV looper:

```
[button]  
button = B1.4
```

```
led = L1.4
```

```
[cvlooper]  
loop = L1.4
```

If you do not want the state of the button to be persisted on the SD card, use **dontsave = 1**. This make sense for the CV looper since the loop is apparently empty anyway when your DROID starts.

```
[button]  
button = B1.4  
led = L1.4  
dontsave = 1
```

```
[cvlooper]  
loop = L1.4
```

Usually the button switches between the two values **0** and **1**. Sometimes, however, you need different values. For this purpose there are the two inputs **offvalue** and **onvalue**. They set two alternative values for the "off" and "on" states. And the output **output** outputs the selected value (**led** still goes to **0** and **1**). Here is an example for a toggle button that switches a clock divider between **2** and **4**:

```
[button]  
button = B1.4  
led = L1.4  
offvalue = 2  
onvalue = 4  
output = _CLOCK_DIV
```

```
[clocktool]  
input = G1 # external clock
```

```
output = G2  
divide = _CLOCK_DIV
```

Of course **offvalue** and **onvalue** are CV controllable. How can this make sense? Well - as they can take variable inputs you can use a button for directly switching between two different input CV signals. The following example will use a button to switch between two different wave forms of an LFO (see page [163](#)). The button **B3.1** switches between sawtooth and sine and sends the result to **01**.

```
[lfo]  
hz = 2  
sawtooth = _SAWTOOTH  
sine = _SINE
```

```
[button]  
button = B3.1  
led = L3.1  
offvalue = _SAWTOOTH  
onvalue = _SINE  
output = 01
```

Buttons with three or four states

Sometime you might want more than just two values. **button** supports switching between up to four values. Use the **states** input and set it to **3** or **4**. In the following examples **output** will go through the values **0**, **1**, **2** and **3**:

```
[button]  
button = B1.1  
led = L1.1  
states = 4  
output = _SOMETHING
```

If you don't like the default values, use the inputs **value1** through **value4** for setting the four values. In fact **offvalue** is the same as **value1** and **onvalue** as **value2**. If you specify **value3** or **value3**, **states** is automatically set accordingly and you can simply omit it. The following example switches between *four* different wave forms of an LFO:

```
[lfo]
hz      = 2
sawtooth = _SAWTOOTH
sine    = _SINE
square   = _SQUARE
triangle = _TRIANGLE

[button]
button  = B3.1
led     = L3.1
value1  = _SAWTOOTH
value2  = _SINE
value3  = _SQUARE
value4  = _TRIANGLE
output   = 01
```

If you have three or four states, the LED will use different brightness levels for indicating the current state.

Momentary buttons

If you just need a momentary button (one that just lights up while you hold it down), strictly spoken you don't need a **button** circuit. You can directly use the **B** register, like in this example:

```
[algoquencer]
nextpattern = B1.1
```

Sometimes, however, you may want to make use of some of the features of the **button** circuit without creating a toggle button. This is easily done by setting **states = 1**:

```
[button]
states = 1
button  = B1.1
led     = L1.1

[algoquencer]
nextpattern = L1.1
```

Now you are ready for adding some fun stuff like overlaying one button with multiple functions (see below) or using the **longpress** output.

Long and short presses

When creating patches, you will constantly run out of buttons. One way to increase the effective number of buttons is to map two different actions on a button depending on whether it is pressed long or short. For this purpose there is the **longpress** output. Consider the following example:

```
[button]
button  = B1.1
led     = L1.1
output   = _SOME_STATE
longpress = _LONG
```

A button press with a duration below 1.5 secs will toggle the LED **L1.1** as usual. If you hold the button longer than 1.5 seconds, the output **longpress** will get high until you release the button. And the state of **L1.1** does **not** toggle.

If you don't want the button to toggle any state, but just distinguish between long and short presses, you can use the **shortpress** output:

```
[button]
button    = B1.1
longpress = _LONG
shortpress = _SHORT
```

Note: The output **led** is not used here since we are just interested in the presses and you cannot really see the LED anyway while your finger is on the button. If you want the LED anyway, set **states = 1** so it won't toggle:

```
[button]
button    = B1.1
led      = L1.1
states   = 1
longpress = _LONG
shortpress = _SHORT
```

Using **output** does not do the same as **shortpress**: it always is high as long as your finger is on the button (and the button is selected).

Sharing buttons

You can never have too many buttons! It's more likely that you have too few. So you want to overlay one or more buttons with multiple functions.

The key to this is the **select** input of the **button** circuit. If you patch this, the circuit will only interact with the actual button and LED if **select** is active (e.g. set to **1**). Otherwise it will continue to output its current value to **output** and leave the control of the button and the LED to some other circuit.

The following example uses the button **B1.1**, (which is not overloaded!) for switching between two "layers" or "banks" of buttons. And in each bank the button has a different meaning. Note how I use the **negated** output of the button. That is **0** if the normal output is **1** and vice versa.

In order to keep things short, the bank just consists of the single button **B1.2**. Of course in practice this wouldn't make sense since you wouldn't actually save a button, but you get the idea...

```
[button]
  button = B1.1
  led = L1.1
  output = _BANK1
  negated = _BANK2

[button]
  select = _BANK1
  button = B3.1
  led = L3.1
  output = _VIRTUAL_BUTTON_1

[button]
  select = _BANK2
  button = B3.1
  led = L3.1
  output = _VIRTUAL_BUTTON_2
```

Note: If you need more than two banks, consider switching with a **buttongroup** (see page [108](#)).

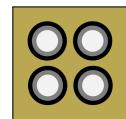
Input	Type	Default	Description
button			The actual push button. Usually you want to wire this to B1.1 , B1.2 and so on: to one of the push buttons of your controllers. Each time that input goes from low to high, the state of the push button will toggle.
onvalue		1.0	Value sent to output when the push button is on. You can also use a dynamic signal here. This is an alternative name for the input value1 .
offvalue		0.0	Value sent to output when the push button is off. This is an alternative name for the input value2 .
value1 ... value4			The up to four values to output at output when the button is on the according state. value1 is the same as offvalue and value2 is the same as onvalue . The default values of these four inputs are 0 , 1 , 2 and 3 , so in many cases you don't need to specify them.
doubleclickmode		off	This input can enable a <i>double click mode</i> when set to 1 . In that mode the button only toggles its constant state if you double press it in a short time. Otherwise it behaves like a momentary button, that inverts the persisted state (which you toggle with the double click). Note: The double click mode is only makes sense if the number of states is 2.
longpresstime		1.5	The number of seconds after which a button press is considered as a <i>long press</i> .
states	1 2 3	2	Number of states this button can have. The default value is 2 , which creates a toggle button which changes between on and off at each press. A value of 1 creates a momentary button. Note: If you just need a plain momentary button, you can directly use B1.1 , B1.2 and so on. You don't need an extra circuit. But if you want things like overloading (with select) or the longpress output, this does make sense. The maximum number of states is 4. When the button has 3 or 4 states, every press will switch to the next state and then back to the first state again.
startvalue	1 2 3	0	State of the push button when you switch on your system or on a trigger to clear . If you have three states, the start value needs to be 0 , 1 or 2 . With four states, it can also be 3 .
select			The select input allows you to overlay buttons and LEDs with multiple functions. If you use this input, the circuit will process the buttons and LEDs just as if select has a positive gate signal (usually you will select this to 1). Otherwise it won't touch them so they can be used by another circuit. Note: even if the circuit is currently not selected, it will nevertheless work and process all its other inputs and its outputs (those that do not deal with buttons or LEDs) in a normal way.
selectat	1 2 3		This input makes the select input more flexible. Here you specify at which value select should select this circuit. E.g. if selectat is 0 , the circuit will be active if select is <i>exactly</i> 0 instead of a positive gate signal. In some cases this is more convenient.
preset	1 2 3		This is the preset number to save or to load. Note: the first preset has the number 0 , not 1 ! For the whole story on presets please refer to page 19 . This circuit has 16 presets, so this number ranges from 0 to 15 .
loadpreset			A trigger here loads a preset. As a speciality you can use the trigger for selecting a preset at the same time.
savepreset			A trigger here saves a preset.

Input	Type	Default	Description
<code>clear</code>			A trigger here loads the default start state into the circuit. The presets are not affected, unless you use direct preset switching with the <code>preset</code> input and without triggers. And that case the current preset is also cleared.
<code>clearall</code>			A trigger here loads the default start state into the circuit and into all of its presets.
<code>dontsave</code>		0	If you set this to 1, the state of the circuit will not be saved to the SD card and not loaded from the SD card when the Droid starts.

Output	Type	Description
<code>led</code>		When the button state is <code>on</code> , a value of 1.0 will be sent to that output - regardless of the values in <code>onvalue</code> and <code>offvalue</code> . If the number of states is 3 or 4 the output gets intermediate values so the attached LED will be dimmed into different brightness levels. Usually you wire that output to a LED register, e.g. to <code>L1.1</code> , <code>L1.2</code> and so on.
<code>output</code>		This output will output the current button states. This is usually 0 for off and 1 for on. If <code>states</code> is 3 or 4, the values 2 or 3 are output for the additional states. You can modify all four values with the inputs <code>offvalue/value1</code> , <code>onvalue/value2</code> , <code>value3</code> and <code>value4</code> . Note: if you haven't changed any of these inputs and <code>states</code> is unchanged or 1 or 2, the <code>led</code> output will output the same values.
<code>inverted</code>		The same as <code>output</code> , but sends <code>onvalue</code> when the button is off and <code>offvalue</code> when the button is on. If <code>states</code> is 3 or 4, the order of the four output values will be mirrored (probably a feature that is rarely of any use).
<code>negated</code>		Similar to <code>inverted</code> , but always sends 1 when the button is off and 0 when the button is on - independent of the values of <code>onvalue</code> and <code>offvalue</code> . When <code>states</code> is 3 or 4, this output will be 1 if the button is off and 0 in the other three states.
<code>longpress</code>		Goes from 0 to 1, when the button is pressed and held for at least 1.5 seconds. If this output is used, the effect of toggling the button's state is delayed until the button is <i>released</i> . When it's released after 1.5 secs, no toggling happens. This will avoid double actions for long presses.
<code>shortpress</code>		Emits a trigger, when the button is pressed, regardless of the settings of <code>states</code> . If at the same time <code>longpress</code> is used (which is the whole point in this output), the trigger is delayed until the button is released and only sent, if it was not a long press.

13.7 buttongroup - Connected buttons

This utility circuit combines a number of push buttons into a group that behave as a unit. One classic operation is to form a group of "radio buttons". This means that at any time just one of these buttons is on and all others are off.



The following example uses four buttons for selecting one of the voltages 0 V, 1V, 2V and -1V. This voltage is then being sent to the output jack. This could be used as an octave switch or the like. The four buttons **B2.1** ... **B2.4** are grouped in a way that just one button is on and the others are off. The four selectable voltages are assigned to one button each. The value of the currently active button is being sent to the output. The outputs **output1** ... **output4** will be set to 1 if their corresponding button is active and are used for controlling the LEDs within the buttons.

```
[buttongroup]
  button1 = B2.1
  button2 = B2.2
  button3 = B2.3
  button4 = B2.4
  led1 = L2.1 # LED in button 2.1
  led2 = L2.2
  led3 = L2.3
  led4 = L2.4
  value1 = 0V
  value2 = 1V
  value3 = 2V
  value4 = -1V
  output = 01
```

If you set **maxactive** to a number greater than one, more than one button can be active at the same time. If this is the case then the sum of the values of all active but-

tons will be sent to the output. Here is an example, where three buttons are being used for selecting a number between 0 and 7 by selecting any combination of the buttons "1", "2", and "4".

```
[buttongroup]
  button1 = B2.1
  button2 = B2.2
  button3 = B2.3
  led1 = L2.1 # LED in button 2.1
  led2 = L2.2
  led3 = L2.3
  value1 = 1
  value2 = 2
  value3 = 4
  minactive = 0 # allow all buttons to be off
  maxactive = 3 # allow all buttons to be on
  output = 01
```

Overlaying buttons

When you make more complex DROID patches, it's likely that you might run out of buttons. In such a situation you can *overlay* buttons with multiple functions and use other buttons to switch between these layers.

Consider the following example: We have one P2B8 controller. The buttons 1 and 2 should switch between the layers *root note* and *scale*. We do this with a simple button group (you could also use a **button** circuit and save one button, but for simplicity we allow us two here):

```
[p2b8]
[buttongroup]
```

```
button1 = B1.1
button2 = B1.2
led1 = L1.1
led2 = L1.2
```

The remaining six buttons select either one of six possible root notes or one of six possible scales (adhering to the scheme of the **minifonion** circuit, see page 200). Please note how we have added a **select** input at each of both circuits to make sure that at any given time exactly one of the two groups is selected:

```
[buttongroup]
  select = L1.1 # be active only when L1.1 is active
  button1 = B1.3
  button2 = B1.4
  button3 = B1.5
  button4 = B1.6
  button5 = B1.7
  button6 = B1.8
  led1 = L1.3
  led2 = L1.4
  led3 = L1.5
  led4 = L1.6
  led5 = L1.7
  led6 = L1.8
  value1 = 0 # C
  value2 = 2 # D
  value3 = 5 # F
  value4 = 7 # G
  value5 = 9 # A
  value6 = 10 # Bb
  output = _ROOT
```

```
[buttongroup]
  select = L1.2 # be active only when L1.2 is active
  button1 = B1.3
```

```
button2 = B1.4
button3 = B1.5
button4 = B1.6
button5 = B1.7
button6 = B1.8
led1 = L1.3
led2 = L1.4
led3 = L1.5
led4 = L1.6
led5 = L1.7
```

```
led6 = L1.8
value1 = 1 # major
value2 = 6 # dorian minor
value3 = 7 # natural minor
value4 = 9 # phrygian minor
value5 = 10 # diminished scale
value6 = 2 # mixolydian
output = _DEGREE
```

Here you can patch `_ROOT` and `_SCALE` to some `minifonion`, `arpeggio` or other circuit that works with scales.

Now, with the top buttons you can switch between root and scale selection and with the remaining six buttons select either the root or the scale.

Input	Type	Default	Description
<code>minactive</code>	1°2°3	1	Minimum number of active buttons. If you set this to 2, then it is guaranteed that at least 2 buttons are active. If you set this to 0, then it is possible to switch off all buttons. The <code>output</code> will be set to 0.0 in that case.
<code>maxactive</code>	1°2°3	1	Maximum number of active buttons. It is an error to set this to 0, since this would make this circuit useless.
<code>longpresstime</code>	~~~	1.5	The number of seconds after which a button press is considered as a <i>long press</i> .
<code>button1 ... button32</code>	---		1 st ... 32 nd button of the group. Any positive trigger seen here will toggle this button. And another button might go on or off in order to make sure that the number of active buttons is within the allowed range.
<code>value1 ... value32</code>	~~~	0	Value that will be sent to the output if the 1 st ... 32 nd button is active. These inputs default to 0 for <code>value1</code> , 1 for <code>value2</code> and so on and 31 for <code>value32</code> .
<code>startbutton</code>	1°2°3	1	If you set this parameter to the number of a button, that button will be selected (and all other deselected) at the start when no state is loaded or at a trigger to <code>clear</code> . This allows you to set useful default values for your button groups. Note: this only makes sense if <code>maxactive</code> is not 0. Also it is not possible to select more than one button, even in a group where <code>maxactive</code> is greater than 1.
<code>select</code>	---	0	The <code>select</code> input allows you to overlay buttons and LEDs with multiple functions. If you use this input, the circuit will process the buttons and LEDs just as if <code>select</code> has a positive gate signal (usually you will select this to 1). Otherwise it won't touch them so they can be used by another circuit. Note: even if the circuit is currently not selected, it will nevertheless work and process all its other inputs and its outputs (those that do not deal with buttons or LEDs) in a normal way.
<code>selectat</code>	1°2°3	0	This input makes the <code>select</code> input more flexible. Here you specify at which value <code>select</code> should select this circuit. E.g. if <code>selectat</code> is 0, the circuit will be active if <code>select</code> is exactly 0 instead of a positive gate signal. In some cases this is more convenient.
<code>preset</code>	1°2°3	0	This is the preset number to save or to load. Note: the first preset has the number 0, not 1! For the whole story on presets please refer to page 19. This circuit has 16 presets, so this number ranges from 0 to 15.

Input	Type	Default	Description
loadpreset			A trigger here loads a preset. As a speciality you can use the trigger for selecting a preset at the same time.
savepreset			A trigger here saves a preset.
clear			A trigger here loads the default start state into the circuit. The presets are not affected, unless you use direct preset switching with the preset input and without triggers. And that case the current preset is also cleared.
clearall			A trigger here loads the default start state into the circuit and into all of its presets.
dontsave		0	If you set this to 1 , the state of the circuit will not be saved to the SD card and not loaded from the SD card when the Droid starts.

Output	Type	Description
led1 ... led32		This output will be on / 1.0 , whenever the 1 st ... 32 nd button is active and off / 0.0 otherwise. Wire this to the LED in the button. If you have wired select , these LED outputs will do nothing (not even send 0) unless this circuit is selected.
buttonoutput1 ... buttonoutput32		These are individual outputs for every button in the group. They output button's value when it is active, otherwise 0 . If valueX is not defined for buttonX , the value 1 is output (not the button's number!). Note: in contrast to the led output, these outputs are not affected by select but always functional. One application of these outputs is to use a buttongroup with maxactive = X and minactive = 0 as a cheap bunch of X toggle buttons in one single circuit and still use select .
output		The sum of the values of all active buttons will be sent here. If no button is active, 0.0 is being output.
buttonpress		Emits a trigger if any button is being pressed
longpress		Emits a trigger, when any button is pressed for at least 1.5 seconds. If this jack is used, buttonpress will emit a signal if the button in question is released before the 1.5 seconds, not immediately. This way you trigger <i>either</i> at buttonpress or at longpress , not at both.
selectionchanged		Emits a trigger when the selection of the buttons has changed. This is not quite the same as shortpress , since a button press might not lead to a change. Also in multi button situations (e.g. maxactive = 4 where you have 7 buttons) the change is delayed up to 25 ms due to detection of bursts of quasi simultaneous presses.

13.8 calibrator - VCO Calibrator

This circuit allows you to precisely compensate for decalibrated or otherwise imperfectly tracking VCOs - which is probably a property of all existing analog VCOs to some degree. It does this by applying one specific adaptation value per individual octave. This way you can make even those VCO track well over 10 octaves, that would normally only do 2 or 3.



The calibration of the error compensation is done manually - by you. At first this may seem like a disadvantage. In practice, however, this is much easier and more accurate than the way some "autotune" modules do it. Those modules have an additional input for "listening" to a waveform output of the oscillator and measure and adjust the tracking at a button press.

The advantages of manual tuning are:

- You don't need an extra waveform output of your VCO.
- You can calibrate sound sources with complex wave forms, whose pitch is hard to grab by autotune devices.
- You can change the correction at any time during a live performance without your audience noticing.
- It's possible to make one VCO follow the (imperfect) tracking of a second one, in order to create perfect FM sounds while just one VCO needs to be adapted.
- It's also possible to fix the tracking of unprecise pitch CV generators, such as sequencers, quantizers or MIDI interfaces.

The calibrator circuit happily profits from the DROID's highly precise, linear and low-jitter ADCs and DACs. And using eight such circuits one DROID could fix the tuning

of up to eight VCOs.

How to use

Here is a typical patch for the use of the calibrator:

```
[calibrator]
  input      = I1
  output     = O1
  nudgeup   = B1.1
  nudgedown = B1.3
  ledup     = L1.1
  leddown   = L1.3
```

The original pitch information from the sequencer, quantizer, MIDI converter or whatever comes into **I1**. The adapted pitch goes to **O1** and from there to the V/Oct input of your VCO. Of course the pitch information could also come from some internal circuit like the **minifonion** (page 200). In that case **input** is connected to an internal patch cable coming from that circuit.

Now with the two buttons **B1.1** and **B1.3** you can adjust the tuning up and down at any time while playing. Each button press just very slightly shifts the pitch up or down. The adjustment is only done for the octave that's currently playing. **calibrator** saves one calibration value for each octave from 0 to 8 and also one for the pitches below 0 V and those above 8 V. Your tuning profile is automatically saved to the memory card.

Pressing both buttons at the same time resets the calibration of the current octave.

For a good result I suggest either using a precise tuner or playing the voice at the same time as a reference voice

and try to minimize the audible beatings.

As second way of using the VCO calibrator is specifying a tuning adjustment for each octave by a fixed number (or a potentiometer if you can afford). This is done with the inputs **tune0** ... **tune8** and **tunelowtail** and **tunehightail**. A value of 1.0 means an upwards tuning of one semitone (100 cents) per octave, and -1.0 likewise downwards.

Persistence

As always, the internal state of the **calibrator** circuit is automatically saved to your SD card and loaded when your **DROID** starts.

But what if you are using several calibrators, each for a different (and differently tracking) VCO? How do you know which of the saved calibration states is applied to which VCO?

The answer to this is: all calibrators in your patch are enumerated starting from 1. For each of them there is one configuration saved to the SD card, based on that number. So when you modify the calibration of the third **calibrator** circuit in your patch, the modified configuration will be saved as belonging to calibrator number 3.

So if you make sure that each VCO is always handled by the same **calibrator** circuit you will always get the right configuration.

If you for example remove the first calibrator from your patch, the second one will become the new first one and load its calibration state when you load the new patch. If you don't want that to happen, simply keep the calibra-

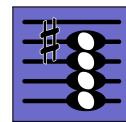
tor in the patch, even if you don't need it anymore. It is sufficient to keep just the line [**calibrator**] without any further jack specifications.

Input	Type	Default	Description
input	 $\frac{1V}{Oct}$	0V	Patch your V/Oct pitch input here.
nudgeup			A trigger here (most likely a button press) will modify the tuning of the currently played note (as read by input) upwards by one cent (or by nudgeamount if that is used).
nudgedown			A trigger here will modify the tuning of the currently played note down.
nudgeamount		0.01	Changes the amount each button press detunes. A value of one would mean one semitone, so the default value of 0.01 corresponds to one cent ($\frac{1}{100}$) of a semitone.
tune0 ... tune8		0.0	Explicit tuning of the octaves 0 through 8 - if you do not want to nudge manually. tune0 sets the tuning for the input pitch of 0 V, tune1 for 1 V and so on. A value of 1 means a tune adjustment of one semitone - which is 100 cent. The maximum detuning is ± 1 Octave (at a value of ± 12).
tunelowtail		0.0	Tuning adaption for the negative voltage range. A value of 1 means an upwards tuning of one semitone <i>per octave</i> , -1 likewise downwards.
tunehightail		0.0	Tuning adaption for voltages > 8 V. A value of 1 means an upwards tuning of one semitone <i>per octave</i> , -1 likewise downwards.
select			The select input allows you to overlay buttons and LEDs with multiple functions. If you use this input, the circuit will process the buttons and LEDs just as if select has a positive gate signal (usually you will select this to 1). Otherwise it won't touch them so they can be used by another circuit. Note: even if the circuit is currently not selected, it will nevertheless work and process all its other inputs and its outputs (those that do not deal with buttons or LEDs) in a normal way.
selectat	1..2..3		This input makes the select input more flexible. Here you specify at which value select should select this circuit. E.g. if selectat is 0, the circuit will be active if select is <i>exactly</i> 0 instead of a positive gate signal. In some cases this is more convenient.
preset	1..2..3		This is the preset number to save or to load. Note: the first preset has the number 0, not 1! For the whole story on presets please refer to page 19. This circuit has 4 presets, so this number ranges from 0 to 3.
loadpreset			A trigger here loads a preset. As a speciality you can use the trigger for selecting a preset at the same time.
savepreset			A trigger here saves a preset.
clear			A trigger here loads the default start state into the circuit. The presets are not affected, unless you use direct preset switching with the preset input and without triggers. And that case the current preset is also cleared.
clearall			A trigger here loads the default start state into the circuit and into all of its presets.
dontsave		0	If you set this to 1, the state of the circuit will not be saved to the SD card and not loaded from the SD card when the Droid starts.

Output	Type	Description
<code>output</code>	$\frac{1V}{Oct}$	The calibrated pitch goes out here.
<code>ledup</code>	$\circ \curvearrowleft_1$	When nudgeup is mapped to a button (which is most likely), map this output to the according LED and it will indicate whenever it's currently adjusting the output pitch upwards.
<code>leddown</code>	$\circ \curvearrowright_1$	This is the LED for nudgedown , which indicates downwards adjustment.

13.9 chord - Chord generator

This circuit creates the pitch information for up to four voices of a musical chord. This means that you can attach the Volts per octave inputs of up to four synth voices and they will play a nice musical chord. Hereby you have the flexibility of building your chord out of any of the seven notes of a selected scale. So you are not limited to root, 3rd, 5th and 7th. The algorithm is similar to that in the Sinfonion but has an adapted mode for three voiced chords in addition.



Minimal example

Here is the most simple (and probably useless) example: it will play a C major 7 chord, i.e. output the respective pitch CVs for the notes C, E, G and B at the outputs **01**, **02**, **03** and **04**:

[chord]

```
output1 = 01
output2 = 02
output3 = 03
output4 = 04
```

Output **01** will be at 0 V, representing a C. Of course, if you just have three voices, don't use **output4** and you will get a C major triad.

Selecting root and scale

Most likely you do not want to play in C major all the time (or even never!), so you can select the root note and the scale with the inputs **root** and **degree**. Setting **root** to 2 and **degree** to 7, for example, will select D natural minor:

```
[chord]
output1 = 01
output2 = 02
output3 = 03
output4 = 04
root    = 2
degree  = 7
```

Both **root** and **degree** range from **0** to **11**. Please refer to the description of **minifonion** (see page 200) for a complete list of all available scales. It has the same logic for **root** and **degree** and is thus compatible with **chord**.

But why the heck is that input named **degree**?? Well, it's a jargon from the Sinfonion and does make sense there in some contexts. Please have a look into the manual of the Sinfonion if you are interested!

Selecting the pitch of the notes

Per default all outputs are in the first octave, i.e. in the range 0 V ... 1 V. Per convention this is very low and probably sounds ugly. With the **pitch** input you can set the *minimum* pitch of the lowest output chord note. In the next example this is read from **I1**. So you could, for example, patch a sequencer here and have the chord outputs play a kind of four voiced melody:

```
[chord]
pitch = I1
output1 = 01
output2 = 02
output3 = 03
output4 = 04
root    = 2
degree  = 7
```

The **spread** parameter controls the **maximum** pitch of the highest output chord note. It is always relative to the pitch of the *lowest* note *plus one octave*. So if **spread** is 1.5 V (or 0.15), for example, the maximum allowed distance between the lowest and the highest chord note is 2.5 octaves. As lowest note the chord generator places the chord note that is nearest above the **pitch** input. As highest note it places the one nearest to upper bound of the allowed range and the remaining notes are distributed in between with the most equal spacing possible.

Selecting the chord notes

What makes the Sinfonion and also the harmonic circuits in the **DROID** stand apart from other modules is the flexibility of note selection. So e.g. in C major, you are not limited to playing the chord C/E/G/B. In fact you can choose *any* subset from the currently selected scale.

For this there are seven inputs **select1**, **select3**, ... **select13** that select the notes of the current scale and another five inputs **selectfill1** ... **selectfill5** that select the notes not in the current scale. These 12 inputs are binary inputs that expect either **0** or one **1**. Each of them selects one of the seven intervals of the scale for being part of the chord. Here is a table of all these inputs and the notes they would select in a C major or C minor scale:

Input	interval	step	C ^{maj}	C ^{min}
select1	root	I	C	C
select3	3rd	III	E	E _b
select5	5th	V	G	G
select7	7th	VII	B	B _b
select9	9th = 2nd	II	D	D
select11	11th = 4th	IV	F	F
select13	13th = 6th	VI	A	A _b

One typical way to select these notes is with seven toggle buttons, which is then much like the Sinfonion does it. Assign the output of each of the seven buttons to one of these functions:

[p2b8]

```
[button]
  button = B1.1
  led = L1.1

[button]
  button = B1.2
  led = L1.2

[button]
  button = B1.3
  led = L1.3

[button]
  button = B1.4
  led = L1.4

[button]
  button = B1.5
  led = L1.5

[button]
  button = B1.6
  led = L1.6
```

```
[button]
  button = B1.7
  led = L1.7

[chord]
  select1 = L1.1
  select3 = L1.2
  select5 = L1.3
  select7 = L1.4
  select9 = L1.5
  select11 = L1.6
  select13 = L1.7
  output1 = 01
  output2 = 02
  output3 = 03
  output4 = 04
```

Now you can use the buttons to change the chord notes on the fly. Of course, however, you also can use other signals for the selection. Maybe random gates, slowly running LFOs, a sequencer, whatever you like!

But what happens, if you do **not** select exactly four notes?

- If you don't select *any* note (or do not patch the **select**-inputs at all), all scale notes are selected.
- If you select just *one* note, all four outputs will play that same note.
- If you select *two* notes, **output1** and **output3** will play the first note and **output2** and **output4** the second one.
- If you select *three* notes, **output4** will play the same as **output1**.
- If you select *five*, *six* or *seven* notes, just the first four notes will be used.

If some of the notes are doubled and you use a large enough **spread**, they will be placed at different octaves.

By the way: It's of course no problem to just use three or

even just two of the outputs, if you don't need or have a total of four voices.

Chord inversion

The chord generator lets you nail down the chord structure to a certain *inversion*. If you set **inversion** to **1**, the root note (or, to be more precise, the first selected note) will be placed as the lowest note. Similarly the inversions **2**, **3** and **4** will make the respective other selected notes the lowest note.

Setting **inversion** to **0** (which is the default) will allow any note to be the lowest. This allows the chord to be closest to the **pitch** input.

Triggered mode

The **trigger** input is essentially a sample & hold for the **outputs**. So as soon as you patch that input, all outputs are frozen until the next trigger.

Chords with three voices

The chord generation circuit can also create chords with just three output voices. Simply omit the output **output4**. When it is not connected, the "three voice mode" is activated:

```
[chord]
  output1 = 01
  output2 = 02
  output3 = 03
  root    = 2
  degree  = 7
```

All parameters work as expected but there are some important adaptions. This is *not* the same as using the four voiced mode and just look at the first three outputs. For example:

- The spreading uses a simplified algorithm with just

a bottom, middle and top note.

- If just three intervals are selected, you don't get a duplication of the first note on **output2**, as you would otherwise.

Chords with two voices

Even if just two outputs are connected, you can still make use of this circuit. Now just the first two **select...** inputs are taken into account. But things like inversion and spreading works nevertheless.

Input	Type	Default	Description
pitch	1V Oct	0V	This sets the minimum pitch of the lowest note of the chord.
spread	1V Oct	0V	Selects the range between the lowest and highest note of the chord measured in 1V/oct, while a spread of 0 means that all chord notes are within <i>one</i> octave, a spread of 1 V means that the notes are spread out over <i>two</i> octaves and so on.
inversion	1 • 2 • 3	0	Selects the inversion of the chord. 1 means that the root note should be the lowest note, 2 will make the second selected note the lowest note, 3 the 3 rd and 4 the 4 th . The default, however, is 0 and doesn't fix the inversion. Rather that inversion is chosen that creates the chord closest to the input pitch.
trigger			This jack is optional. If you patch it, the Chord generator just reads a new input pitch when it receives a trigger.

Input	Type	Default	Description																										
root	1◦2◦3	0	<p>Set the root note here. 0 means C, 1 means C#, 2 means D and so on. If you multiply the value of an input like I1 with 120, then you can use a 1V/Oct input for selecting the root note via a sequencer, MIDI keyboard or the like. Also then you are compatible with the ROOT CV input of the Sinfonion.</p> <table border="1"> <tr><td>0</td><td>C</td></tr> <tr><td>1</td><td>C#</td></tr> <tr><td>2</td><td>D</td></tr> <tr><td>3</td><td>D#</td></tr> <tr><td>4</td><td>E</td></tr> <tr><td>5</td><td>F</td></tr> <tr><td>6</td><td>F#</td></tr> <tr><td>7</td><td>G</td></tr> <tr><td>8</td><td>G#</td></tr> <tr><td>9</td><td>A</td></tr> <tr><td>10</td><td>A#</td></tr> <tr><td>11</td><td>B</td></tr> <tr><td>12</td><td>C</td></tr> </table>	0	C	1	C#	2	D	3	D#	4	E	5	F	6	F#	7	G	8	G#	9	A	10	A#	11	B	12	C
0	C																												
1	C#																												
2	D																												
3	D#																												
4	E																												
5	F																												
6	F#																												
7	G																												
8	G#																												
9	A																												
10	A#																												
11	B																												
12	C																												

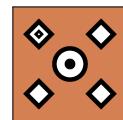
Input	Type	Default	Description																								
degree	1 o 2 o 3	0	Set the musical scale. This is a number from 0 to 11 . At 12 this repeats over again. Please refer to the introduction for the list of scales. If you multiply an input like I1 with 120 , this will internally scale to one scale per semitone and you are compatible with the DEGREE CV input of the Sinfonion.																								
			<table border="1"> <tr><td>0</td><td>lyd - Lydian major scale (it has a #4)</td></tr> <tr><td>1</td><td>maj - Normal major scale (ionian)</td></tr> <tr><td>2</td><td>X7 - Mixolydian (dominant seven chords)</td></tr> <tr><td>3</td><td>sus - mixolydian with 3rd/4th swapped</td></tr> <tr><td>4</td><td>alt - Altered scale</td></tr> <tr><td>5</td><td>hm⁵ - Harmonic minor scale from the 5th</td></tr> <tr><td>6</td><td>dor - Dorian minor (minor with #13)</td></tr> <tr><td>7</td><td>min - Natural minor (aeolian)</td></tr> <tr><td>8</td><td>hm - Harmonic minor (b6 but #7)</td></tr> <tr><td>9</td><td>phr - Phrygian minor scale (with b9)</td></tr> <tr><td>10</td><td>dim - Diminished scale (whole/half tone)</td></tr> <tr><td>11</td><td>aug - Augmented scale (just whole tones)</td></tr> </table>	0	lyd - Lydian major scale (it has a #4)	1	maj - Normal major scale (ionian)	2	X7 - Mixolydian (dominant seven chords)	3	sus - mixolydian with 3 rd /4 th swapped	4	alt - Altered scale	5	hm ⁵ - Harmonic minor scale from the 5 th	6	dor - Dorian minor (minor with #13)	7	min - Natural minor (aeolian)	8	hm - Harmonic minor (b6 but #7)	9	phr - Phrygian minor scale (with b9)	10	dim - Diminished scale (whole/half tone)	11	aug - Augmented scale (just whole tones)
0	lyd - Lydian major scale (it has a #4)																										
1	maj - Normal major scale (ionian)																										
2	X7 - Mixolydian (dominant seven chords)																										
3	sus - mixolydian with 3 rd /4 th swapped																										
4	alt - Altered scale																										
5	hm ⁵ - Harmonic minor scale from the 5 th																										
6	dor - Dorian minor (minor with #13)																										
7	min - Natural minor (aeolian)																										
8	hm - Harmonic minor (b6 but #7)																										
9	phr - Phrygian minor scale (with b9)																										
10	dim - Diminished scale (whole/half tone)																										
11	aug - Augmented scale (just whole tones)																										
select1			<p>Gate input for selecting the <i>root note</i> as being an allowed interval. When you want to create a playing interface for live operation you can patch the output of a toggle button (made with the circuit [button]) here.</p> <p>Note: When all select and selectfill inputs are 0, automatically all seven scale notes are selected, i.e. select1 ... select13 will be set to one.</p>																								
select3			Gate input for selecting the 3 rd .																								
select5			Gate input for selecting the 5 th .																								
select7			Gate input for selecting the 7 th .																								
select9			Gate input for selecting the 9 th (which is the same as the 2 nd).																								
select11			Gate input for selecting the 11 th (which is the same as the 4 th).																								
select13			Gate input for selecting the 13 th (which is the same as the 6 th).																								

Input	Type	Default	Description
selectfill1		off	Selects the alternative 9 th (i.e. the 9 th that is <i>not</i> in the scale).
selectfill2		off	Selects the alternative 3 rd (i.e. the 3 rd that is <i>not</i> in the scale).
selectfill3		off	Selects the alternative 4 th or 5 th . In most cases this is the diminished 5 th .
selectfill4		off	Selects the alternative 13 th (i.e. the 1 st 3 that is <i>not</i> in the scale).
selectfill5		off	Selects the alternative 7 th (i.e. the 7 th that is <i>not</i> in the scale).
tuningmode		off	While this is 1 , the circuit will output the value set by tuningpitch instead of the actual pitch. This is meant to be a help for tuning your VCOs.
tuningpitch	$\frac{1V}{Oct}$	0V	This pitch CV will be output while the tuning mode is active.
transpose	$\frac{1V}{Oct}$	0V	This value is being added to the output pitch when not in tuning mode. It can be used for musical transposition or adding a vibrato.

Output	Type	Description
output1 ... output4	$\frac{1V}{Oct}$	1 st ... 4 th pitch output

13.10 clocktool - Clock divider / multiplier / shifter

This circuit implements various clock modifications, such as a clock divider, a clock multiplier, a tool for changing the length of an incoming gate signal and a clock time shift.



Multiply and divide

Here is an example of a simple clock divider that divides the incoming clock by 7 (i.e. for 7 incoming clocks one outgoing clock is being produced).

```
[clocktool]
  clock    = I1 # patch a clock here
  output   = 01
  divide   = 7
```

This example doubles the speed of the clock by inserting one additional clock tick right in the middle between two incoming ones: right in the middle between

```
[clocktool]
  clock    = I1 # patch a clock here
  output   = 01
  multiply = 2
```

By using multiplication and division at the same time you can create rhythms like “two over three”:

```
[clocktool]
  clock    = I1 # patch a clock here
  output   = 01
  divide   = 3
  multiply = 2
```

Per default the outgoing clock has a duty cycle of 50%, which means that it is 50% of the time high and 50% of the time low – basically a symmetrical square wave. You can change this with the **dutycycle** input, e.g. to 20%:

```
[clocktool]
  clock    = I1 # patch a clock here
  output   = 01
  dutycycle = 20% # same as 0.2
```

Time shifting the clock

The input **delay** can be used to delay the clock signal. It needs a steady input clock to work. The possible range of **delay** is -1.0 ... 1.0. A value of **1.0** is equivalent of delaying each clock by exactly one cycle – which is pretty useless, since it results in the same output clock. But for example a value of **0.1** will delay the clock by 10%. Here is an example:

```
[clocktool]
  clock    = I1 # patch a clock here
  output   = 01
  delay    = 0.1 # same as 10%
```

Using a negative number will result in a clock that is always slightly *before* the original clock. This example shifts the output clock 10% *ahead* of the input clock:

```
[clocktool]
  clock    = I1 # patch a clock here
  output   = 01
  delay    = -0.1
```

Please note that this is *not* a trigger delay, since it requires a steady input clock. Otherwise funny and strange things can happen. Also it should be obvious, that shifting a clock ahead needs knowledge when exactly the next input clock tick will happen.

Feeding a trigger sequencer like the **algoquencer** (see page 80) with a shifted clock allows you to fine tune the exact timing of that voice. You can easily map the shift amount to a pot for tuning that live by ear:

```
[clocktool]
  clock    = I1 # patch a clock here
  output   = _SHIFTED_CLOCK
  delay    = P1.1 * 0.2 - 0.1 # limit to +/- 10%
```

```
[algoquencer]
  clock    = _SHIFTED_CLOCK
  ...
  ...
```

Please also have a look at **timing** (see page 273). That can do a similar thing but is also able to shift the timing differently for each beat in a sequence of several beats.

If you combine **delay** with **divide** or **multiply**, the delay is applied *first*. This means that the amount of delay is in relation to one *input clock cycle*. The delayed input clock is then run through the divider and multiplier. If you like it vice versa, split things up into two **clocktool** circuit, where the first one does the divide/multiply, feed that output into the second one and do the delaying there.

Gate length

Per default the length of the output gate is 10 ms - independently of the length of the input gate. You can change the gate length either with the jack **gatelength** and specify a fixed number of seconds, or by using **dutycycle**, which is a percentage of the *output* clock rate. Please note: if your gate length exceeds the time until the next output gate, both will be "joined" and thus no new gate will be emitted.

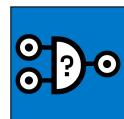
Please note if you use **dutycycle**: right at the start of the clock signal or after a greater speed change of the clock, **clocktool** needs a short time to learn the new clock speed and correctly adapt the new gate length. This might lead to two merging gates, which in turn causes a missing gate output.

Input	Type	Default	Description
clock			Patch a steady clock here for this circuit to be of any use
reset			A trigger here resets the internal counters. This is useful if you use the clock divider and want to restart the internal counting from 0, in order to align the clock divider with some external sequencers or the like
divide	1◦2◦3	1	Number to divide the clock through. This will be rounded to the nearest integer number. Note: if you want to use an external CV then you need to multiply that with some useful number, since otherwise you will get a number between 0 and 1 which is not useful at all. Remember: 10 V translates to a number of 1 .
multiply	1◦2◦3	1	Number to multiply the clock with. Same considerations hold as for divide .
dutycycle	0◦0.5◦1		Output duty cycle of the clock – which is essentially a square wave – in a range from 0.0 to 1.0 or 0% to 100% . If you don't patch anything here, the length of the trigger output pulses will be 10 ms (DROID's standard trigger duration).
gatelength			This jack is alternative to dutycycle and will override it if it is used. It sets the length of each output pulse to a fixed value that is independent of the incoming clock. A value of 0.5 (a CV of 5 volts) translates into a gate length of 0.5 seconds.
delay		0.0	<p>This CV allows you to shift the <i>input</i> clock beat around in time. A value of 0.1 will delay each beat by 10% of a clock cycle. A value of -0.1 is also allowed and shifts the beat 10% <i>ahead</i>.</p> <p>For an unmodulated delay -0.1 and 0.9 is just the same, because the output clock will have the same relation to the input clock. But if you <i>modify</i> the delay from 0.0 to 0.9, the next tick will be delayed by 90% of one cycle, where as a modification from 0.0 to -0.1 will play the next tick by 10% earlier.</p>

Output	Type	Description
output		Here comes the modified clock
inputpitch		Experimental output that outputs a representation of the input clock's pitch on a 1V/octave base, based on the reference of 60 BPM (1 Hz). This means that an input clock of 120 BPM will output 1V (a value of 0.1), since 120 BPM is one octave higher than 60 BPM. If you feed that value to the rate input of an LFO you get that running at exactly the same speed (not in the same phase, however).
outputpitch		Same for the modified output clock

13.11 compare - Compare two values

This simple utility circuit allows you to make a decision by comparing an input value (at **input**) against a reference value (at **compare**) and output one of three values depending on whether the input is less than, greater than or equal to the reference.



The following simple example checks if the pot **P1.1** is left of the center (a value less than 0.5). If that is so, it outputs **1**, otherwise **0**.

```
[compare]
  input = P1.1
  compare = 0.5
  ifless = 1
  output = 01
```

You can change the default output value of **0** with the input **else**. That specifies what happens if the condition is *not* met. The following example outputs **-1**, if **P1.1** is greater or equal to 0.5.

```
[compare]
  input = P1.1
  compare = 0.5
  ifless = 1
  else = -1
  output = 01
```

Equality, analog unprecision

You can also check if two values are *equal*. This is done with **ifequal**. Check this out:

```
[compare]
  input = B1.1
  compare = 1
  ifequal = 4
  else = 8
  output = 01
```

Now while you hold the button **B1.1** this circuit will output the value **4** and otherwise **8**.

Note: equality can be tricky when it comes to values from *analog* things like inputs or potentiometers. They always undergo tiny random fluctuations. So the following example, that should compare the current voltages of two inputs, will never really work:

```
[compare]
  input = I1
  compare = I2
  ifequal = 1 # will never happen!
  output = 01 # This won't work!
```

If you try this out, you will probably *never* get both inputs equal. Even a single electron too much could theoretically make the difference. So in order to make such comparisons possible, there is a way to allow for a *slight unprecision* when doing the comparison. This is set with the **precision** parameter:

```
[compare]
  input = I1
  compare = I2
  precision = 0.1
  ifequal = 1
  output = 01
```

Now the inputs **I1** and **I2** are being treated as equal as long as their difference is **0.1** (1 V) at most.

Makeing a three-way switch

It is possible to check all three relations at once. Make sure that you apply a **precision** if you deal with analog values:

```
[compare]
  input = I1
  compare = I2
  precision = 0.1
  ifless = 0
  ifequal = 1
  ifgreater = 2
  output = 01
```

Now you get **0**, **1** or **2**, depending on wether **I1** is less, equal or greater than **I2**.

Note: Better do not use just **ifless** and **ifgreater** without using **ifequal** or **else**. This lets the equality undefined and will output **0** if for any chance the two input values are equal. Better use **ifless** / **ifgreater** in combination with **else** if you are not interested in the exact equality.

Omitted inputs

It is allowed to omit any of the inputs **ifless**, **ifequal**, **ifgreater** or **else**. Any of these is treated as **0** with one exception: If you omit all four, **ifequal** defaults to **1**. This make a super basic **compare** circuit just check if two values are equal:

```
input = B1.1
compare = 0
output = 01
```

This will output **1** if button **B1.1** as the value **0** (is not pressed).

Dynamic output values

As often, instead of using fixed values for **ifless**, **ifequal**, **ifgreater** and **else** you can use dynamic values from somewhere else, of course. The following example will output a sine wave at **01** if the pot is left of the center or else a square wave:

```
[lfo]
    hz = 2
```

```
sine = _SINE
square = _SQUARE
```

```
[compare]
    input = P1.1
    compare = 0.5
    ifless = _SINE
    else = _SQUARE
    output = 01
```

Input	Type	Default	Description
input		0.0	A value to compare.
compare		0.0	A reference value to compare the input with.
ifgreater			Value to be output if input is greater than compare . If you patch nothing here, the value of the input else will be used.
ifless			Value to be output if input is less than compare . If you patch nothing here, the value of the input else will be used.
ifequal			Value to be output if input is equal to compare within the precision defined by precision . If you patch nothing here, the value of the input else will be used.
else		0.0	Specifies the output value in case none of the stated conditions are met.
precision		0.0	An optional precision to be used by ifequal

Output	Type	Description
output		Here one of ifgreater , ifless or ifequal is output.

13.12 contour - Contour generator

An enhanced version of the classic ADSR-envelope generator with the six phases predelay, attack, hold, decay, sustain and release.



For triggering there are two alternative inputs: **gate** and **trigger**. Use **trigger** if you are not interested in the length of the gate signal. There will be no decay / sustain phase in that case.

The minimal patch just connects **gate** or **trigger** and the output. It creates an envelope with standard timings, triggered at **I1** and output to **01**:

```
[contour]
  gate  = I1
  output = 01
```

Assigning pots to the classic four inputs lets you use the DROID just as a normal ADSR envelope:

```
[p2b8]
[p2b8]

[contour]
  gate  = I1
  attack  = P1.1 * 20
  decay   = P1.2
  sustain = P2.1
  release = P2.2
  output  = 01
```

When you try this out, you will notice that the time range of the **attack** parameter is much shorter than that of **decay** and **release**. In fact it is just $\frac{1}{20}$ of these. This has been chosen in this way because I believe that this

makes sense from a musical point of view. Very long attack times are quite unusual and I wanted to be able to directly map the four values to pots. But if you don't like that you can - of course - make all three timing parameters have the same range simply by multiplying attack by 20:

```
[p2b8]
[p2b8]
```

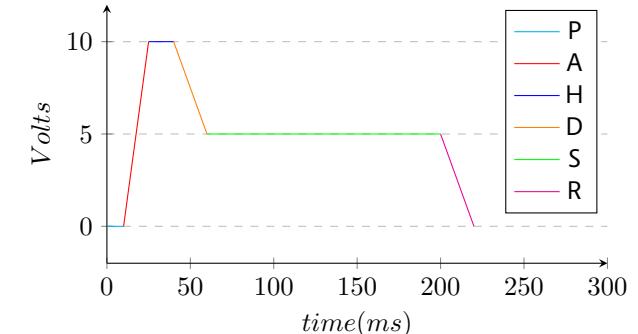
```
[contour]
  gate    = I1
  attack  = P1.1 * 20
  decay   = P1.2
  sustain = P2.1
  release = P2.2
  output  = 01
```

If you do not change the **shape** parameter, the duration of the attack phase is 0.1 sec at a value of 1. The phases decay and release have a duration of 2.0 sec at a value of 1.

The Phases

In addition to the traditional ADSR phases this circuit also has a an optional predelay (**P**) phase - which acts like a delay before the envelope starts - and an optional hold (**H**) phase which keeps the envelope at maximum level for a short time right after attack and before decay.

The following diagram shows an example envelope with all six phases. The gate starts at 0 ms and ends at 200 ms.



Attack, Decay and Release

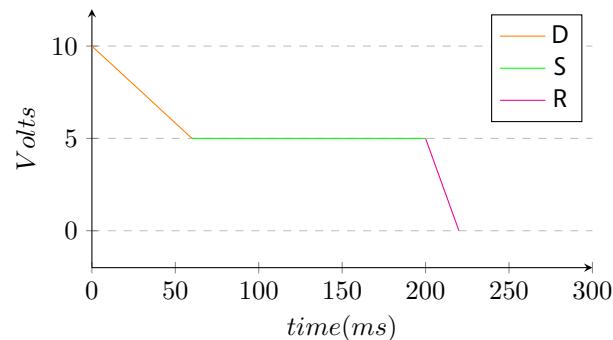
The phases attack, decay, release are phases where the level of the envelope starts at one level and then approaches another level within a certain time. In the upper example all these phases had a *linear* characteristic. That means that the output voltage changes by a constant amount per time.

DROID's **contour** allows you to control the shape of these phases in order to get them *bent* in either direction. For that purpose there are the inputs **attackshape**, **decayshape** and **releaseshape**.

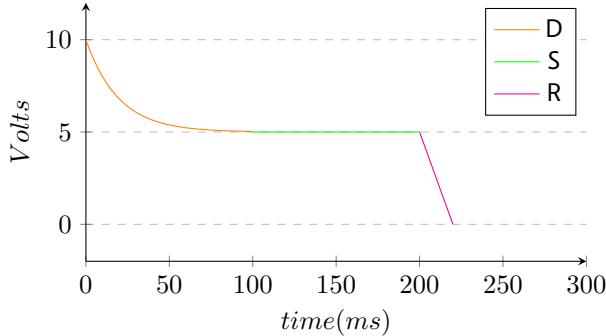
Let's take decay as an example. During the decay phase the envelopes voltage falls from the maximum level of 10 V (you can change this with the input **level**) to the sustain level defined by the input **sustain**. For simplicity let's assume that you have not used these inputs, so the maximum level is 10 V (**1.0**) and the sustain level is 5 V (**0.5**). Also we assume attack, predelay and hold to be **0.0**.

When **decayshape** is not patched or otherwise set to its

default of **0.5**, the shape of the decay curve is *linear*. This means that it goes down by the same voltage each second until it reaches **0.5**.



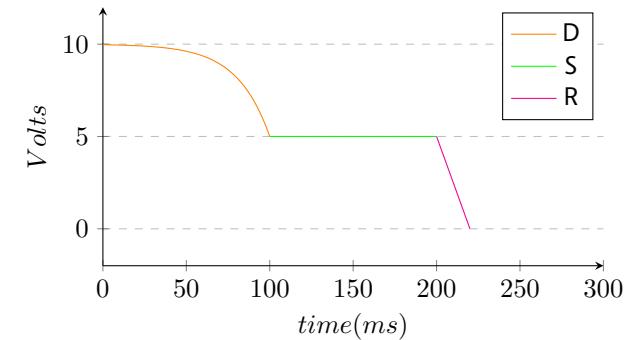
Now, if you set **decayshape** to **1.0**, the curve is completely *exponential*:



Such an envelope sounds completely different - of course also depending on whether you feed this into a linear VCA, exponential VCA or a VCF. For fine control you can use any number between **1.0** and **0.5** of course. In that case you will get a curve that is bent to a certain degree. Assigning **decayshape** to a pot helps you *listening* to the different sounds:

[contour]
gate = I1
decayshape = P1.1
output = O1

If the shape gets a value less than **0.5**, the curve is bent into the opposite direction (some call this *logarithmic* but mathematically this is not true). Here is an example where **decayshape** is set to **0.0**:



Input	Type	Default	Description
gate			Patch a gate signal here that triggers the envelope. Gate means that the length of the signal is relevant. While the gate is high the sustain phase holds on. As soon as gate is going low the release phase is being entered.
trigger			This is an alternative method of starting the envelope. If you use trigger instead of gate , there are the following differences: <ul style="list-style-type: none"> The duration of the trigger signal is being ignored. There is no decay / sustain phase. Attack and hold are immediately followed by release. The inputs sustain and decay have no impact anymore. The predelay and attack phases are continued until their end even when the trigger signal ends (When using gate and the gate signal ends during predelay, the envelope does not start. When it ends during attack, decay / sustain are being skipped and release starts at the current level of the envelope. That way short gates can result in "quieter" envelopes).

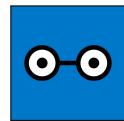
Input	Type	Default	Description
retrigger		1	If you patch 0 or off here, a gate or trigger impulse will not immediately restart the envelope unless it already has reached its release phase. The default on , which means that a trigger will immediately restart the envelope in any case.
startfromzero		0	If you set this to 1 or on , a trigger or gate will reset the envelope's current level immediately to zero. This is sometimes called "digital mode". In the normal analog mode the envelope resumes from where it is. This means that when a trigger occurs right in the release phase where the level is still high, will start it's attack not from zero but from this hight value.
abortattack		0	This is an on / off setting that decides what happens if the input gate goes off while the predelay or attack phase is still not finished. Per default that phase will be finalized regardless of the gate state. If abortattack is on , the end of the gate will immediately stop the attack phase and move on to hold. Note: In this case the value of the envelope will not reach the maximum level. If the gate ends during the predelay phase, no envelope will be started at all. Note: This setting is only functional when the gate input is being used for triggering the envelope. If you use trigger , the attack phase is always completely executed and this setting has no influence.
loop		0	This is an on / off input that switches loop on or off. When loop is on , the envelope will immediately start again once it has finished. It also starts without triggering. This converts contour into a kind of fancy LFO. gate / trigger and loop can be combined. Any gate or trigger will restart the envelope just as usual - even in loop mode.
predelay		0.0	The predelay phase inserts a delay between the incoming gate and the begin of the envelope. The length of the predelay is 0.1 seconds per volt, so a value of 1.0 means 1 second
attack		0.0	Length of the attack phase, i.e. the time from the beginning of the gate until the maximum level is reached. See the general description for information about the scaling of this input.
hold		0.0	If this is non-zero, the envelopes lingers a certain amount of time at its maximum level after the attack and before the decay phase. The input value specifies a number of seconds. A value of 0.5 (this is 5 V) will create a hold time of 0.5 seconds.
decay		0.2	Time of the decay phase
sustain		0.5	Sustain level
swell		0.0	If this jack is set to a value greater than 0.0 , the level of the envelope will go up or down again during the sustain phase until it reaches swelllevel .
swelltime		5.0	Time of the swell phase
swelllevel		1.0	Level the swell phase is approaching. Setting this to the same as sustain effectively disables swell.

Input	Type	Default	Description
release		0.2	Timing of the release phase
level		1.0	Maximum level and scaling of the envelope. It is basically an output attenuator of the envelope. Sudden changes in the level will immediately have an (audible) impact on the envelope.
velocity		1.0	<i>energy of the attack:</i> The velocity is similar to the level , but is effective just during the attack phase. During that phase that maximum voltage that is read from the velocity jack and will be used as the velocity of the envelope. Further changes during the other phases will be ignored. This makes it ideal of using with a sequencer. For example you can patch an <i>accent</i> output here and add some offset. Sudden changes in this input will not affect the shape of the envelope.
pitch		0V	This is a <i>one volt per octave</i> input affecting all timings of the envelope. When you set this to 0 (the default), it is neutral. A value of 0.1 (1 Volt) will exactly double the speed of all phases - just as one octave up doubles the frequency of an oscillator. This jack can be used to easily implement envelopes where the length very naturally follows this pitch - just like on a piano, glockenspiel or marimba lower notes last longer than higher ones.
taptempo			Tap tempo is an alternative method of specifying a pitch information. When you patch a clock to tap tempo, all time parameters in the envelope are relative to that clock. If the clock speeds up, the envelope gets faster and vice versa. The reference speed is 120 BPM. This means that if you patch a 120 BPM clock here, nothing changes. Clocks faster than 120 BPM will speed up the envelope. Clocks slower than 120 BPM will slow it down. Please see page 21 for details on using taptempo inputs.
shape		0.5	If you use this jack, it sets the shape for all of the relevant phases, which are attack, decay, swell and release. Note: this input is only effective for those phases where the dedicated input (like attackshape , etc.) is <i>not</i> being used.
attackshape			Shape of the attack curve. If nothing is patched here, the value of shape will be used. See the general description for how curve shapes work.
decayshape			Shape of the curve in the decay phase. If nothing is patched here, the value of shape will be used.
swellshape			Shape of curve during the swell phase. If nothing is patched here, the value of shape will be used.
releaseshape			Shape of the curve in the release phase. If nothing is patched here, the value of shape will be used.
zerocrossing			This is an experimental feature: If you patch the output of an oscillator here, an incoming gate or trigger signal will be delayed until the next zero crossing of that signal. That allows you to start the envelope exactly when the audio signal is at 0 and avoid nasty klicks, even if the attack is set to 0. It comes at a price, however. The delay between the trigger and the first zero crossing might vary a lot from note to note and that could make your rhythm untight, especially if the frequency of the oscillator is low.

Output	Type	Description
output		Main output of the envelope. Patch this to your filter, VCA or wherever you like.
negated		The negated output is the same as the output but in negative voltage.
inverted		The inverted output always outputs <i>positive</i> voltages but is inverted relative to the level of the envelope. When the normal output outputs 0 V, the inverted output outputs level and vice versa
endofpredelay		This output will emit a trigger with a length of 10 ms when the predelay phase has ended.
endofattack		This output will emit a trigger with a length of 10 ms when the attack phase has ended.
endofhold		This output will emit a trigger with a length of 10 ms when the hold phase has ended.
endofdecay		This output will emit a trigger with a length of 10 ms when the decay phase has ended.
endofrelease		This output will emit a trigger with a length of 10 ms when the release phase has ended.

13.13 copy - Copy a signal

This circuit is a simple utility that copies a signal from an input to an output. Since every input generally can be attenuated and offset this can be used for scaling and offsetting a signal on its path.



The following example outputs the sine wave of the same LFO to **01** and **02**, where **02** is being inverted. This is also an example of using an output as an input.

```
[lfo]  
hz = 0.5 * P1.1
```

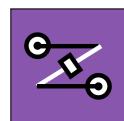
```
sine = 01  
[copy]  
input = 01  
inverted = 02
```

Input	Type	Default	Description
input	~~~	0.0	Connect the signal you want to copy here.

Output	Type	Description
output	~~~	The resulting signal will be sent here.

13.14 crossfader - Morph between 8 inputs

This utility circuit creates CV a controlled mix of two out of up to eight inputs. With two inputs this acts like a classical cross fader. The following example lets you fade between the signals at **I1** and **I2** by turning the pot **P1.1**:



```
[crossfader]
  input1 = I1
  input2 = I2
  fade   = P1.1
  output = O1
```

At fully CCW (**0.0**) only the signal of the first input is being output, at fully CW (**1.0**) only that of the second one. In the center position (**0.5**) you get the average of both inputs, namely $0.5 \times \mathbf{I1} + 0.5 \times \mathbf{I2}$.

Using more than two inputs is possible. The **fade** input then maps the range **0.0 ... 1.0** to a journey from the first

to the last input. Let's see the following example:

```
[lfo]
  hz      = 0.1
  sawtooth = _FADE
```

```
[crossfader]
  input1    = I1
  input2    = I2
  input3    = I3
  input4    = I4
  fade      = _FADE
  output    = O1
```

Now during one LFO cycle of 10 seconds the output **O1** begins with the signal at **I1** and then morphs to that of **I2**. It reaches 100% of **I2** at a fade value of $\frac{1}{3}$. Then it continues to **I3**, which it reaches at $\frac{2}{3}$ and finally - after 10 seconds - it ends at **I4**. After that it immediately jumps back to **I1**, in order to begin the next cycle.

Values beyond **1.0** for **fade** are allowed and allow you to morph from the last input to the first one. In the previous example that would be the range from **1.0** to **1.3333**. So if you scale up the sawtooth to a total range of **0.0 ... 1.3333** you will get a smooth cyclic morph between all four inputs:

```
[lfo]
  hz      = 0.1
  sawtooth = _FADE
```

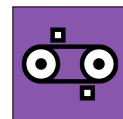
```
[crossfader]
  input1    = I1
  input2    = I2
  input3    = I3
  input4    = I4
  fade      = _FADE * 1.3333
  output    = O1
```

Input	Type	Default	Description
input1 ... input8		0.0	The input signals that you want to crossfade between. At least input1 and input2 need to be patched. Otherwise they are treated like 0 V signals.
fade		0.5	This value decides which of the two inputs should be mixed and to which degree each one should go into the mix. At 0.0 the mix consists of 100% of the first inputs, at 1.0 of 100% of the last patched input.

Output	Type	Description
output		Output of the mix

13.15 cvlooper - Clocked CV looper

Easy to use clocked CV looper that also loops an additional gate and can do overlay and overdub.



This circuit is a very easy to use CV looper. It records an incoming CV (and optionally a gate as well) on a virtual tape loop with a resolution of one sample per ms. The length of this tape is eight seconds. If you need a longer loop time, you can reduce the tape speed. At a speed of **0.5** you have a maximum loop time of 16 seconds and a resolution of one sample per 2 ms (which is still pretty decent for most applications).

This looper is meant to be playable in a live situation as easily as possible. For that purpose it does not implement the typical *loop start* → *loop stop* scheme - which requires the musician to know beforehand that she will start a loop. Instead the looper is *always* recording. The loop length is specified in *clock ticks*. And as soon as the looping is activated, the previous *x* clock ticks of CV information will be repeated over and over.

Here is an example for a simple looper for one CV without a gate:

```
[button]
    button    = B1.1
    led       = L1.1

[cvlooper]
    cvin     = I1
    clock    = I8    # steady clock
    cvout   = O1
    length   = 16   # 16 clock ticks
    loopswitch = L1.1
```

The button **B1.1** is converted into a toggle button for activating the looping. The CV is read from **I1** and is sent to

01. As long as the loop switch is **off** the looper is in bypass mode and simply copies **I1** to **O1**. At the same time it is always recording to its internal endless tape. When the loop switch is switched **on**, the last 16 clock ticks of CV information is looped to **O1** and **I1** is ignored.

Please note: for your convenience the exact time when the loop switch is switched **on** is *quantized to the nearest clock tick* - may it be in the future or past. This makes playing exactly in time much easier.

The second example adds a gate signal - such as output by a ribbon controller. The gate is running through **I2**→**O2**.

```
[button]
    button    = B1.1
    led       = L1.1

[cvlooper]
    cvin     = I1
    gatein   = I2
    clock    = I8    # steady clock
    cvout   = O1
    gateout  = O2
    length   = 16   # 16 clock ticks
    loopswitch = L1.1
```

Using a gate changes the behaviour of the CV looper. The state of **gatein** (not the exact voltage) is being looped as well. The CV is recorded to the tape *only while the gate is high*.

Using a gate makes two additional features possible:

1. When **overlay** is **on** and the input gate is active, the input CV will override that on the tape and instead the source signal from **cvin** is bypassed to

the output. The tape's content stays untouched. This allows you to overlay the loop CV with your own from time to time.

2. On the other hand, when **overdub** is **on** and the input gate is active, the input CV will be written to the tape and *replaces the recorded CV* at those places. And it also will be routed to the output at the same time.

Toggle buttons would fit nicely for these two functions.

Please note: you always need a clock! The CV looper is useless without one. If you do not want to use an external clock, you can make use of the LFO circuit for creating an internal clock.

What if you want to loop more than one CV? Just create more **cvlooper** circuits - one for each CV. And control them from the same set of buttons.

Changing the tape or clock speed

It is possible to change the tape speed on the fly in order to slow down or speed up the recorded loop's content. It is important - however - to always change the tape speed and clock speed *at the same time and in the same manner*. Otherwise you will get stuttering effects. So if you double the **tapespeed** you also need to double the frequency of the clock.

Changing the length

Changing **length** parameter on the fly is supported and just works. Remember: it does not set the length of the tape loop but just the length of that part that is played

back. The recording is always done with the maximum length. So if you *increase* the length while playing back you will get access to the older parts of the CV history that way. Just don't make the length longer than the actual tape (see below).

Limitations

Memory (RAM) is a valuable resource. The CV looper limits itself to 8000 samples in order not to waste too

much memory and leave space for other circuits as well (the Droid master has about 100.000 bytes of memory and 8000 samples need 16.000 bytes). But if you want to make longer loops, you can reduce the tape speed and thus use less samples per second.

A second limitation is that the total loop length can be 128 clock ticks at most. If you need more ticks, you can divide the input clock down, using **clocktool**:

[clocktool]

```
clock      = G1
divide    = 2
output    = _LOOP_CLOCK

[cvlooper]
clock      = _LOOP_CLOCK
cvin       = I5
tapespeed  = 0.2 # max loop five x longer
cvout     = 05
length    = 128 # = 256 original ticks
loopswitch = _SOME_BUTTON
```

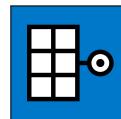
Input	Type	Default	Description
cvin		0.0	Input CV that should be looped.
gatein		1	Optional input gate. If you do not patch something here, the gate is assumed to be always high.
clock			Input clock. The clock is mandatory and is the base for the definition of the loop length. Also the loop switch is quantized in time to the nearest clock.
reset			A trigger here resets the playback head immediately to the start of the loop, if you are in playback mode.
length	1..2..3	16	Length of the loop in clock ticks. Example: You get a length of 16 ticks by patching the number 16 to length . If you want to set the length by means of an external CV that would require 160 Volts. So you need to multiply your input by some useful number in that case.
tapespeed		1.0	Relative tape speed, where 1.0 is the normal speed. So a value of 0.5 slows down the speed thus increasing the effective tape length from 8 to 16 seconds while reducing the sampling rate from 1 ms to 2 ms per sample. Changing the tape speed on the fly probably leads to interesting results.
loopswitch			Mandatory parameter: While the loop switch is off the CV looper simply sends all input CV and gate to their respective outputs. At the same time CV and gate are also recorded to the tape. When the loop switch is on , the CV and gate are being read from the tape, instead. The input CV and gate are now ignored.
pause		off	This is a binary input. If you send a high signal here, the looper pauses. This is only works in playback mode. The current CV value is hold the entire time. This is <i>not</i> the same as bypass, since in bypass mode the original CV will be routed through.
overlay		off	Overlaying changes the behaviour while looping is active. If overlay is set to on , while the input gate is active the gate and CV will be sent directly from the inputs rather than read from the tape.

Input	Type	Default	Description
overdub		off	Overdubbing also changes the behaviour during the looping: If it is active then while the input gate is high the input gate and CV will be written to the tape - thus changing the loop on the fly.
bypass		off	Setting bypass to on copies the input CV and gate from their inputs to their outputs <i>while keeping the loop's content untouched</i> . This disabled the looping for the while, but you can get back to it later. Note: this is different from turning off the loop switch, because then your tape's content would be overwritten.

Output	Type	Description
cvout		Output of the bypassed or looped CV
gateout		Output of the bypassed or looped gate

13.16 dac - DA Converter with 12 bits

This circuit converts a binary representation of up to 12 bits into an output value in a given range. Consider the following example:



```
[dac]
bit1 = I1
bit2 = I2
bit3 = I3
output = O1
```

In this example three bits are being used.

Three bits can represent a number from 0 to 7. These are mapped to the input range from 0 to 1 (or 0 V to 10 V) in the following way:

bit1	bit2	bit3	bit value	output
0	0	0	0	0.000
0	0	1	1	0.143
0	1	0	2	0.286
0	1	1	3	0.429
1	0	0	4	0.571
1	0	1	5	0.714
1	1	0	6	0.857
1	1	1	7	1.000

In other words: this circuit will convert three different gate inputs into one analog output value. **bit1** has the most influence, **bit3** the least.

The normal output range is 0 to 1 (i.e. 10 V) per default, but you can change that with the parameters **minimum** and **maximum**. For example you could have the three bits mapped to just the range of 0.1 to 0.5:

```
[dac]
bit1 = I1
bit2 = I2
bit3 = I3
minimum = 0.1 # 1V
maximum = 0.5 # 5V
output = O1
```

Now the table looks like this:

bit1	bit2	bit3	bit value	output
0	0	0	0	0.100
0	0	1	1	0.157
0	1	0	2	0.214
0	1	1	3	0.271
1	0	0	4	0.329
1	0	1	5	0.386
1	1	0	6	0.443
1	1	1	7	0.500

If you use more of the **bit**-outputs you get more resolution. For example if you use **bit1**... **bit8**, the total range will be divided into 256 possible output values. The maximum is 12 bits. Since bit 1 is the most significant bit, adding more and more bits will not change the influence of the already used bits.

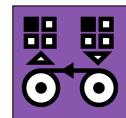
Please also have a look at the circuit **adc** (see page 78, which does the exact opposite!

Input	Type	Default	Description
bit1 ... bit12			The 12 bit input bits. bit1 is the MSB - the most significant bit. The LSB (least significant bit) is the highest input that you actually patch.
minimum		0.0	This sets the lower bound of the output range, i.e. the value that the bit sequence 000000000000 will produce.
maximum		1.0	This sets the upper bound of the output value, i.e. the value that the bit sequence 111111111111 will produce.

Output	Type	Description
output		Output signal.

13.17 delay - A tape delay for CVs, gates and numbers

Use this circuit to delay the movement CVs, gates or integer numbers in time. The usage is very simple. Feed input signals into the circuit, set a delay time and these signals are output again delayed by that time.



Note: This circuit is still experimental. In a future firmware version it might be changed or removed. Also the file format on the SD card for the saved recordings might change and a new version might not be able to load old recordings.
The basic usage of the delay is very simple:

```
[delay]
cvin = I1
cvout = 01
delay = 0.5
```

Here the signal from **I1** is output again at **01** with a delay of 0.5 seconds.

You can make the delay time depend on the speed of a clock signal. just feed a steady clock into **clock**. Now the **delay** parameter is measured in clock ticks - not in seconds anymore.

```
[delay]
cvin = I1
cvout = 01
clock = G1 # input clock
delay = 4 # delay by 4 ticks
```

Use as a trigger delay

Alongside the continuous CV, eight gate signals can be fed through the delay. Use **gatein1** ... **gatein8** and **gateout1**

... **gateout8** for this purpose:

```
[delay]
gatein1 = G1
gatein2 = G2
gatein3 = G3
gatein4 = G4
gateout1 = G5
gateout2 = G6
gateout3 = G7
gateout4 = G8
delay = 0.5
```

Now the gate patterns at the inputs **G1** through **G4** appears time shifted by 0.5 seconds at the outputs **G5** through **G8**.

Technical background and limitations

The two circuits **recorder** (see page 251) and **delay** (see page 138) are based on the same implementation of a virtual tape. This virtual tape has three tracks with three recording and playback heads:

1. One head for recording a continuous CV in the range $-1 \dots +1$ (which is $-10\text{ V} \dots 10\text{ V}$)
2. One head for recording eight gate tracks in parallel (CVs where just 0 and 1 is recorded)
3. One head for recording a discrete integer number in the range 0 ... 255

All these are recorded in parallel, so for example it's easy to record a CV/gate signal with just one **cvrecorder**. The discrete number is useful for recording the outputs of **buttongroup** (see page 108) circuits or the switches on the S10 similar things.

Note: The dynamic range of CV signal on the tape is just $-1 \dots +1$ (or $-10\text{ V} \dots 10\text{ V}$). Any "too hot" signal is clipped to that range. The internal resolution of the CV is 16 bit (precisely: one Volt is divided in 3200 steps). If you need a larger range, you need to divide the input signal and multiply the output signal by some factor, but lose a bit precision that way.

The track with the eight gates records just **0** and **1**. Any other value will be squeezed into that format: values below 0.1 (1 V) are considered **0**, all other values **1**.

In order to use the RAM of the **DROID** as efficient as possible (and allow for many multiple instances of these circuits), the tape uses just 256 samples. Each time the state of one of the gates or the value of the number changes, a new sample is created. A change in the input CV is handled more intelligently as the CV values of the samples are *interpolated* linearly. The maximum error between the interpolated value and the actual stored CV is limited to 0.0001 (which is 1 mV).

If the input CV is more chaotic, however, the number of samples per time is limited to an average of one sample every 20 ms, while short periods with up to 10 samples without this limitation are allowed. This ensures that the minimum recordable tape length is $256 \times 20\text{ ms}$, which is 5.12 seconds. Usually CVs are not so chaotic but either stepped or moving smoothly, so the recordings can be much longer.

If you have the special case of a stepped input CV - such as the output from a sequencer or from a CV/gate keyboard - you can switch to an alternative mode. Patch the gate output of the sequencer or keyboard into the **sample** input of the circuit. This enables the "triggered mode". Here a new sample is just and only created at each posi-

tive gate edge of the **sample** input. So the recordings can be as long as 256 notes.

Note: That way you would loose the gate length, since the end of the gate does not trigger a new sample. Use the **gatetool** (see page [160](#)) with the **inputgate** and **outputedge** to get one trigger at each edge and feed that into **sample**.

Saving the tape to disk

The **delay** does not support presets because of memory limitations. But you can save the current contents of the tape to your SD card. This is done by the two trigger inputs **save** and **load**, which are usually mapped to some buttons. Here is a simple example.

```
[delay]
  save = B1.5
  load = B1.6
  ...
```

If you hit button **B1.5**, the file **tape0001.bin** is created on your SD card. Button **B1.6** loads that file into the circuit.

You can use any file number from **1** to **9999** by using the parameter **filenumber**. You might want to map that to a rotary switch of an S10:

```
[delay]
  save = B1.5
  load = B1.6
  filenumber = S2.1
  ...
```

Note: Loading and saving is done in real time from/to your SD card. The files are very small, but the operation can take a small number of milliseconds. During that time no circuit will do its job. And if your SD card is missing, things lag a bit more due to timeouts.

One important difference to presets is that these files can be shared among circuits and even among different

patches. A recording of the **recorder** circuit can be loaded with every **recorder** or delay circuit.

Loading and saving

You might wonder, why this circuit offers loading and saving of the tape's content to the SD card. The reason is not because it's super useful but because **delay** uses the same tape implementation as **recorder** and saving is part of that.

When you load a file into the tape, its contents will be audible for a short time. But soon after the tape is overwritten with the new incoming data.

Saving might make more sense. You could make a snapshot of the tape's content and load that into a **recorder** (see page [251](#)) for playback. But even that doesn't seem to be game changing material.

Input	Type	Default	Description
delay		1.0	The CVs are delayed by this amount of seconds. If you patch clock as well, the delay is specified in clock tick, so then delay = 1 means "delay by one clock tick".
cvin		0.0	Continuous input CV
numberin	1 2 3		Discrete input number in the range 0 ... 255
gatein1 ... gatein8			Input gates
clock			If you use this clock input, all time inputs are measured in clock ticks instead of seconds.
sample			If you patch this input, "triggered" mode is enabled. In this mode, the virtual tape just records a new CV on each trigger at sample . So it just records stepped CVs, no slopes and no CV changes between the triggers.

Input	Type	Default	Description
timewindow		0.0	<p>When in triggered mode, this optional parameter helps tackling a problem that many hardware sequencers show: often their pitch CV is not at its final destination value at the time their gate is being output. Often you see a very short "slew" ramp of say 5 ms after the gate. During that time the pitch CV moves from its former to the new value.</p> <p>Now if you trigger the cvtape circuit with the sequencer's gate you will essentially sample the <i>previous</i> pitch CV instead of the new one. Or maybe something in between.</p> <p>The timewindow parameter configures a short time window after the trigger to trigger. During that time period the tape will constantly adapt the last sample to a changed input CV. When that time is over, the input is finally frozen on the tape.</p> <p>The timewindow parameter is in seconds. So when you set timewindow to say 0.005 (which means 5 ms), you give the input CV 5 ms time for settling to its final value after a trigger to sample before freezing it.</p>
bypass		off	Setting bypass to on copies the input signals directly to the outputs, regardless of any other stuff going on.
save			Send a trigger here to save the current contents of the tape to a file on the SD card. The filename is tapeXXXX.bin , where XXXX is replaced by the number set by filenumber .
load			Send a trigger here to load a previously saved file into the tape. Use filenumber so specify which file to load.
filenumber	1 ◊ 2 ◊ 3	1	Number of the file to load or save. The range is 0 - 9999. If filenumber is 123, the name on the SD card is tape0123.bin . These files are shared between all recorder and delay circuits.

Output	Type	Description
cvout		Output of the continuous input CV
numberout	1 ◊ 2 ◊ 3	Output of the discrete number
gateout1 ... gateout8		Output of the gates
overflow		When the internal memory of the tape is exceeded and data got lost, this gate goes to 1 for 0.5 seconds. If you are suspecting this situation, you can wire this output to an LED and observe the memory status that way.

13.18 droid - General DROID controls

This circuit gives access to some general DROID configuration settings. It does not make sense to create more than one instance of this.



The **droid** circuit gives you access to miscellaneous functions that affect the system as a whole. The most commonly used functions are that for lowering the brightness of the LEDs on the master, G8 and X7 via

ledbrightness and that of reducing the force feedback power of virtual notches of the motor faders of the M4. This is done with **m4notchpower**.

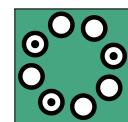
Input	Type	Default	Description
ledbrightness		1.0	Let's you dim all of the 24 LEDs of the master and the G8. This is mainly for those who think they are too bright. But since this parameter can be CV-controlled, you could of course also do funny things with it. Beware: if you set this to zero, the LEDs will be completely dark. This also includes possible error messages.
maxslope1 ... maxslope8		0.25	<p>Sets a threshold for a voltage change between two samples until the internal logic of the DROID outputs assumes that this step is intentional and should not be smoothed out. A typical case where you do not want smoothing is the pitch output of a sequencer.</p> <p>The default value is 0.25. A value of 0.0 turns off smoothing altogether since the slightest voltage change is considered an intentional jump.</p>
lpfilter1 ... lpfilter8		0.25	<p>Configures a digital low pass filter on the output in order to smooth out digital noise resulting from the DROID's main loop. This loop is running somewhere between 3 and 6 kHz - depending on the number of circuits you use.</p> <p>Per default this filter is set to 0.25 - which means a mild filtering - thus still allowing fast and snappy envelopes and other rapidly changing signals while filtering away most of the digital artefacts.</p> <p>If you use an output for a slow envelope that is combined with an audio path in a way that you hear digital artifacts then increase that value. This is e.g. the case if you modulate a VCA that in turn modulates a very low pitched audio wave with very few harmonics (such as a sine or triangle wave).</p> <p>The maximum value of 1.0 leads to a very strong filtering - i.e. removing all fast transients. Snappy envelopes will be smoothed out heavily. Square wave LFOs will be converted into lower level almost sine waves.</p>
m4faderspeed			Set the force / speed of the motor faders. Faster speeds need more electrical power and might wear off the faders faster. Too slow speeds might lead to poor operation. This value goes from 0.0 (slowest possible speed) to 1.0 (maximum speed). If you don't use this parameter, some reasonable default is used that depends on the firmware of the M4 module.

Input	Type	Default	Description
<code>m4notchpower</code>		<code>0</code>	Set the force feedback power of the M4 motor fader units when they operate with virtual notches. The range is from 0 (minimum notch power) to 1 (maximum notch power). Note: 0 does not turn the notches off, there is still some minimal feedback. If you don't use this parameter, the notch force feedback operates at some default power, which is dependent on the M4 firmware version.
<code>calibrate</code>			Immediately enter the calibration procedure, that's contained in the maintenance menu. Skips the menu. After calibration is done, resets.
<code>startx7upgrade</code>			Immediately starts the X7 firmware upgrade procedure (which is located at position 8 of the maintenance menu). After the upgrade of the X7 resets the master.
<code>clear</code>			<p>A trigger here sends a trigger to the <code>clear</code> input of all circuits that support this. That brings the state of those circuits to their start state. Circuits that have presets do keep those presets untouched. Just the current state is affected.</p> <p>That trigger is not sent to circuits whose <code>clear</code> input is patched.</p> <p><i>Note:</i> Just that part of the state is affected that is saved to the SD card. For example the <code>algoquencer</code> (see page 80) does not reset to the first step, it just clears its current pattern.</p>
<code>clearall</code>			<p>A trigger here sends a trigger to the <code>clearall</code> input of all circuits that support this. That's like a global factory reset for all of your circuits. Everything is set to its starting state, including all presets of those circuits.</p> <p>That trigger is not sent to circuits whose <code>clearall</code> input is patched.</p> <p><i>Note:</i> Just that part of the state is affected that is saved to the SD card. For example the <code>algoquencer</code> (see page 80) does not reset to the first step, it just clears its current pattern.</p>

13.19 euklid - Euclidean rhythm generator

This circuit creates trigger patterns according to the well-known *Euclidean rhythms* and is of course CV controllable. The pattern is described by three numbers:

- The number of steps in the pattern
- The number of beats in the pattern
- An offset for shifting the beats forward



The number of beats are distributed as evenly as possible in the pattern - but of course are all placed precisely on clock beats. Here are a few examples of various patterns:

length: 16, beats: 4, offset: 0



length: 16, beats: 5, offset: 0



length: 16, beats: 5, offset: 1



length: 16, beats: 11, offset: 0



length: 13, beats: 5, offset: 0



length: 13, beats: 5, offset: 1



length: 4, beats: 2, offset: 1



Here is an example without CV control:

```
[euklid]
  clock  = G1
  reset  = G2
  length = 16
  beats   = 5
  offset  = 0
  output  = G3
```

Now let's change that in order to make the beats controllable by the pot P1.1. Please note how the pot range is being changed from the default 0 ... 1 to the necessary 1 ... 16 by using a factor of 15 and an offset of 1:

```
[euklid]
  clock  = G1
  reset  = G2
  length = 16
  beats   = P1.1 * 15 + 1
  offset  = 0
  output  = G3
```

By the way: Since the default for **length** is 16 and for **offset 0** you can drop those two lines if you like:

```
[euklid]
  clock  = G1
  reset  = G2
  beats   = P1.1 * 15 + 1
  output  = G3
```

Offbeats

The output **offbeats** does the exact opposite of **outputs**: it triggers at those clock beats where **output** does not. So at any given clock tick exactly either **output** or **offbeats** triggers.

Gate length

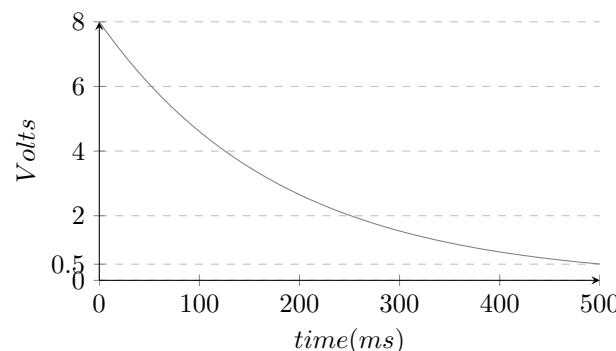
The length of the output gate is the same as that of the input gate. Also the exact voltage from the input is copied to the output while the current step is active.

Input	Type	Default	Description
clock			Patch a clock signal here. It does not need to be steady - even if this is the most usual application. Note: this input is classified as a gate input, since the length of the gate is being preserved when forwarded to output and offbeats .
reset			A trigger here resets the pattern to the start
outputsignal			Usually on active steps euklid just lets the original input clock get through to the output. If this parameter is used, it will be sent to the output on active steps instead. The easiest application is just setting it to 1 . The output will then become 1 the whole time while the current step is active. This is useful if you want to use euklid as modulation CV rather than as trigger.
length	1 ◦ 2 ◦ 3	16	The length of a pattern. This is interpreted as an integer number, which must be greater than 0. If it is not then 1 is assumed. If you CV control the length, use multiplication. The maximum accepted length is 64.
beats	1 ◦ 2 ◦ 3	5	The number of active beats that should be distributed amongst the length steps. If that number is greater than length , it is capped to that number.
offset	1 ◦ 2 ◦ 3	0	rotates or shifts the pattern by that number of steps. This number can be positive or negative.

Output	Type	Description
output		Output of the beats in the current pattern. The gate length is directly taken from the input clock - just as the voltage.
offbeats		Here those impulses will be output where there is <i>no</i> beat in the pattern.

13.20 explin - Exponential to linear converter

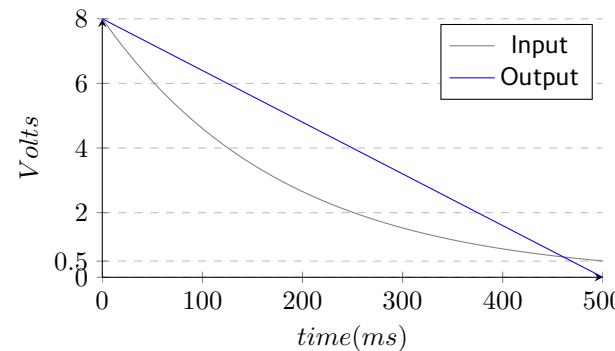
This circuit converts an exponential input curve into a linear output curve. Imagine you have an analog envelope outputting an exponential curve like the following one:



The curve starts at 8 V and reaches 0.5 V at about 500 ms later.

The following droid patch will convert this into a linear curve:

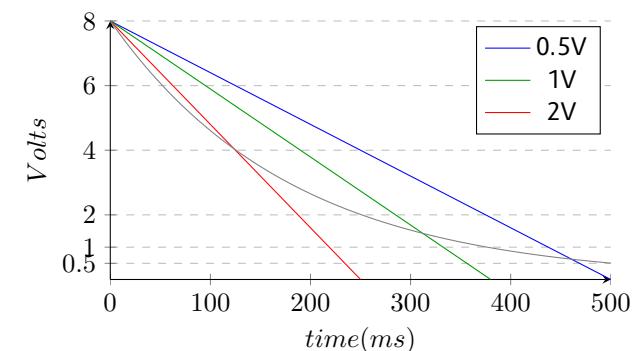
```
[explin]
  input    = I1
  output   = O2
  startvalue = 8V
  endvalue  = 0.5V
```



With the values **startvalue** and **endvalue** you configure how this translation is scaled. The **startvalue** selects the voltage where the exponential input curve and the linear output curve should be the same. If the input is

an envelope voltage then **startvalue** would be the start or maximum voltage of that envelope.

A falling exponential curve will never reach 0 in theory. So with **endvalue** you set a value (or voltage) in that you consider the curve to be low enough to be inaudible. At that voltage the linear output will exactly be zero. This voltage can be used to control the slope of the linear output curve. The following example shows how different values of **endvalue** affect the output:



Input	Type	Default	Description
input		0.0	Patch an exponential envelope output or a similar signal here. This value must be positive or otherwise it will be set to 0.0 .
startvalue		1.0	The assumed maximum value of the input signal (the start voltage from where it decays in an exponential way).
endvalue		0.01	The value at which it is assumed to be zero (at which the linear output will be set to zero. This value must be positive. It is forced to be ≥ 0.001 .
mix		1.0	Sets the mix between the "dry" and "wet" signal: At 0.0 the output is the same as the input. At 1.0 the output is the linear curve. At a value in between it is some average. You are even allowed to use values > 1.0 . A value of 2.0 will overcompensate and bend the curve beyond linearity into a curve some modularists would call <i>logarithmic</i> .

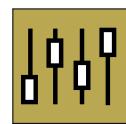
Output	Type	Description
output	~V~	Here comes the resulting linear output

13.21 faderbank - Create multiple virtual faders in M4 controller

This circuit is very similar to **motorfader** (see page 226) but controls up to 16 faders at once. Its purpose is to reduce the number of **motorfader** circuits in situations where you control banks or arrays of parameters in a similar way. It does not add any extra functionality to **motorfader**.

That being said, it is easiest to just show the differences to a single **motorfader** circuit. And these are:

- Instead of **fader** you set **firstfader** to specify which faders you want to control. The number of faders does not need to be set since it corresponds to the number of output jacks you use.



- Instead of **output** you have **output1**, **output2** and so on. This determines the number of faders that are controlled by this circuit.
- The parameters **notches** and **ledcolor** are common for all controlled faders. They are identical as those in **motorfader**.
- The parameters **ledvalue1**, **ledvalue2**, ... can set the brightness of the individual LEDs below the faders.
- Because of memory limitations you only have 6 presets (**motorfader** has 8).

Here is an example of a fader bank of the three faders 3, 4 and 5 (spreading over two M4s). We use a pot to se-

lect one of six presets (from 0 to 5). Turning the pot will immediately switch the preset (and the faders will move accordingly). And the CVs will be sent to outputs **01**, **02** and **03**:

[p2b8]

[m4]

```
[faderbank]
  preset = P1.1 * 6
  output1 = 01
  output2 = 02
  output3 = 03
```

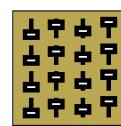
Input	Type	Default	Description
firstfader	1°2°3	1	First M4 fader of the virtual fader bank (starting with 1).
notches	1°2°3		Number of artificial notches. 0 disables the notches. 1 creates a pitch bend wheel. 2 creates a binary switch with the output values 0 and 1 . Higher number create that number of notches. E.g. 8 creates eight notches and output will output one of the value 0 , 1 , ... 8 . The maximum number of notches is 201 . But if you select more than 25 notches, the force feedback is turned off as the notches would get too small to work.
startvalue	~~~	0.0	This sets the value the faders should get when the circuit starts for the first time or when you send a trigger to clear .
ledcolor	~~~		When you use this input, it will set the color of the LED below the faders, when the circuit is selected. If the LED is off, this setting has now impact.
ledvalue1 ... ledvalue16	~~~		When you use this input, it will override the brightness of the LEDs below the faders, but just when this circuit is selected.
select	_____	☒	The select input allows you to overlay buttons and LEDs with multiple functions. If you use this input, the circuit will process the buttons and LEDs just as if select has a positive gate signal (usually you will select this to 1). Otherwise it won't touch them so they can be used by another circuit. Note: even if the circuit is currently not selected, it will nevertheless work and process all its other inputs and its outputs (those that do not deal with buttons or LEDs) in a normal way.

Input	Type	Default	Description
selectat	1 o 2 o 3		This input makes the select input more flexible. Here you specify at which value select should select this circuit. E.g. if selectat is 0 , the circuit will be active if select is <i>exactly 0</i> instead of a positive gate signal. In some cases this is more convenient.
preset	1 o 2 o 3		This is the preset number to save or to load. Note: the first preset has the number 0 , not 1 ! For the whole story on presets please refer to page 19 . This circuit has 6 presets, so this number ranges from 0 to 5 .
loadpreset			A trigger here loads a preset. As a speciality you can use the trigger for selecting a preset at the same time.
savepreset			A trigger here saves a preset.
clear			A trigger here loads the default start state into the circuit. The presets are not affected, unless you use direct preset switching with the preset input and without triggers. And that case the current preset is also cleared.
clearall			A trigger here loads the default start state into the circuit and into all of its presets.
dontsave		0	If you set this to 1 , the state of the circuit will not saved to the SD card and not loaded from the SD card when the Droid starts.

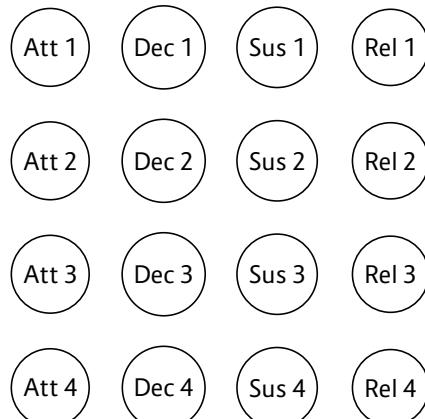
Output	Type	Description
output1 ... output16		Outputs the current value if the virtual motor faders that these outputs.
button1 ... button16		Outputs the current value of the touch buttons of the faders to these output which this circuit is selected.

13.22 fadermatrix - Matrix of up to 4x4 virtual motor faders

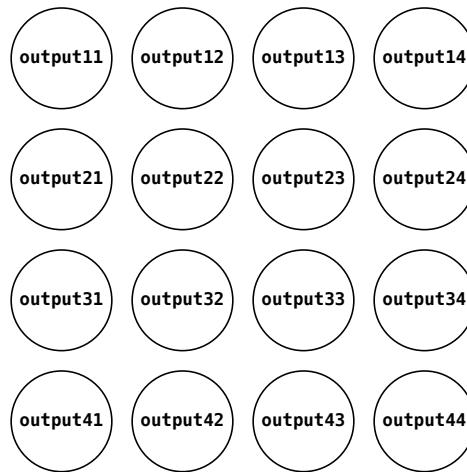
This circuit is a clever way of controlling a four by four matrix of parameters, which allows you to select either a row or a column.



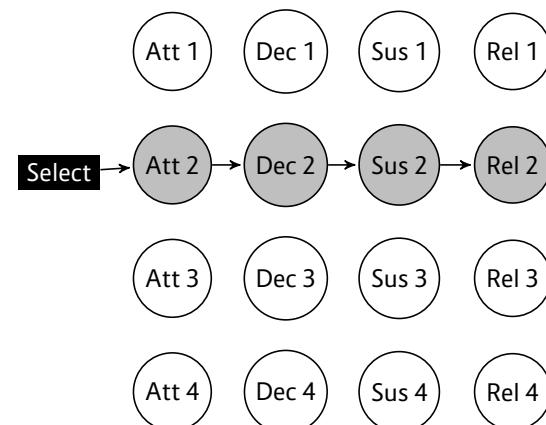
As an example let's think of a bank of four envelope generators. Each of them has the settings attack, decay, sustain and release (ADSR). That nicely forms a 4×4 matrix:



The **fadermatrix** has 16 outputs that map to these matrix positions:

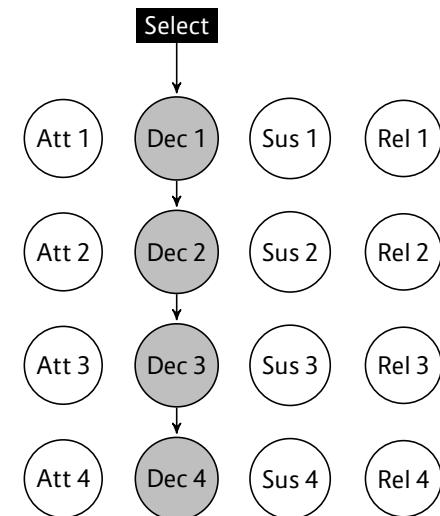


Now when you design a patch for controlling these 16 parameters with 4 motor faders you basically have the choice of selecting *rows* or *columns*! One way would be to always select one of the envelopes to be displayed and edited on your faders, for example the second one:



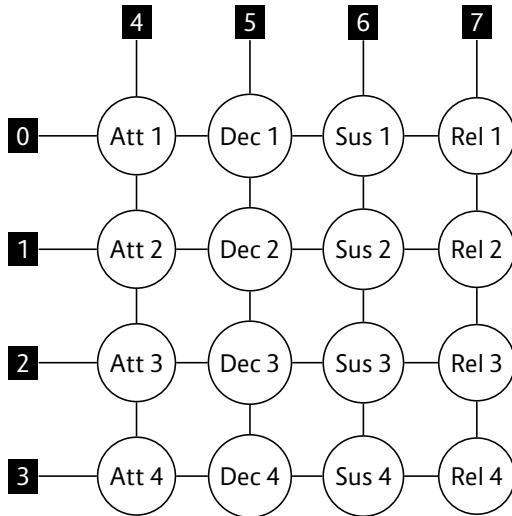
An alternative would be to have control over all decay

parameters of the four envelopes - in order to shape for synth voices at the same time without switching between those:



With **faderbank** you would have to decide for one of the two options. But with **fadermatrix** you can have both at the same time.

With the **rowcolumn** input you can select each column and each row as follows:



If you create a **buttongroup** with eight buttons and patch the output to the **rowcolumn** input, you have access to all four rows and columns. The nice thing about the **buttongroup** is that it automatically outputs the values from 0 to 7. Here is an example:

```
[p2b8]
[m4]

[buttongroup]
button1 = B1.1
button2 = B1.3
button3 = B1.5
button4 = B1.7
button5 = B1.2
button6 = B1.4
button7 = B1.6
button8 = B1.8
led1 = L1.1
led2 = L1.3
led3 = L1.5
led4 = L1.7
led5 = L1.2
```

```
led6 = L1.4
led7 = L1.6
led8 = L1.8
output = _ROWCOLUMN
```

Now we add a **fadermatrix**. We send all 16 outputs to intercal patch cables to be picked up later by four **contour** circuits:

```
[fadermatrix]
rowcolumn = _ROWCOLUMN
output11 = _ATTACK_1
output12 = _DECAY_1
output13 = _SUSTAIN_1
output14 = _RELEASE_1
output21 = _ATTACK_2
output22 = _DECAY_2
output23 = _SUSTAIN_2
output24 = _RELEASE_2
output31 = _ATTACK_3
output32 = _DECAY_3
output33 = _SUSTAIN_3
output34 = _RELEASE_3
output41 = _ATTACK_4
output42 = _DECAY_4
output43 = _SUSTAIN_4
output44 = _RELEASE_4
```

And here is the example for the first contour (the other three are similar):

```
[contour]
gate = I1
attack = _ATTACK_1
decay = _DECAY_1
sustain = _SUSTAIN_1
release = _RELEASE_1
output = 01
```

If you don't want to waste 8 buttons for just switching, you can also use a pot and scale it to the range of 0 ... 7:

```
rowcolumn = P1.1 * 7
```

And of course the rotary switch of an S10 would also be a perfect match, since it outputs exactly the number from 0 to 7.

Notches

As discussed in **motorfader** (see page 226), faders can set to have artificial notches. Also in the fader matrix you can set notches. Here the idea is that every parameter in the same *column* of the matrix has the same number of notches. Example:

```
notches3 = 8
```

This sets all four parameters in column 3 to have eight notches. This affects the four outputs **output13**, **output23**, **output33** and **output43** so that they get notches when selected and also change their output behaviour to outputting one of the values 0, 1, 2 ... 7.

As you can see the matrix always assumes that you edit four similar *things* with four parameters each. Every row of the matrix is one such thing. Every column is one parameter.

Smaller matrices

You also can create smaller matrices, for example 3x3. Simply omit the outputs **output14**, **24**, **34**, **44**, **41**, **42** and **43** in that case. Also 2x2 is possible.

Because we always need to be able to swap rows and columns, those numbers always have to be identical. So you cannot create a 3×4 matrix, for example.

Larger matrices

If you have eight faders, you can create even larger matrices. A 8×8 matrix can be created by four **fadermatrix**

circuits. Here you need some extra logic. At any time exactly two of the circuits must be selected. Use the **select** inputs in combination with **rowcolumn** in order to set this up (left as an exercise); -)

Input	Type	Default	Description
firstfader	1◦2◦3	1	First M4 fader of the virtual fader matrix (starting with 1).
rowcolumn	1◦2◦3	0	Currently selected row or column as follows:

0	Control output11, output12, output13 and output14
1	Control output21, output22, output23 and output24
2	Control output31, output32, output33 and output34
3	Control output41, output42, output43 and output44
4	Control output11, output21, output31 and output41
5	Control output12, output22, output32 and output42
6	Control output13, output23, output33 and output43
7	Control output14, output24, output34 and output44

Input	Type	Default	Description												
notches1 ... notches4	1°2°3	0	<p>Number of artificial notches in the respective column. For example notches2 controls the notches of output12, output22, output32 and output42.</p> <table border="1" data-bbox="819 298 1298 589"> <tr><td>0</td><td>disables the notches</td></tr> <tr><td>1</td><td>creates a pitch bend wheel</td></tr> <tr><td>2</td><td>creates a binary switch</td></tr> <tr><td>3</td><td>creates a switch with four positions</td></tr> <tr><td>8</td><td>creates eight notches</td></tr> <tr><td>25</td><td>creates 25 notches</td></tr> </table> <p>Enabling notches also changes the output value. When you have two or more notches, the output values become discrete. For example with four notches the output will be 0, 1, 2 or 3.</p> <p>Note: The maximum number of notches is 201. But if you select more than 25 notches, the force feedback is turned off as the notches would get too small to work.</p>	0	disables the notches	1	creates a pitch bend wheel	2	creates a binary switch	3	creates a switch with four positions	8	creates eight notches	25	creates 25 notches
0	disables the notches														
1	creates a pitch bend wheel														
2	creates a binary switch														
3	creates a switch with four positions														
8	creates eight notches														
25	creates 25 notches														
startvalue1 ... startvalue4	~~~		<p>These inputs allow to set a defined start value for each column. When the DROID starts first and there is either no saved state or state saving is disabled via dontsave = 1, these start values are used. Also a trigger to clear loads the start avlues. There is one start value for each column. All rows share the same start value for a column.</p>												
ledvalue11 ... ledvalue14	~~~		<p>With these inputs you can address the LEDs below the virtual faders of output11 ... output14. As opposed to using direction (e.g. L1.1), these inputs will only affect the LED if the according output is selected.</p>												
ledvalue21 ... ledvalue24	~~~		<p>With these inputs you can address the LEDs below the virtual faders of output21 ... output24. As opposed to using direction (e.g. L1.2), these inputs will only affect the LED if the according output is selected.</p>												
ledvalue31 ... ledvalue34	~~~		<p>With these inputs you can address the LEDs below the virtual faders of output31 ... output34. As opposed to using direction (e.g. L3.2), these inputs will only affect the LED if the according output is selected.</p>												
ledvalue41 ... ledvalue44	~~~		<p>With these inputs you can address the LEDs below the virtual faders of output41 ... output44. As opposed to using direction (e.g. L4.2), these inputs will only affect the LED if the according output is selected.</p>												
ledcolor1 ... ledcolor4	~~~		<p>Sets the color of the LEDs below the faders if ledvalueXY is used. There are just four inputs since every column of outputs has the same LED color (in order to identify them). The color works as with the R registers for the LEDs on the master module.</p>												

Input	Type	Default	Description
select			The select input allows you to overlay buttons and LEDs with multiple functions. If you use this input, the circuit will process the buttons and LEDs just as if select has a positive gate signal (usually you will select this to 1). Otherwise it won't touch them so they can be used by another circuit. Note: even if the circuit is currently not selected, it will nevertheless work and process all its other inputs and its outputs (those that do not deal with buttons or LEDs) in a normal way.
selectat			This input makes the select input more flexible. Here you specify at which value select should select this circuit. E.g. if selectat is 0 , the circuit will be active if select is <i>exactly</i> 0 instead of a positive gate signal. In some cases this is more convenient.
preset			This is the preset number to save or to load. Note: the first preset has the number 0 , not 1 ! For the whole story on presets please refer to page 19. This circuit has 6 presets, so this number ranges from 0 to 5 .
loadpreset			A trigger here loads a preset. As a speciality you can use the trigger for selecting a preset at the same time.
savepreset			A trigger here saves a preset.
clear			A trigger here loads the default start state into the circuit. The presets are not affected, unless you use direct preset switching with the preset input and without triggers. And that case the current preset is also cleared.
clearall			A trigger here loads the default start state into the circuit and into all of its presets.
dontsave		0	If you set this to 1 , the state of the circuit will not saved to the SD card and not loaded from the SD card when the Droid starts.

Output	Type	Description
output11 ... output14		Outputs for the CV values of the first row of parameter.
output21 ... output24		Outputs for the CV values of the second row of parameter.
output31 ... output34		Outputs for the CV values of the third row of parameter.
output41 ... output44		Outputs for the CV values of the fourth row of parameter.
button11 ... button14		Give access to the state of the touch button below the faders when the respective output in the first row is selected.
button21 ... button24		Give access to the state of the touch button below the faders when the respective output in the second row is selected.
button31 ... button34		Give access to the state of the touch button below the faders when the respective output in the third row is selected.
button41 ... button44		Give access to the state of the touch button below the faders when the respective output in the fourth row is selected.

13.23 firefacecontrol - Control a RME Fireface interface (experimental)

This experimental circuit allows you to control the most import volumes and mixes of an RME Fireface audio interface. It's also a perfect match for the M4 motor fader units. You need an X7 in order to use this circuit.

Please note that this circuit is still experimental. Its main



problem is that the MIDI implementation of the Fireface is more designed for user interaction via a Mackie Control and not for general automation. This is very sad.

For example there is a MIDI CC for changing the panning of a channel. But instead of simply having the CC going from 0 (left) via 64 (mid) to 127 (right), it uses various CC values as *commands* for modifying the existing panning

by some fixed value to the left or to the right. So without *knowing* the current setting, it's not possible to send the correct CC commands. And for DROID there is no way to know, since MIDI is a one way communication.

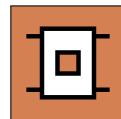
RME: if you are reading this: please contact me so that we can fix this.

Input	Type	Default	Description
<code>outputlevel1 ...</code>	<code>o ↕ 1</code>		
<code>outputlevel16</code>			
<code>mainoutput</code>	<code>1 ◊ 2 ◊ 3</code>	<code>1</code>	
<code>phonesoutput1,</code>			
<code>phonesoutput2</code>			
<code>outputmix1in1 ...</code>	<code>o ↕ 1</code>		
<code>outputmix1in16</code>			
<code>outputmix2in1 ...</code>	<code>o ↕ 1</code>		
<code>outputmix2in16</code>			
<code>outputmix3in1 ...</code>	<code>o ↕ 1</code>		
<code>outputmix3in16</code>			
<code>outputmix4in1 ...</code>	<code>o ↕ 1</code>		
<code>outputmix4in16</code>			
<code>outputmix5in1 ...</code>	<code>o ↕ 1</code>		
<code>outputmix5in16</code>			
<code>outputmix6in1 ...</code>	<code>o ↕ 1</code>		
<code>outputmix6in16</code>			
<code>outputmix7in1 ...</code>	<code>o ↕ 1</code>		
<code>outputmix7in16</code>			
<code>outputmix8in1 ...</code>	<code>o ↕ 1</code>		
<code>outputmix8in16</code>			

Input	Type	Default	Description
<code>outputmix9in1 ...</code>			
<code>outputmix9in16</code>			
<code>outputmix10in1 ...</code>			
<code>outputmix10in16</code>			
<code>outputmix11in1 ...</code>			
<code>outputmix11in16</code>			
<code>outputmix12in1 ...</code>			
<code>outputmix12in16</code>			
<code>outputmix13in1 ...</code>			
<code>outputmix13in16</code>			
<code>outputmix14in1 ...</code>			
<code>outputmix14in16</code>			
<code>outputmix15in1 ...</code>			
<code>outputmix15in16</code>			
<code>outputmix16in1 ...</code>			
<code>outputmix16in16</code>			
<code>postfader1 ...</code>			
<code>postfader16</code>			
<code>pan1 ... pan16</code>			
<code>unmute1 ... unmute16</code>			
<code>update</code>			
<code>select</code>			The select input allows you to overlay buttons and LEDs with multiple functions. If you use this input, the circuit will process the buttons and LEDs just as if select has a positive gate signal (usually you will select this to 1). Otherwise it won't touch them so they can be used by another circuit. Note: even if the circuit is currently not selected, it will nevertheless work and process all its other inputs and its outputs (those that do not deal with buttons or LEDs) in a normal way.
<code>selectat</code>	<code>1 · 2 · 3</code>		This input makes the select input more flexible. Here you specify at which value select should select this circuit. E.g. if selectat is 0 , the circuit will be active if select is <i>exactly</i> 0 instead of a positive gate signal. In some cases this is more convenient.

13.24 flipflop - Simple flip flop

This circuit implements a flip flop that stores one bit, which can be manipulated with various triggers.



Here is a simple example for a flip flop that toggles at each trigger. Fun fact: this implements a clock divider by two:

```
[flipflop]
  toggle = I1
  output = O1
```

As an alternative you can work with **set** and **reset** triggers:

```
[flipflop]
  set = I1
  reset = I2
  output = O1
```

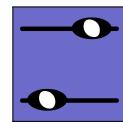
Note: The flip flop does not save its state to the SD card. And it has no presets. If you need any of these, have a look at **button** (see page [103](#)).

Input	Type	Default	Description
toggle			A trigger here inverts the state of the flip flop. It changes 0 to 1 and 1 to 0 .
set			Sets the flip flop to 1 .
reset			Sets the flip flop to 0 .
clear			Sets the flip flop to the value defined by startvalue .
startvalue		0	The flip flop starts its live with this value. Also clear will set the flip flop to this value.
load			Loads the value into the flip flop that's defined with loadvalue .
loadvalue		1	Value to set the flip flop to, when load is triggered.

Output	Type	Description
output		Outputs the current value of the flip flop: either 0 or 1 .

13.25 fold - CV folder - keep (pitch) CV within certain bounds

This circuit can keep an incoming CV within defined bounds, but not by limiting to these bounds, but by *folding* it in case it exceeds these bounds.



The main application is keeping the pitch of a voice within a certain range by octaving it up and down when necessary. Octaving keeps the actual note value. Here is an example for that application:

```
[fold]
    input = I1
    output = O1
    foldby = 1V # one octave
    minimum = 1.2V
    maximum = 2.5V
```

If the input value at **I1** is going below 1.2 V, 1 V will be added over and over until the output voltage is at least 1.2 V. So the upper example will convert as follows:

- 0.7 V → 1.7 V
- 2.0 V → 2.0 V
- -4.3 V → 1.7 V
- 4.4 V → 2.4 V

If you apply that to a bass voice, you make sure that it never goes to low or too high, which is helpful if that voice is the result of a combination of sequences, random numbers, transpositions and other funny generative ideas.

Note: If you do not specify **minimum** or **maximum**, no folding will take place at that boundary. If you specify neither of them, this circuit is completely useless.

Anomalies

Two anomalies can happen if the parameters are a bit “crazy”. This first one happens, when the space between **minimum** and **maximum** is less than one **foldby**. Consider the following example:

```
[fold]
    input = I1
    output = O1
    foldby = 1V
    minimum = 1.1V
    maximum = 1.3V
```

Now if the input voltage is e.g. 1.0 V, it will be folded up to 2.0 V, which is then above the maximum range. But it will stay there, since there is no way to fold it into the range anyway.

The second anomaly is if **minimum** is greater than **maximum**. Look:

```
[fold]
    input = I1
    output = O1
    foldby = 1V
    minimum = 2.5V
    maximum = 1.5V
```

Here any voltage below 2.5 V will be folded up until it is above that value. so 2.4 V will be folded to 3.4 V. Well, you could also argue that because 2.4 V is also above the maximum value it should get folded down instead. While that is true, **fold** behaves asymmetrical here and gives folding up the precedence.

But why would you set such strange parameters? Well,

because they can be CVs of course. Try the following patch and send the output **O1** to the pitch input of a voice:

```
[p2b8]
[p2b8]

[lfo]
    hz = 2 * P1.1
    triangle = _CV

[lfo]
    hz = 2 * P1.2
    triangle = _MIN

[lfo]
    hz = 2 * P2.1
    triangle = _MAX

[lfo]
    hz = 2 * P2.2
    triangle = _FOLDBY
    level = 2V

[fold]
    input = _CV
    minimum = _MIN
    maximum = _MAX
    foldby = _FOLDBY
    output = O1

[lfo]
    rate = O1 * 0.2
    hz = 110
    output = O2
```

Here all four inputs are from slowly running LFOs and funny things happen. Play with the four pots and you will get all sorts of very interesting random patterns.

Input	Type	Default	Description
input		0.0	Input CV to be folded.
foldby		0.1	Amount to be added or subtracted from the input CV if it is not within the allowed range. This CV must be positive. If it is negative or zero, no folding will be done.
minimum			Lower bound of the allowed range. If unpatched, no lower bound will be applied.
maximum			Upper bound of the allowed range. If unpatched, no upper bound will be applied.

Output	Type	Description
output		Folded output voltage

13.26 fourstatebutton - Button switching through 4 states (OBSOLETE)

This circuit has been superseded by the new circuit **button** (see page 103). **button** can do all **fourstatebutton** can do and much more. So **fourstatebutton** will be removed soon.



This circuit converts one of the push buttons of your controllers into a button that switches through up to four different states. This is very similar to **togglebutton** but that supports just two states.

The LED will be off in state 1, 100% bright in state 4 and somewhere in between in the other two states.

The use case is to have a way to manually switch through three or four options. The following example implements an octave switch for a VCO. The button steps you through

the sequence $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 0$ octaves. The pitch is being read from **I1** and output again at **01** - possibly shifted by up to 3 octaves (3 V).

```
[fourstatebutton]
    button = B1.1
    led    = L1.1
    value1 = I1 + 0V
    value2 = I1 + 1V
    value3 = I1 + 2V
    value4 = I1 + 3V
    output = 01
```

Of course the values need not be fixed values. The next examples shows you a **DROID** patch where the button is used to cycle through four different wave forms of an LFO and send that to output **01**:

```
[lfo]
    hz      = 2
    square = _W1
    triangle = _W2
    sawtooth = _W3
    sine    = _W4
```

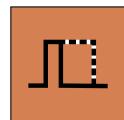
```
[fourstatebutton]
    button = B1.1
    led    = L1.1
    value1 = _W1
    value2 = _W2
    value3 = _W3
    value4 = _W4
    output = 01
```

Input	Type	Default	Description
button			The button.
reset			A positive trigger here will reset the button to the first state.
value1 ... value4			The values that output should get when the four various states are active.
startvalue		1 0 2 0 3	By setting this to 0, 1, 2 or 3 you set the initial state of the button when the DROID is powered up to state 1, 2, 3 or 4. It also disabled the automatic saving of the button's state in the DROID 's internal flash memory.

Output	Type	Description
output		Depending on the current state of the button here the value of input1 , input2 , input3 or input4 will be copied.
led		The LED in the button

13.27 gatetool - Operate on triggers and gates, modify gatelength

This utility works with triggers, gates and edge-triggers, can convert each type into each other type and can change the length of gates in flexible ways.



gatetool has three different types of inputs. Usually you would patch only one of these:

- **inputgate** expects a gate signal and honors the time span during which the gate is high. It takes into account the length of the input gate.
- **inputtrigger** expects triggers signals. Here the time span during which the input is high is not relevant. Just the start of the trigger counts. If you patch a “normal” gate signal here, the length of it is ignored (which could be just what you wanted).
- **inputedge** looks for transitions between low and high or high and low. Such transitions are called “edge”. Each time the input level swaps is considered as a trigger. So patching a normal gate signal here will count as a trigger when the gate goes high and another trigger when it goes low again.

For each input gate, trigger or edge, the **gatetool** outputs an output gate *and* an output trigger *and* an output edge:

- **outputgate** goes high on an input gate, trigger or edge. The length of the output gate can be modified in various ways (see below).
- **outputtrigger** outputs a short trigger of 10 ms on an input gate, trigger or edge.
- **outputedge** toggles between **0** and **1** on each input gate, trigger or edge.

Modifying the gate length

The length of the output gate on **outputgate** can be specified in various ways. First let's assume that you use the **inputtrigger** or the **inputedge** input. In this case there is no “input gate length”. The length of the output gate is set by **gatelength**, which is a number in seconds:

```
[gatetool]
    inputtrigger = I1
    outputgate = 01
    gatelength = 0.5 # 500 ms
```

As an option, you can set the gate length in relation to a reference clock (please see page 21 for details on using **taptempo** inputs). As soon as you patch **taptempo**, the **gatelength** parameter is in relation to one input clock tick (in DROID language 0.3 is just the same as 30%):

```
[gatetool]
    inputtrigger = I1
    taptempo = I2 # some steady clock
    gatelength = 0.3 # 30% of one clock tick
    outputgate = 01
```

Note: The **taptempo** input has the one and only purpose of setting a time reference to **gatelength**.

Now let's assume that you have an input gate signal, that has a specific length and you want to keep that or work on that. For that purpose use the **gateinput** and the **gateoutput**:

```
[gatetool]
    inputgate = I1
    outputgate = 01 # keep gate length
```

This example is not very useful, though, since it just copies the input gate to the output without changing the gate length. Use the **gatelength** parameter to switch the behaviour to that of **triggerinput**: the input gate length is ignored and overruled by that parameter:

```
[gatetool]
    inputgate = I1
    outputgate = 01
    gatelength = 0.5 # 500 ms
```

More interesting is **gatestretch**. This is the first time the length of the input gate is honored and has any relevance: Here you specify a percentage by that the gate should be made *longer*:

```
[gatetool]
    inputgate = I1
    outputgate = 01
    gatestretch = 0.3 # make gate 30% longer
```

For obvious reasons you cannot make a gate shorter by a percentage since nobody - not even Droid - can look into the future...

Note: **gatestretch** obviously only makes sense if you don't use **gatelength**!

If you want to keep the gate length within certain bounds, you can make use of **mingatelength** and **maxgatelength**. They set a lower or upper limit on the length of the output gate. They only are effective when **gatelength** is not used. Both parameters are in seconds or - if **taptempo** is used - in fractions of one clock tick.

The following example forwards the input gates unchanged to the output, but makes sure that the length

is never shorter than 10% and never longer than 90% of a clock tick:

```
[gatetool]
  inputgate = I1
  taptempo = I2 # steady clock
  outputgate = O1
  mingatelength = 0.1
  maxgatelength = 0.9
```

Building a clock divider

The edge triggers can help you building a clock divider that divides by two. Of course you could do that with **clocktool** (see page [121](#)), as well. But this example illustrates a bit, how the edge triggers work. Consider the following example:

```
[gatetool]
  inputtrigger = I1
  outputedge = O1
```

Now for every trigger in **I1**, the edge output *changes* its

level. So in order to go from low to high and low again, you need two input triggers. The output signal at **O1** then just outputs one gate signal in that time. So two triggers are converted into one.

Use edges for pinging filters

Another application of edges is pinging filters with a *zero length* impulse. Use the same patch snippet as above and patch **O1** to the input of a resonant filter. By just using the edge, you really get exactly *one* ping. A trigger – regardless how short – always has two edges and thus pings the filter twice, which can sound unclean.

Input	Type	Default	Description
inputgate			Input gate. Use this if the length of the input gate is relevant.
inputtrigger			Input trigger. Use this if the length of the input gate should be ignored.
inputedge			Input edge: Use this if every low/high or high/low transition should count as a trigger.
gatelength			Sets the length of the gate of outputgate in seconds. If you use taptempo the length is in fractions of a clock tick, instead.
gatestretch		0.0	Makes the output gate longer than the input gate by the given percentage. This parameter is ignored if gatelength is used.
mingatelength		0.01	Defines a minimum length of the output gate in seconds or clock ticks.
maxgatelength			Defines a maximum length of the output gate in seconds or clock ticks.
taptempo			If you patch a reference clock here, gatelength , mingatelength and maxgatelength are fractions of one clock tick, not seconds anymore. Please see page 21 for details on using taptempo inputs.

Output	Type	Description
outputgate		Outputs a gate with controllable length for every gate, trigger or edge event.
outputtrigger		Outputs a 10 ms trigger for every gate, trigger or edge event.

Output	Type	Description
outputedge		Toggle between 0 and 1 at every gate, trigger or edge event.

13.28 lfo - Low frequency oscillator (LFO)

A flexible low frequency oscillator with seven different waveforms, phase modulation, flexible sync mechanisms, randomization, wave form morphing and other interesting features.



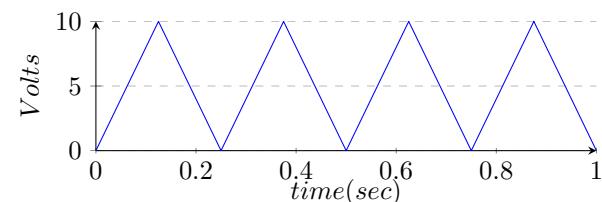
Please note also that this LFO is not intended to be used at audio rate. It can probably operate until roughly 1000-1500 Hz, but will sound ugly, distorted and with many digital artefacts - especially the waveforms with steep edges like square, ramp and sawtooth. If that's exactly what you intend, then maybe you will have fun anyway.

Waveforms

Here is the simplest possible patch. In this example the frequency is specified in Hertz (cycles per seconds) and the **triangle** output is routed directly to **01**:

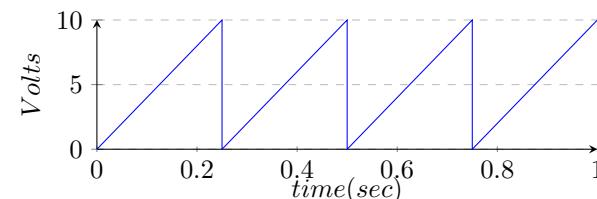
```
[lfo]
    hz      = 4
    triangle = 01
```

The resulting output looks like this:

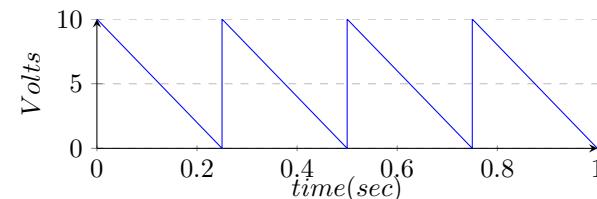


This is how the **sawtooth** output looks like:

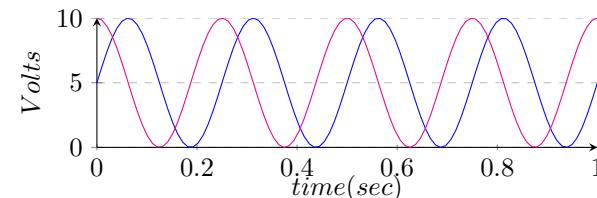
```
[lfo]
    hz      = 4
    sawtooth = 01
```



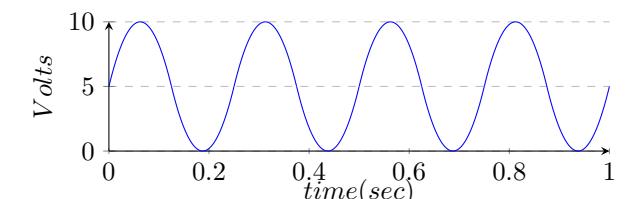
The **ramp** is similar but falling instead of rising:



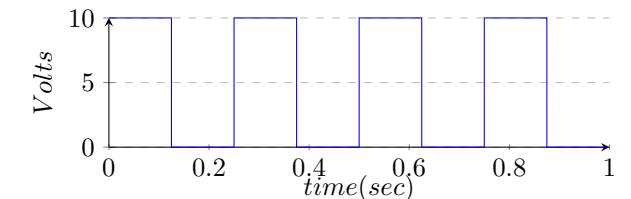
The waveforms **sine** and **cosine** are similar but are one quarter cycle (90°) apart:



paraboloid is very similar to sine, but is constructed based on quadratic equations (which is faster):



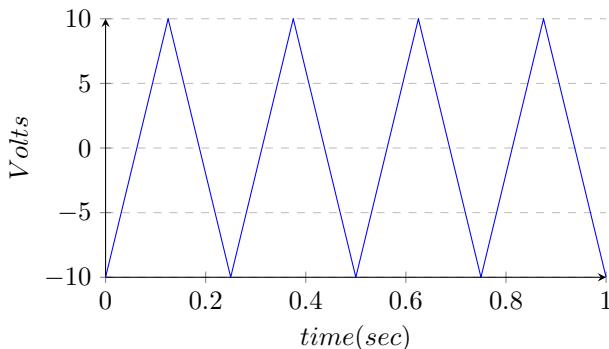
Maybe the simplest waveform is **square**:



Bipolar output, Level and Offset

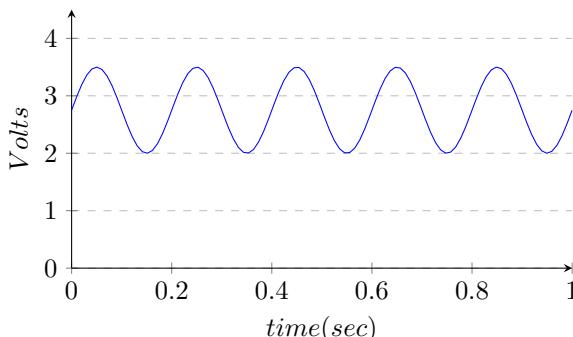
Please note that the LFO outputs just positive voltage ranges until you set **bipolar = on**. That extends the waveform to negative voltages (while doubling the peak-to-peak voltage):

```
[lfo]
    hz      = 4
    bipolar = on
    triangle = 01
```



The inputs **level** and **offset** can be used to control the voltage range of the outputs - which is here for your convenience and avoids the need for additional circuits for doing this. The following example outputs a sine wave at 5 Hz to 04 that is gently oscillating between 2 V and 3.5 V:

```
[lfo]
hz      = 5
level   = 1.5V
offset   = 2V
sine    = 04
```



Frequency control

The frequency of the LFO can be controlled in various ways. In the upper examples we used the input **hz**. Here you specify the frequency of the LFO directly in Hz. This is ideal when you want to set a fixed frequency with a discrete number, rather than a control voltage. Here is a rectangle LFO running at 1.5 cycles per second:

```
[lfo]
hz      = 1.5
rectangle = 03
```

A more eurorack-like way is using the **rate** input, which works on a 1V/octave scheme. One “octave” here means that the frequency doubles. Here is an example for creating a triangle LFO running at 4 Hz, since 2 V doubles the base frequency of 1 Hz two times (instead of **2V** you could also write **0.2**):

```
[lfo]
rate    = 2V
bipolar = on
triangle = 01
```

The third way is to use *tap tempo* by sending a steady clock into **taptempo**. The LFO then mimics the speed of that input clock. This can even be combined with **rate**: If you use both, then first **taptempo** is being used to set the speed and then **rate** is used for altering that speed. So sending -1 V into **rate** will create an LFO running at half clock speed (since -1 V pitches down the LFO by one octave).

```
[lfo]
taptempo = G1 # steady clock here
rate     = -1V # run at half clock speed
sawtooth = 02
```

And even **hz** can be used in combination. Now the speed of the taptempo is multiplied with the value of **hz**. Otherwise stated: 1 Hz is the reference. The following sets the LFO's frequency to three times the tap tempo:

```
[lfo]
taptempo = G1
hz = 3
sawtooth = 02
```

Hz vs BPM

Sometimes people ask for help converting BPM into Hz or vice versa. And some even express their unhappiness about the fact that the Droid uses Hz rather than BPM. Well - that decision was made because in general I see the LFO rather as an oscillator than as a clock. And for oscillators Hz is the usual way to measure the speed or frequency.

So when you use an LFO as your master clock, how can you convert specify BPM as Hz? **Simply divide your BPM by 15** to get the correct value for **hz**. So 120 BPM would be **hz = 8**.

That sounds surprising, since Hz means *oscillations per second* and BPM *beats per minute*. The point is: BPM means *beats per minute*, not clock ticks per minute. In a modular environment it is most common to run your clock at 16th notes. And the “beat” in BPM refers to *quarter* notes. For playing one quarter note we need to play four 16th notes, so after dividing by 60 to convert minutes into seconds, we need to multiply by 4, to convert quarters into 16ths.

That - of course - assumes that your master clock is running at 16th notes, sometimes written as 4 PPQN (4 pulses per quarter note).

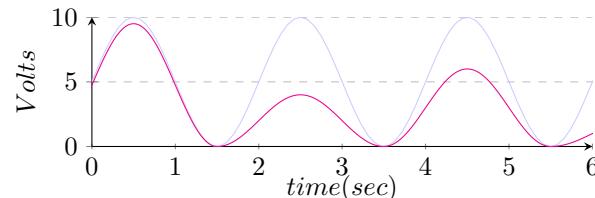
Randomization

Randomization is an experimental new feature that combines random voltages with an LFO. If you turn this parameter up, then for each “hill” of the output waveform has a different height. The parameter **randomize** controls how strong that effect is. With **0.0** randomization is turned off. At **1.0** it is at its strongest and the random level of each hill is in the range **0.0 ... 1.0**.

Here is an example of a randomized sine wave:

```
[lfo]
    hz      = 0.5
    randomize = 0.8
    sine     = 01
```

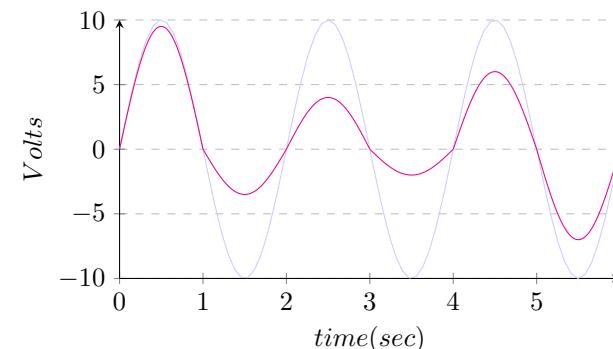
The original wave if printed **light** and the and the randomized wave at output **01** is **magenta**:



Please note: If you turn **bipolar** on, then a “hill” is considered to be something *above* or *below* the zero line. That means that now the sine wave has twice as much hills and the randomization works different. Here is an example patch:

```
[lfo]
    hz      = 0.5
    randomize = 0.8
    sine     = 01
    bipolar   = 1
```

And this is how the output looks like:



Note: Since not all waveform have there “hills” at the same place and the start and end of a hill might even be affected by **skew** or **pulsewidth**, each waveform output has its own independent randomization. Therefore **cosine** is *not* the phase shifted output of **sine** anymore, if you use randomization.

Wave form selection and morphing

As an alternative to the seven individual waveform outputs there is a common output simply called **output**. The waveform can be selected with the input **waveform** and defaults to **0**, which means *square wave*. So for a simple clock you can write:

```
[lfo]
    hz = 2
    output = G1
```

A triangle wave is selected with the code **2**:

```
[lfo]
    hz = 2
```

```
output = G1
waveform = 2
```

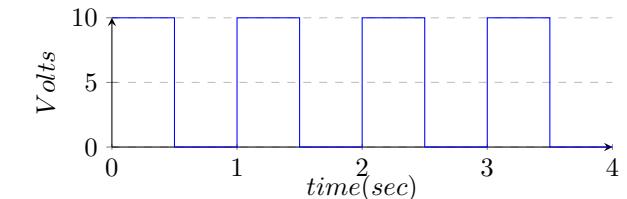
Here is the complete list of available waveforms:

0	square
1	sawtooth
2	triangle
3	ramp
4	paraboloid
5	sine
6	cosine

It is allowed to use non-integer values, like **0.5**. This will create a mixture between two adjacent waveforms - while respecting the ratio. For example **2.1** will select 90% triangle and 10% ramp. That way you can smoothly morph through the available waveforms. Here is an example. Let's start with **waveform = 0.0**, which gives a plain square wave:

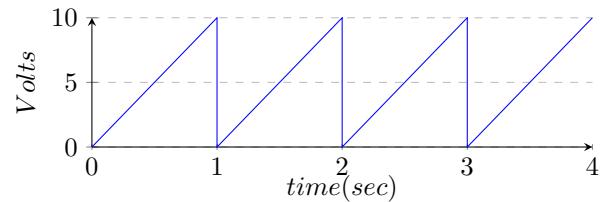
```
[lfo]
    hz = 4
    output = 01
    waveform = 0.0
```

And this is what it looks like:



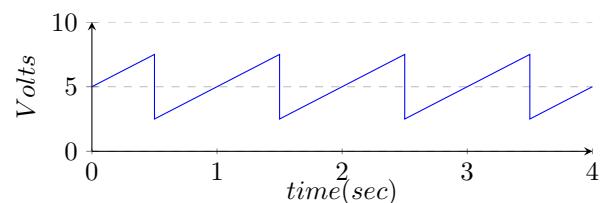
At **1.0** we get a saw tooth:

```
[lfo]
  hz = 4
  output = 01
  waveform = 1.0
```



And in between - at **0.5** - we get some mixture:

```
[lfo]
  hz = 4
  output = 01
  waveform = 0.5
```



Input	Type	Default	Description
rate		0.0	Frequency control: The default frequency of the LFO is 1 Hz (one cycle per second). Each volt doubles the frequency. So an input of 1 V (a number of 0.1) speeds up the LFO to 2 Hz, 2 V (0.2) create 4 Hz and so on. On the other hand negative voltages reduce the speed, so -1 V (-0.1) will give 0.5 Hz and so on.
taptempo			Feed a reference clock here and the LFO will run at the speed of that clock - albeit optionally modified by rate and hz . Please see page 21 for details on using taptempo inputs.
hz		1.0	Set the frequency in Hz directly by setting a number here. Note: you cannot use hz at that same time as taptempo . But both can be combined with rate .
level		1.0	The maximum positive output level of the LFO. The default of 1.0 means a swing between 0 V and 10 V - unless you enable bipolar , in which case it moves from -10 V to 10 V.
randomize		0.0	Randomization is an experimental new feature that combines random voltages with an LFO. If you turn this parameter up, then for each <i>hill</i> of the LFO's waveform output a new random attenuation is being chosen and multiplied with the current level. The result is an output, where each cycle of the waveform has a different level.
offset		0.0	The output of the LFO is shifted by that voltage right before the output. This is the same as adding or mixing a fixed voltage to the output. Not very fancy, but practical if you want to output a modulation voltage within a certain range.
bipolar		0	If this switch is set to on , then the LFO will output a full swing from -level to +level . When set to off it will swing between 0V and +level .
phase		0.0	Shift the LFOs phase by this value. A value of 0.0 leaves the LFO run in its normal phase. 0.5 will shift bei 180°. And 1.0 will shift by a complete phase of 360°, which is the same as 0.0 .
pulsewidth		0.5	This sets the pulse width of the square LFO and only affects the output square . It ranges from 0.0 to 1.0 . Please note that a pulse width of exactly 0.0 or 1.0 will make the output stick to the respective lower or upper level.
skew		0.5	Modifies the symmetry of the triangle output by shifting the “peak” of the triangle left and right. The default of 0.5 creates a symmetric waveform. Smaller values speed up the rising part of the triangle and create more and more a ramp like waveform until a skew of 0.0 creates an exact ramp - just the same as the ramp output. A skew of 1.0 create a sawtooth waveform.
sync			A positive trigger edge at this input will reset the LFO. It will force to restart the waveform at its “beginning”. By using the input syncphase you can change that behaviour.
syncphase		0.0	This input changes the behaviour of the sync input. It changes the phase the waveform restarts at when it receives a sync trigger. E.g. by setting this to 0.5 a sync trigger will restart the waveform right at its middle. This is an interesting feature that cannot be found in analog LFOs since it would be very hard to build in actual circuits.

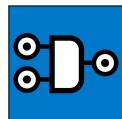
Input	Type	Default	Description
<code>waveform</code>		0.0	If you use <code>output</code> - rather than the individual waveform outputs like <code>square</code> , <code>saw</code> and so on - this input selects the Waveform. An integer number from 0 to 6 selects one of the seven available waveforms. Any number in between selects a mixture of the two neighboring waveforms. That way you can smoothly morph through all the available waveforms. The codes for the waveforms are:

0	square
1	sawtooth
2	triangle
3	ramp
4	paraboloid
5	sine
6	cosine

Output	Type	Description
<code>output</code>		Main output of the LFO.
<code>square</code>		A square waveform - modified by <code>pulsewidth</code> .
<code>sawtooth</code>		Outputs a sawtooth waveform - i.e. a rising ramp
<code>triangle</code>		Outputs a triangle waveform - modified by <code>skew</code> .
<code>ramp</code>		Outputs a falling ramp - like a sawtooth that is mirrored. Note: if the LFO is set to bipolar then this is the negation of <code>sawtooth</code> . If it is set to unipolar then this is not the case. The waveform will be positive then!
<code>paraboloid</code>		An experimental waveform that looks very similar to a sine wave but is derived from a triangle by computing the square of each waypoint's distance to <code>level</code> .
<code>sine</code>		A sine waveform.
<code>cosine</code>		A sine waveform shifted by 90°. This output is for your convenience and avoids needing two LFO circuits in cases where you want to make quadrature applications. Please note that 180° and 270° can easily be achieved by negating the outputs <code>sine</code> and <code>cosine</code> at a later stage.

13.29 logic - Logic operations utility

Utility circuit for logic operations on gate signals. It can do operations like AND, OR, NAND, NOR, etc.



Basic operation

In this example we do an **and** operation. **O1** will output 1 (**on**) if all of **I1**, **I2** and **I3** see **on** (voltage above 1 V):

```
[logic]
  input1 = I1
  input2 = I2
  input3 = I3
  and    = O1
```

Here is how to do a logic negate of a signal:

```
[logic]
  input   = I1
  negated = O1
```

If you do not like the 1 V threshold, you can change it:

```
[logic]
  input     = I1
  negated  = O1
  threshold = 5V
```

Doing logic without this circuit

Please note, that many times when you think you need the logic circuit you can do the same much simpler. Here

is an example, where you use a toggle button to switch on a clock, which is sent to output **O1**. The idea is to make an AND combination of the clock signal and the button state:

```
[button]
  button = B1.1
  led    = L1.1

[lfo]
  hz      = 2
  square = _LFO
```

```
[logic]
  input1 = L1.1
  input2 = _LFO
  and    = O1
```

While this works pretty well, here is a solution that makes use of the fact, that the *multiplication* of two gate signals is in fact a kind of AND combination, since $A \times B$ is just 1, if A and B are 1 and 0 otherwise:

```
[button]
  button = B1.1
  led    = L1.1

[lfo]
  hz      = 2
  square = _LFO

[copy]
  input  = _LFO * L1.1
  output = O1
```

You even can avoid the Copy-circuit if you make use of the **level** input of the LFO, since setting the level to 0 disables it:

```
[button]
  button = B1.1
  led    = L1.1
```

```
[lfo]
  hz      = 2
  square = _LFO
  level  = L1.1
```

Another nice solution is to make use of **offvalue** and **onvalue** of the **button** circuit. **offvalue** is 0 per default, so we just need to define **onvalue**:

```
[lfo]
    hz      = 2
    square = _LFO

[button]
    button = B1.1
    led    = L1.1
    onvalue = _LFO
```

If you need to combine two gates in order to create a common gate pattern, you can use *addition* - which is very similar to a logic OR combination. The following example creates two overlayed euclidean rhythms:

```
[euklid]
    length = 16
    beats   = 3
    output  = _E1

[euklid]
    length = 13
    beats   = 2
    output  = _E2

[copy]
    input   = _E1 + _E2
    output  = 01
```

Note: When both **_E1** and **_E2** are 1 at the same time, the sum is 2, of course. This does not matter, since the output voltage is capped at 10 V (**1.0**) anyway.

Input	Type	Default	Description
<code>input1 ... input8</code>			1st ... 8th input. Note: this input is declared as a CV gate input, but in fact you can use it as a CV input in combination with various or random values set for the threshold .
<code>threshold</code>		0.1	Input values at, or above this threshold value, are considered high or on . The default is 0.1 which corresponds to an input voltage of 1 V. You can get interesting results when both the inputs are variable CVs (like from LFOs) and this threshold is being modulated as well.
<code>lowvalue</code>		0.0	Output value that is output for logic low, false or off .
<code>highvalue</code>		1.0	Output value that is output for a logic high, true or on .
<code>countvalue</code>		0.1	Value added to the <code>count</code> output for each input with a high level

Output	Type	Description
<code>and</code>		A logic AND operation on all patched inputs: This output is set to highvalue if all inputs are high (i.e. at least threshold), else lowvalue
<code>or</code>		A logic OR operation on all patched inputs: This output is set to highvalue if at least one of the inputs is high
<code>xor</code>		Exclusive OR: This is high, if the number of high inputs is odd! This means that any change in one of the inputs will also change the output.
<code>nand</code>		Like AND but the outcome is negated.
<code>nor</code>		Like OR but the outcome is negated.
<code>negated</code>		Logical negate of <code>input1</code> (which can abbreviated as <code>input</code>). Note: The inputs <code>input2 ... input7</code> are ignored here. Another note: If you use <code>input1</code> anyway, <code>negated</code> always outputs exactly the same as <code>nand</code> and <code>nor</code> . It's just more convenient to write and easier to understand. Hence a dedicated output for a logic negate.
<code>count</code>	<code>1°2°3</code>	Adds <code>countvalue</code> to this output for each input that is high.
<code>countlow</code>		Adds <code>countvalue</code> to this output for each input that is low.

13.30 math - Math utility circuit

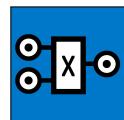
This circuit provides mathematic operations. Some of these use `input1` and `input2` - such as `sum` or `product`. Other ones just use `input1` (which can be abbreviated as `input`) - such as `negation` or `reciprocal`.

Example for computing the quotient $\frac{I1}{I2}$:

```
[math]
input1 = I1
input2 = I2
quotient = 01
```

Example for computing the square root of `I1`:

```
[math]
input = I1
```



```
root = 01
```

Note: As long as you do not send a value directly to an output like `01`, the range of the value is not limited by this circuit. You can generate almost arbitrary small or large positive and negative numbers. When you send a value to an output, it will be truncated into the range $-1 \dots +1$ (which corresponds to $-10 \text{ V} \dots +10 \text{ V}$).

Unused inputs

When you don't use both inputs for an operation that usually needs two values, the omitted input will make the operation "neutral". For example in the multiplication an omitted input will be treated as `1.0` where as in the sum it defaults to `0.0`. This is useful when you want to temporarily disable a line in your patch. Consider the follow-

ing patch, which multiplies the incoming CV from `I1` with the pot value of `P1.2` and outputs it to `01`.

```
[math]
input1 = I1
input2 = P1.2
product = 01
```

If you now remove the line with `input2`, the output will simply copy the input, not set it to 0:

```
[math]
input1 = I1
# input2 = P1.2
product = 01 # will be set to I1, not 0
```

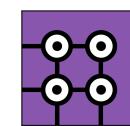
Input	Type	Default	Description
<code>input1, input2</code>			The two inputs

Output	Type	Description
<code>sum</code>		<code>input1 + input2</code>
<code>difference</code>		<code>input1 - input2</code>
<code>product</code>		<code>input1 × input2</code>
<code>quotient</code>		<code>input1 / input2</code> . If <code>input2</code> is zero, a very large number will be returned, while the correct sign is being kept. This is mathematically not correct but more useful than any other possible result.
<code>modulo</code>		<code>input1 modulo input2</code> . This needs some explanation: With this operation you can "fold" the value from <code>input1</code> into the range $0 \dots \text{input2}$. For example if <code>input2</code> is 1 V, the output will convert 1.234 V to 0.234 V, -2.1 V to 0.9 V and 0.5 V to 0.5 V. If <code>input2</code> is zero or negative, the output will be zero.

Output	Type	Description
power		<code>input1</code> to the power of <code>input2</code> . Please note that the power has several cases where it is not defined when either the base or the exponent is zero or less than zero. In order to be as useful for your music making as possible the <code>math</code> circuit behaves in the following way: <ul style="list-style-type: none"> If <code>input1 < 0</code>, <code>input2</code> is rounded to the nearest integer. If <code>input1 = 0</code> and <code>input2 < 0</code>, a very large number is output.
average		The average of <code>input1</code> and <code>input2</code>
maximum		The maximum of <code>input1</code> and <code>input2</code>
minimum		The minimum of <code>input1</code> and <code>input2</code>
negation		<code>-input1</code>
reciprocal		<code>1 / input1</code> . If <code>input1</code> is zero, a very large number is being output, while the sign is being kept.
amount		The absolute value of <code>input1</code> (i.e. <code>-input1</code> if <code>input1 < 0</code> , else <code>input1</code>)
sine		The sine of <code>input1</code> in a way, the input range of 0.0 ... 1.0 goes exactly through one wave cycle. Or more mathematically expressed: $\sin(2\pi \times \text{input1})$.
cosine		The cosine of <code>input1</code> in a way, the input range of 0.0 ... 1.0 goes exactly through one wave cycle. Or more mathematically expressed: $\cos(2\pi \times \text{input1})$.
square		<code>input1^2</code>
root		$\sqrt{\text{input1}}$. Please note that you cannot compute the square root of a negative number. In order to output something useful anyway, the result will be $-\sqrt{-\text{input1}}$, if <code>input1 < 0</code> .
logarithm		The natural logarithm of <code>input1</code> : $\ln \text{input1}$. The logarithm is only defined for positive numbers. <code>mathcircuit</code> behaves like this: <ul style="list-style-type: none"> If <code>input1 = 0</code>, a negative very large number is output. If <code>input2 < 0</code>, $-\ln -\text{input1}$ is output.
round		The integer number nearest to <code>input1</code>
floor		The largest integer number that is not greater than <code>input1</code>
ceil		The smallest integer number that is not less than <code>input1</code>

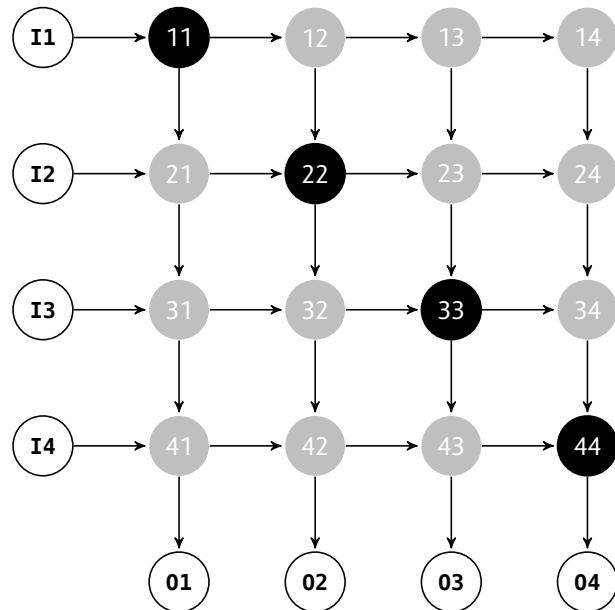
13.31 matrixmixer - Matrix mixer for CVs

This circuit is a 4×4 matrix mixer with four inputs and four outputs that is operated by push buttons. Each of the 16 matrix nodes has a toggle button for adding or removing one specific input to or from one specific output. The mixing is always done with unity gain. This means that each output is the sum of all inputs that are enabled on its path.



The following picture shows a matrix with the four inputs **I1** ... **I4** and the four outputs **01** ... **04**. As you can see the button 23 mixes input 2 to output 3.

If you have not pushed any buttons yet, the mixer enables four buttons in a diagonal so that inputs **I1** is connected to output **01** and so on:



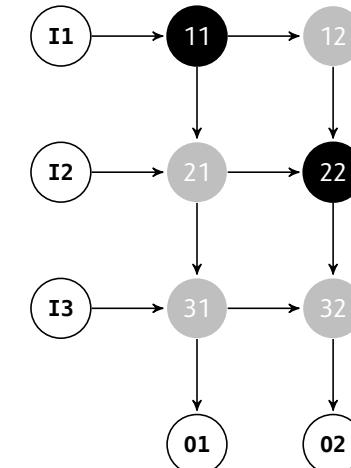
As an alternative operation, instead of summing the enabled signals you can compute the *maximum* signal. This is useful when combining envelope signals - e.g. from different rhythmic patterns. Adding envelope signals would either make them "too loud" or even distort them.

The current state of the sixteen buttons is saved in the DROID's internal flash memory.

Of course it is possible to use a smaller part of the matrix, e.g. just 3×2 , simply by not patching the according inputs, outputs and buttons. Here is an example of a 3×2 mixer:

```
[matrixmixer]
  input1 = I1
  input2 = I2
  input3 = I3
  output1 = 01
  output2 = 02
  button11 = B1.1
  button12 = B1.2
  button21 = B2.1
  button22 = B2.2
  button31 = B3.1
  button32 = B3.2
  led11 = L1.1
  led12 = L1.2
  led21 = L2.1
  led22 = L2.2
  led31 = L3.1
  led32 = L3.2
```

This matrix looks like this:



Mixers with more inputs / outputs

The four auxiliary inputs **auxin1** ... **auxin4** can be used to create matrix mixers with more than four inputs. You can

create a mixer with 8 inputs and 4 outputs by sending the four outputs of one matrix mixer into the four auxiliary inputs of a second one.

If you want to create a mixer with more than 4 *outputs* then simply use several mixers and feed the same inputs to all of them.

Input	Type	Default	Description						
input1 ... input4		0.0	The up to four CV inputs that you want to mix						
auxin1 ... auxin4			These auxiliary inputs will be mixed directly into the four outputs output1 ... output4 and are used for cascading several matrix mixers into one with more than four inputs.						
mixmax		0.0	If this is 0.0 , normal mixing is done (the enabled inputs CVs will be added). At a value of 1.0 instead each output is the maximum of the enabled inputs. Any number in between will create a weighted average between these two values.						
startvalue		1	This input selects in which state the matrix begins life. Also a trigger to clear will create that starting state. The following three configurations can be selected with startvalue :						
			<table border="1"> <tr> <td>0</td><td>All buttons are cleared.</td></tr> <tr> <td>1</td><td>The buttons on the diagonal are active.</td></tr> <tr> <td>2</td><td>All buttons are set.</td></tr> </table>	0	All buttons are cleared.	1	The buttons on the diagonal are active.	2	All buttons are set.
0	All buttons are cleared.								
1	The buttons on the diagonal are active.								
2	All buttons are set.								
			When set to 1, input1 is sent to output1 , input2 to output2 and so on.						
button11 ... button14			These four buttons decide, to which of the four outputs input1 is being mixed.						
button21 ... button24			These four buttons decide, to which of the four outputs input2 is being mixed.						
button31 ... button34			These four buttons decide, to which of the four outputs input3 is being mixed.						
button41 ... button44			These four buttons decide, to which of the four outputs input4 is being mixed.						
select			The select input allows you to overlay buttons and LEDs with multiple functions. If you use this input, the circuit will process the buttons and LEDs just as if select has a positive gate signal (usually you will select this to 1). Otherwise it won't touch them so they can be used by another circuit. Note: even if the circuit is currently not selected, it will nevertheless work and process all its other inputs and its outputs (those that do not deal with buttons or LEDs) in a normal way.						
selectat			This input makes the select input more flexible. Here you specify at which value select should select this circuit. E.g. if selectat is 0 , the circuit will be active if select is <i>exactly</i> 0 instead of a positive gate signal. In some cases this is more convenient.						

Input	Type	Default	Description
preset	1..2..3		This is the preset number to save or to load. Note: the first preset has the number 0 , not 1 ! For the whole story on presets please refer to page 19 . This circuit has 16 presets, so this number ranges from 0 to 15 .
loadpreset			A trigger here loads a preset. As a speciality you can use the trigger for selecting a preset at the same time.
savepreset			A trigger here saves a preset.
clear			A trigger here loads the default start state into the circuit. The presets are not affected, unless you use direct preset switching with the preset input and without triggers. And that case the current preset is also cleared.
clearall			A trigger here loads the default start state into the circuit and into all of its presets.
dontsave		0	If you set this to 1 , the state of the circuit will not saved to the SD card and not loaded from the SD card when the Droid starts.

Output	Type	Description
output1 ... output4		The four outputs
led11 ... led14		The LEDs in the buttons button11 ...button14
led21 ... led24		The LEDs in the buttons button21 ...button24
led31 ... led34		The LEDs in the buttons button31 ...button34
led41 ... led44		The LEDs in the buttons button41 ...button44

13.32 midifileplayer - MIDI file player

This circuit can read MIDI files from your Micro SD card and “play” them by creating respective CVs for gate, pitch, velocity, pitch bend and other outputs, which you can then route to synth voices in your modular - or do other crazy stuff with that information.



MIDI files are organized in tracks. Each circuit of this type can play just *one track* at a time. If you want to play more tracks, use more **midifileplayer** circuits in parallel.

Just as MIDI streams, MIDI files contain *channel* information for each note and each controller event. These channels are currently completely ignored. If you think you can convince me that this is bad and that you have a useful interpretation of the channels within the scope of the MIDI file player, please let me know.

Some limitations of the current implementation are:

- Just one track can be played at a time.
- The maximum length of a track is 6000 bytes. Longer tracks cannot be loaded. Sorry. But this is quite long and is enough for approximately 1500 note events. Note: The size of the total file can be as large as you like.
- The channel information is ignored.
- Some meta events such as program change, all notes off, etc. are not yet recognized. Many of them just make sense in MIDI streams, not in files, anyway.

Features of the current implementation:

- Up to eight voices in parallel with flexible voice allocation algorithms
- Support for velocity, pitch bend, mod wheel, and global volume

- You can output the original MIDI clock from the file.
- You can adjust the tempo continuously.
- You can use external clocking (ignoring the tempo of the file).

Getting started

Here is the simplest possible example: Copy your MIDI file to the SD card and name it **midi1.mid**. And here is the patch that plays the first track with a single voice:

```
[midifileplayer]  
pitch = 01  
gate = 02
```

Now patch **01** to the 1V/Oct of a synth voice and **02** to its gate. This voice should then play the notes from the first track of the file.

The playback starts immediately when the DROID starts. Per default the track is looped. You can restart the playback with the **reset** input. And the other way round: you get a trigger at **endoftrack** when the playback of the track has finished.

Selecting file and track

You can have more than one MIDI file on your SD card. The MIDI files on the card must be named **midi1.mid**, **midi2.mid**, and so on. Gaps are allowed. You can have up to 9999 MIDI files that way. The last one would have the name **midi9999.mid**. Don't use leading zeroes! The file **midi0001.mid** cannot be played!

You can then select one of these files with the **file** parameter, so e.g. **file = 17** would play **midi17.mid**. If you omit that, **midi1.mid** will be played. If no such file is present on the card, nothing will be played.

A MIDI file can contain several tracks. The **track** parameter specifies the number of the track in the file you want to play. Hereby only the non-empty tracks will be counted. This is important since many MIDI files have tracks that just contain meta information and no note events.

If you omit the track number, the first non-empty track will be played. If your track number is out of range, the last track in the file will be selected.

The parameters **file** and **track** are - of course - CV controllable. So you can switch between files and tracks by means of buttons, switches, external CV, you name it. Whenever the file or track changes, **DROID** loads the selected track from the SD card into its memory. This is also the case when the **DROID** starts. Also a track change restarts playback.

Note: loading a track from the SD card might take a couple of milliseconds. During that time **DROID** won't run as usual. All inputs will be ignored and all outputs freeze. So switching at a high rate might lead to unexpected results. If you need to have a playback started in perfect timing, use the **reset** input as an exact trigger. If you do not want to use a trigger but rather a play/stop gate, you can use the **speed** input for that. Setting the speed to **0** stops playback and **1** starts it immediately.

Polyphonic tracks

MIDI streams and files consist of *note on* and *note off* events. So there is no length parameter in a note. It just contains the note number (in semitones) and a velocity. If the track contains situations where a new note starts while another one is still on, the track is polyphonic, as you need more than one synth voice to play correctly.

The MIDI file player allows you to define up to *eight* voices for playing notes. Each voice consists of a **pitchX** and a **gateX** output (and an optional **velocityX** output). By patching these outputs the player knows how many voices are available.

If the number of simultaneous notes exceeds the number of attached voices, some notes have to be cut off or completely omitted. You can flexibly change the behaviour in such a situation. See the description of the parameter **dropnotes** for details.

Here is an example for playing with up to three voices:

```
[midifileplayer]
  file = 2
  track = 1
  pitch1 = 01
  pitch2 = 02
  pitch3 = 03
  gate1 = G1
  gate2 = G2
  gate3 = G3
```

Speed and Clocking

A MIDI file contains absolute timing information of when to exactly play which note. For that purpose every note event in the file has a relative *time stamp*, measured in

ticks. The player honors this information and plays the tracks exactly in their original speed... unless... you change it of course.

To do so you have two options. The first one is the **speed** parameter. At **1.0** you get the original playing speed. **0.5** will play at half the speed and **2.0** at the double speed. This can be mapped to a pot, of course (here I chose a range from 0 to 2):

```
[midifileplayer]
  pitch = 01
  gate = 02
  speed = P1.1 * 2
```

Turning the pot totally CCW will completely freeze the playback.

If you need the internal clock of the MIDI player in order to synchronize with the rest of your patch, you can get two clocks running at different resolutions at the two outputs **clockout** and **midiclock**. See their descriptions below for details.

The second option is clocking the player externally. In that case the tempo information from the MIDI file is ignored. External clocking allows you to synchronize the MIDI playback with the rest of your patch, which may contain additional sequencers and stuff. Patch your external clock into the **clock** input. Each clock will then play a 16th note's time equivalent of content:

```
[midifileplayer]
  pitch = 01
  gate = 02
  clock = G1
```

Note: this does *not* mean that the notes are quantized to 16th notes. You still have the complete resolution.

Other controls and parameters

MIDI files may contain information about pitch bend, a global volume (CC 7), the mod wheel (CC 1) and velocity (per note). These are all available as CV outputs. See the table of outputs for details. Most other CCs are currently not available since they are very rarely used in MIDI files. Future versions of the MIDI file player might give access to these.

Error handling

When working with files, errors can happen. The MIDI file might be missing, corrupted, whatever. In order to make life easier for you, the MIDI file player can show you an error status at the output **error**. Write the error to an **R** register that is free, that will make one of the LEDs lit up and show an error color.

The following patch shows the errors at the LED of input 1:

```
[midifileplayer]
  pitch = 01
  gate = 02
  error = R1
```

Please see the table of outputs below for the various errors and their color codes.

Input	Type	Default	Description
file	1◦2◦3	1	Number of the MIDI file to play. 7 will select midi7.mid .
track	1◦2◦3	1	Number of the track in the file to play, starting at 1. Empty tracks do not count. Any number smaller than 1 will be interpreted as one. If the number is too big, the last track in the file is played.
clock			Patch an external clock here and the MIDI file will be played according to that clock. In order to be modular-friendly, this is <i>not</i> a MIDI clock but one counting the sixteenth, which is typically the step resolution of analog sequencers. This clock is then internally multiplied in order to create the necessary resolution. Note: The input speed has no effect when using an external clock.
reset			A trigger here sets the play back position to the start.
loop		1	When loop mode is active (set to 1), the track will start over again immediately when it has reached its end. This is the default. Otherwise playback stops at the end of the track.
end	1◦2◦3		If you set this value, it defines the playing end of the track. This is set in quarters as counted from the start. Setting the end beyond the end of the track will insert some pause.
speed		1.0	Change the relative speed of the playback with this setting. At 1 the speed is unchanged. 1.5 makes the speed 50% faster, 0.5 plays at half speed. At 0 the playing is completely frozen. Note: speed is being ignored when using the input clock .
channel	1◦2◦3		Only execute / play commands from a certain MIDI channel. There are 16 MIDI channels. It ranges from 1 to 16 .
tuningmode		off	If set to 1 , all pitch outputs will go to the CV selected for tuningpitch (which defaults to 2V), and all gate outputs will play gates at 120 BPM. This helps getting all attached voices tuned when working with many voices.
tuningpitch	$\frac{1V}{Oct}$	2V	This pitch CV will be output while the tuning mode is active.
transpose	$\frac{1V}{Oct}$	0V	Transposes all output pitches by this value by adding the value. So in order to transpose one octave down, set this input to -1V or -0.1 . Changes in the transposition are immediately reflected, even for currently already active notes.
holdvelocity		0	If this is set to 1 , the velocity output for a voice will not be affected by note off events. It's just altered at the beginning of new notes. The velocity is kept after the note ends. This way during the release phase of an envelope triggered by the gate, the original velocity still lasts on. In most cases the note off velocity is set to 0, which would immediately cut off the release phase when the velocity is patched into a VCA.
pitchbendrange	$\frac{1V}{Oct}$	$\frac{1}{6}V$	Sets the value to the desired maximum that pitchbend should output, and likewise its negative counterpart at its minimum value. At the middle position it always outputs 0. This defaults to $\frac{2}{12}V$, which corresponds to one whole tone. Note: setting this to a negative value is allowed and will invert pitch bend.
bendpitch		1	When set to 1 (which is the default), the pitch bend will directly be applied to all output pitches. Alternatively you can set it to 0 and use the output pitchbend , for using it elsewhere.

Input	Type	Default	Description								
roundrobin		0	<p>Normally when looking for a free output for playing the next note, this circuit will start from output1 in its search. This way, if there are not more notes than outputs at any time, the notes played first will always be played at the lowest numbered outputs. This leads to a deterministic behaviour when it comes to playing things like chords. The same voice will always be used for the first note in the stream of MIDI events.</p> <p>When you switch roundrobin to 1, this changes. Now the outputs are scanned in a round-robin fashion, like in a rotating switch. That way every output has the same chance to get a new note. Here it can even make sense to define multiple voices even if the track is monophonic. When you use envelopes with longer release times, you can transform such a melody into chords with simultaneous notes.</p> <p>Note: When all outputs are currently used by a note, roundrobin has no influence. Here voiceallocation selects which of the notes will be dropped.</p>								
voiceallocation	1◦2◦3	0	<p>When the MIDI stream, at any given time, needs to play more notes than you have voices assigned, normally the “oldest” notes would be cancelled. This behaviour can be configured here by setting voiceallocation to one of the following values:</p> <table border="1"> <tr> <td>0</td><td>The oldest note will be cancelled (default)</td></tr> <tr> <td>1</td><td>The new note will not be played and simply be omitted</td></tr> <tr> <td>2</td><td>The lowest note will be cancelled</td></tr> <tr> <td>3</td><td>The highest note will be cancelled</td></tr> </table>	0	The oldest note will be cancelled (default)	1	The new note will not be played and simply be omitted	2	The lowest note will be cancelled	3	The highest note will be cancelled
0	The oldest note will be cancelled (default)										
1	The new note will not be played and simply be omitted										
2	The lowest note will be cancelled										
3	The highest note will be cancelled										
notegap		0.0	<p>When your MIDI devices plays a note so “long” that it lasts exactly until the next note begins - or if due to a lack of used pitch outputs one currently played note has to be replaced with a new one, the gate output will have no time to go low for a sufficient time between the two notes. In effect it won’t trigger any envelope for the new note but will play “legato”.</p> <p>If you don’t like this, you can use notegap. This input specifies a number of milliseconds that the gate will be forced down before the new note begins. This has the drawback of introducing some latency, of course! So I suggest that you start with notegap = 1 and then check out if your envelope is fast enough to trigger. If not, increase the value.</p> <p>If you are using DROID’s own contour circuit or trigger something else internally in your patch, you can use notegap = 0.1. That is sufficient and introduces barely any latency. A value of 0.0 keeps the default of the legato mode.</p> <p>Note: the notegap parameter does not affect the trigger outputs.</p>								
ccnumber1 ... ccnumber4	1◦2◦3	0	<p>You can <i>listen</i> to up to four CCs (control changes). For example if you are interested in the current value of CC#17, set ccnumber1 = 17 and use the output cc1 for getting the value of CC 17.</p>								

Input	Type	Default	Description
lowestnote	1..2..3	0	With this input you can restrict the notes being played by setting a lower bound. In MIDI the notes range from 0 (C-2) to 127 (G9). By setting lowestnote to 24 (C0), all notes below this note are simply ignored. This allows for example for a keyboard split by using a second circuit with a highestnote of 23. Note gates are not being affected by this bound.
highestnote	1..2..3	127	Sets an upper limit to the note being played, similar to lowestnote . The “Notegates” are not being affected by this bound.
note1 ... note16	1..2..3		Selects up to 16 individual notes for which you can get a dedicated gate signal. Per default these values are set to 0 for note1 (meaning C-2), 1 for note2 (meaning C#-2) and so on. For each of these notes you get a corresponding gate output (see notegate1 , notegate2 , etc.). These gates are high as long as the selected notes are being held. One application is to use just one midifileplayer or midiiin circuit for sequencing up to 16 drum voices. Another application is to use a MIDI keyboard or controller as a button expander – just like a P2B8 or B32.

Output	Type	Description										
clockout		Outputs a steady clock of 1 tick per 16 th note.										
midiclock		Outputs a steady MIDI clock, i.e. 24 ticks per quarter note of the tune. This is 6 times faster than clock .										
endoftrack		Outputs a trigger when the end of the track is reached.										
error		This output will be set to a value other than zero in case of an error while loading and parsing the MIDI file. This is intended for wiring it to one of the R registers. Here different errors will be displayed as different colors. Here is the list of all possible values of error :										
<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0</td><td>black - Everything is fine.</td></tr> <tr> <td>-1</td><td>white - The SD card or MIDI file is missing.</td></tr> <tr> <td>1</td><td>magenta - The file is corrupted, garbled or no MIDI file.</td></tr> <tr> <td>0.75</td><td>orange - The file does not contain any non-empty track.</td></tr> <tr> <td>0.25</td><td>cyan - the track is too long (max 6000 bytes are allowed).</td></tr> </table>			0	black - Everything is fine.	-1	white - The SD card or MIDI file is missing.	1	magenta - The file is corrupted, garbled or no MIDI file.	0.75	orange - The file does not contain any non-empty track.	0.25	cyan - the track is too long (max 6000 bytes are allowed).
0	black - Everything is fine.											
-1	white - The SD card or MIDI file is missing.											
1	magenta - The file is corrupted, garbled or no MIDI file.											
0.75	orange - The file does not contain any non-empty track.											
0.25	cyan - the track is too long (max 6000 bytes are allowed).											
pitch1 ... pitch8	 $\frac{1V}{Oct}$	Pitch outputs. Since MIDI tracks can be polyphonic – i.e. play several notes at the same time – you can assign up to eight outputs here. The notes will be distributed to the defined outputs according to the settings roundrobin and voiceallocation .										
velocity1 ... velocity8		For each voice there is an optional velocity output, which translates the MIDI velocity into values from 0 to 1.										

Output	Type	Description
pressure1 ... pressure8	 1	MIDI provides two different messages for sending "after-touch" information, i.e. information about how strong a key is pressed down after the initial hit. Some keyboards just have one pressure sensor in total and send the current maximum pressure information of all keys in one message ("channel pressure"). Others have one pressure sensor per key and send "polyphonic key pressure" messages. This circuit maps both to a pressure output per note that is being played. So if your keyboard (or sequencer or DAW or whatever) sends polyphonic key pressure events and you use multiple pitchX outputs, wire the individual pressureX outputs to wherever you like. Otherwise you can simply use pressure1 for all notes (which can be abbreviated with pressure), since it is the same for all note outputs anyway. pressure outputs a value from 0 to 1.
gate1 ... gate8		Gate outputs for the up to eight simultaneous note outputs.
cc1 ... cc4	 1	Outputs the current value of the four CC number that are defined with the inputs ccnumber1 ... ccnumber4 . CCs have a range from 0 to 127, but this is converted in the range 0.0 .. 1.0 here, in order to make it easier to use that as a CV. If you need the raw number, multiply the output with 127. Note: as long as no CC message with the selected number happened, this output will be set to 0.
notegate1 ... notegate16		Outputs a high gate whenever the corresponding note (which is selected by note1 through note16) is currently being played.
pitchbend		Outputs the current pitch bend value as a bipolar voltage. The range can be set with pitchbendrange .
programchange		Sends a trigger whenever a <i>MIDI program change</i> message arrives. Just before sending the trigger sets program to the new program number (something from 0 to 127). Note: This trigger is also being output when the program change messages sends the same program number as previously, i.e. if there is no actual <i>change</i> .
program	1 ◊ 2 ◊ 3	The number of the last program change. This starts at 0 .
bank	1 ◊ 2 ◊ 3	Outputs the number of the currently selected bank - from 0 to 16384. MIDI defines the MSB of the bank to be changed with CC#0 and the LSB with CC#32. That means if you just use CC#0, you will only be able to select the banks 0, 128, 256, and so on. As long as no bank select CC has been received, bank will output 0.
modwheel	 1	Output the current state of the mod wheel level - within the range from 0.0 to 1.0 . The mod wheel is changed by MIDI control change 1.
volume	 1	Outputs the current global volume as set by MIDI control change 7.
portamento		This output gives you access to the current state of the "portamento pedal" (MIDI CC 65). You can use it to enable an external slew circuit for creating portamento effects.
soft		This output gives you access to the current state of the "soft pedal" (MIDI CC 67). It is 1 while the pedal is hold and 0 otherwise.

13.33 midiin - MIDI to CV converter

This circuit converts incoming MIDI data into CV, gate and trigger signals. It needs the X7 expander in order to work (see page 49 for general information about the X7).



There are various useful applications of this circuit, some of which are:

- Attaching an external keyboard to your modular.
- Using an external hardware sequencer for playing melodies and beats in your modular.
- Use an external MIDI controller to influence your DROID patch.
- Use your phone or tablet as a MIDI controller to influence your patch (via USB).
- Connect two DROIDS (both with X7) and exchange real time data.

The X7 MIDI implementation is very comprehensive and gives you convenient access to most of the MIDI features. Please refer to the table of inputs and outputs for details. Here are just some very basic examples:

Basic operation

The basic operation is quite simple. Per default midiin listens on the 3.5 mm TRS jack of the X7. The following example controls one synth voice by converting MIDI note on / note off messages into CV / gate signals:

```
[midiin]
pitch = 01
gate = 02
```

It's really as simple as that! Connect your MIDI keyboard or sequencer with the X7 MIDI input, wire 01 to the

1V/Oct input of a synth voice and 02 to its gate input and enjoy your music!

When you add `usb = 1` you can get a MIDI stream via the USB-C port on the X7 instead of the TRS jack.

Polyphonic patches

Do you have more than one synth voice to control? Then you can play several notes at the same time by using up to *eight pitch* and *gate* outputs. Here is an example with three voices, which uses a G8 expander for the gates:

```
[midiin]
pitch1 = 01
pitch2 = 02
pitch3 = 03
gate1 = G1
gate2 = G2
gate3 = G3
```

Here the parameters `roundrobin` and `voiceallocation` are interesting. `roundrobin` influences which of the three outputs should be used for the next note, in situations where more than one is free. `voiceallocation`, in contrast, controls what should happen if the MIDI stream wants to play more simultaneous notes than you have setup in `midiin`. The default is to cancel the oldest currently playing note, but you can change that behaviour in various ways.

Sequencing drums and triggers

When you use a MIDI sequencer for triggering drums, often each drum voice (bass drum, snare drum, etc.) is triggered by a certain note, for example C-2 for the bass drum, C#2 for the snare drum and so on. In this case it is more convenient to use the `notegate` outputs. Check the following example:

```
[midiin]
note1 = 24
note2 = 25
notegate1 = 01
notegate2 = 02
```

Now whenever note 24 is played by the sequencer, `notegate1` will trigger. The note numbers range from 0 to 127, with 0 being the lowest note and 127 the highest. The MIDI standard specifies that note 0 is usually C-2 (two octaves below C0). So note 24 would be C0 and note 25 C#0.

Another application of note gates is to use keys on a MIDI keyboard or touch pads of a MIDI controller as buttons in your DROID patch! In fact the `button` circuit can be wired to such note gates. It's just that you don't have a corresponding LED. But you can use the DROID's own LEDs for that.

The following example uses the note 24 in order to toggle a (virtual) button and use the first input LED of the master as LED for the button:

```
[midiin]
note1 = 24
notegate1 = _NOTE24
```

```
[button]
  button = _NOTE24
  led = R1
  output = _SOMETHING # ...
```

Please note: **midout** has similar **note1** ... **note8** inputs. But there the pitches are specified in 1V/Oct. So don't mix them up!

Start, Stop and Clock

MIDI sequencers usually send a steady MIDI clock at 24 PPQ, which means 24 pulses per quarter note, which in turn means 6 pulses per 16th note, which is the typical clock speed for modular systems. But also 48 PPQ and 96 PPQ are possible.

You get easy access to the clock by various clock outputs running at different speeds. The jack labelled just **clock** outputs the 16th note clock. The following example just sends that clock to the O1 output:

```
[midiin]
  clock = 01
```

Hereby it is assumed that the MIDI clock is running at 24 PPQ. If its running faster, simply use one of the other clock outputs, which divides down the clock. Or use **clocktool** (see page 121) for dividing yourself.

Also the START and STOP messages of MIDI sequencers are accessible, either as two separate triggers, or as a running state. For example you can use the **start** output as a reset signal for some **DROID** circuit:

```
[midiin]
  clock = _CLOCK
  start = _RESET
```

```
[sequencer]
  clock = _CLOCK
  reset = _RESET
  ...
```

Getting CCs

MIDI does not only transport note events but also *controllers*. Most of these are continuous values, much like CVs. **midiin** gives you access to the current value of a couple of standard controllers like **volume** and **modwheel** with dedicated outputs. And in addition up to four custom CCs can be output. All such controllers are converted into values from 0 to 1 (or 0 V to 10 V if you output them directly):

```
[midiin]
  volume = 01
  modwheel = 02
  ccnumber1 = 10 # get update from CC#10
  cc1 = 03 # send current CC value to 03
```

Using multiple midiins

You are not restricted to one **midiin** circuit but can use up to 32 of these in your patch. There are different reasons why multiple ones can be useful, e.g.:

- You want to control different voices from different MIDI channels
- You want to fetch more than four CCs.

All **midiin** circuits will get their own copy of the MIDI data stream and can do their own things with it. You might want to use **channel = ...** in order to just get only the events of a specific MIDI channel.

Pedals

The MIDI standard defines five different types of foot pedals. The state of these - up or down - is transmitted by means of five different control changes (CCs). **midiin** automatically interpretes them corresponding to their intended meaning as follows:

- *Damper pedal* (CC 64): While down, notes still linger on, even if they end. Internally, the "note off" event of all notes will be delayed until the pedal is up. This pedal is sometimes also called "sustain pedal", since it makes notes sustain.
- *Portamento pedal* (CC 65): Sets the **portamento** output to 1 while down. You can use that output for enabling a slew limiter with the circuit **slew** (see page 263).
- *Sostenuto pedal* (CC 66): Sostenuto is the smarter version of sustain. Such a pedal is found as the middle of three pedals on grand pianos. When it goes down, all notes that are *currently played* are sustained as long as the pedal is held. But *new* notes, that start during that period, at *not* sustained. That's the difference. The **midiin** circuit automatically makes CC 66 behave in exactly that way. That, of course, just makes sense in a polyphonic patch, where you have enough voice that can play the sustained notes.
- *Soft pedal* (CC 67): Sets the **soft** output to 1 while held.
- *Legato pedal* (CC 68): While down, ties consecutive notes together by keeping **gate** at 1 between notes.

Input	Type	Default	Description						
usb		0	Selects the physical port to receive MIDI data. The default is usb = 0 , which selects the TRS (3.5mm stereo jack) port of the X7. Set usb = 1 for receiving data from the USB-C port.						
initialrunning	1 2 3	2	<p>This parameter sets which “running” state is assumed when your DROID starts. The idea behind this parameter is, that at this point of time you cannot know the real running state of the MIDI stream, since e.g. the DROID might have started after the sequencer at the sending end of the line.</p> <p>You have three ways to set this: start in stopped state, start in running state and an inbetween “automatic” mode. In the auto mode, you start in stopped state but automatically switch to running as soon as a note on event is received. At that moment a MIDI START event is simulated.</p> <table border="1"> <tr> <td>0</td><td>Start stopped state</td></tr> <tr> <td>1</td><td>Start in running state</td></tr> <tr> <td>2</td><td>Automatic: start in stopped state, switch to running on first “note on”</td></tr> </table> <p>Note: as this parameter is just read once the absolute system start, you cannot assign a dynamic CV input or control here.</p>	0	Start stopped state	1	Start in running state	2	Automatic: start in stopped state, switch to running on first “note on”
0	Start stopped state								
1	Start in running state								
2	Automatic: start in stopped state, switch to running on first “note on”								
systemreset			A trigger here resets the whole MIDI state of this circuit. It does the same as a MIDI RESET message: It stops all playing note, resets the controllers, the states of the pedals and so on.						
channel	1 2 3		Only execute / play commands from a certain MIDI channel. There are 16 MIDI channels. It ranges from 1 to 16 .						
tuningmode		off	If set to 1 , all pitch outputs will go to the CV selected for tuningpitch (which defaults to 2 V), and all gate outputs will play gates at 120 BPM. This helps getting all attached voices tuned when working with many voices.						
tuningpitch	$\frac{1V}{Oct}$	2V	This pitch CV will be output while the tuning mode is active.						
transpose	$\frac{1V}{Oct}$	0V	Transposes all output pitches by this value by adding the value. So in order to transpose one octave down, set this input to -1V or -0.1 . Changes in the transposition are immediately reflected, even for currently already active notes.						
holdvelocity		0	If this is set to 1 , the velocity output for a voice will not be affected by note off events. It's just altered at the beginning of new notes. The velocity is kept after the note ends. This way during the release phase of an envelope triggered by the gate, the original velocity still lasts on. In most cases the note off velocity is set to 0, which would immediately cut off the release phase when the velocity is patched into a VCA.						
pitchbendrange	$\frac{1V}{Oct}$	$\frac{1}{6}V$	Sets the value to the desired maximum that pitchbend should output, and likewise its negative counterpart at its minimum value. At the middle position it always outputs 0. This defaults to $\frac{2}{12}V$, which corresponds to one whole tone. Note: setting this to a negative value is allowed and will invert pitch bend.						

Input	Type	Default	Description								
bendpitch		1	When set to 1 (which is the default), the pitch bend will directly be applied to all output pitches. Alternatively you can set it to 0 and use the output pitchbend , for using it elsewhere.								
roundrobin		0	<p>Normally when looking for a free output for playing the next note, this circuit will start from output1 in its search. This way, if there are not more notes than outputs at any time, the notes played first will always be played at the lowest numbered outputs. This leads to a deterministic behaviour when it comes to playing things like chords. The same voice will always be used for the first note in the stream of MIDI events.</p> <p>When you switch roundrobin to 1, this changes. Now the outputs are scanned in a round-robin fashion, like in a rotating switch. That way every output has the same chance to get a new note. Here it can even make sense to define multiple voices even if the track is monophonic. When you use envelopes with longer release times, you can transform such a melody into chords with simultaneous notes.</p> <p>Note: When all outputs are currently used by a note, roundrobin has no influence. Here voiceallocation selects which of the notes will be dropped.</p>								
voiceallocation	1 ◦ 2 ◦ 3	0	<p>When the MIDI stream, at any given time, needs to play more notes than you have voices assigned, normally the “oldest” notes would be cancelled. This behaviour can be configured here by setting voiceallocation to one of the following values:</p> <table border="1"> <tr> <td>0</td><td>The oldest note will be cancelled (default)</td></tr> <tr> <td>1</td><td>The new note will not be played and simply be omitted</td></tr> <tr> <td>2</td><td>The lowest note will be cancelled</td></tr> <tr> <td>3</td><td>The highest note will be cancelled</td></tr> </table>	0	The oldest note will be cancelled (default)	1	The new note will not be played and simply be omitted	2	The lowest note will be cancelled	3	The highest note will be cancelled
0	The oldest note will be cancelled (default)										
1	The new note will not be played and simply be omitted										
2	The lowest note will be cancelled										
3	The highest note will be cancelled										
notegap		0.0	<p>When your MIDI devices plays a note so “long” that it lasts exactly until the next note begins – or if due to a lack of used pitch outputs one currently played note has to be replaced with a new one, the gate output will have no time to go low for a sufficient time between the two notes. In effect it won’t trigger any envelope for the new note but will play “legato”.</p> <p>If you don’t like this, you can use notegap. This input specifies a number of milliseconds that the gate will be forced down before the new note begins. This has the drawback of introducing some latency, of course! So I suggest that you start with notegap = 1 and then check out if your envelope is fast enough to trigger. If not, increase the value.</p> <p>If you are using DROID’s own contour circuit or trigger something else internally in your patch, you can use notegap = 0.1. That is sufficient and introduces barely any latency. A value of 0.0 keeps the default of the legato mode.</p> <p>Note: the notegap parameter does not affect the trigger outputs.</p>								

Input	Type	Default	Description
ccnumber1 ... ccnumber4	1°2°3	0	You can <i>listen</i> to up to four CCs (control changes). For example if you are interested in the current value of CC#17, set ccnumber1 = 17 and use the output cc1 for getting the value of CC 17.
lowestnote	1°2°3	0	With this input you can restrict the notes being played by setting a lower bound. In MIDI the notes range from 0 (C-2) to 127 (G9). By setting lowestnote to 24 (C0), all notes below this note are simply ignored. This allows for example for a keyboard split by using a second circuit with a highestnote of 23. Note gates are not being affected by this bound.
highestnote	1°2°3	127	Sets an upper limit to the note being played, similar to lowestnote . The “Notegates” are not being affected by this bound.
note1 ... note16	1°2°3		Selects up to 16 individual notes for which you can get a dedicated gate signal. Per default these values are set to 0 for note1 (meaning C-2), 1 for note2 (meaning C#-2) and so on. For each of these notes you get a corresponding gate output (see notegate1 , notegate2 , etc.). These gates are high as long as the selected notes are being held. One application is to use just one midifileplayer or midin circuit for sequencing up to 16 drum voices. Another application is to use a MIDI keyboard or controller as a button expander - just like a P2B8 or B32.

Output	Type	Description
clock		If the MIDI sender sends a MIDI clock, you get a 16 th note clock output here. This is the same as the clock16 jack and just a convenient abbreviation.
clock8		Gets an 8 th clock here (like clock divided by 2)
clock8t		Gets a 8 th triplets clock here. This is faster than clock8 but slower than clock .
clock16		The same as clock : a clock running at 16 th notes.
clock4		A clock at the speed of quarter notes.
midiclock		Here you get the original MIDI clock. This is 6 times faster than clock and 24 times faster than clock4 . This is because the MIDI clock is specified to run at 24 PPQ, i.e. 24 pulses per quarter note.
start		This jack sends a trigger when a MIDI START message arrives.
continue		This jack sends a trigger when a MIDI CONTINUE message arrives.
stop		This jack sends a trigger when a MIDI STOP message arrives.
running		This jack remembers the current running state according to previous START and STOP messages.
active		If the sending device supports active sensing , this output is high as long as a device is connected. Otherwise its high if at least one MIDI message has been received.

Output	Type	Description
pitch1 ... pitch8	 $\frac{1V}{Oct}$	Pitch outputs. Since MIDI tracks can be polyphonic - i.e. play several notes at the same time - you can assign up to eight outputs here. The notes will be distributed to the defined outputs according to the settings roundrobin and voiceallocation .
velocity1 ... velocity8		For each voice there is an optional velocity output, which translates the MIDI velocity into values from 0 to 1.
pressure1 ... pressure8		MIDI provides two different messages for sending "after-touch" information, i.e. information about how strong a key is pressed down after the initial hit. Some keyboards just have one pressure sensor in total and send the current maximum pressure information of all keys in one message ("channel pressure"). Others have one pressure sensor per key and send "polyphonic key pressure" messages. This circuit maps both to a pressure output per note that is being played. So if your keyboard (or sequencer or DAW or whatever) sends polyphonic key pressure events and you use multiple pitchX outputs, wire the individual pressureX outputs to wherever you like. Otherwise you can simply use pressure1 for all notes (which can be abbreviated with pressure), since it is the same for all note outputs anyway. pressure outputs a value from 0 to 1.
gate1 ... gate8		Gate outputs for the up to eight simultaneous note outputs.
cc1 ... cc4		Outputs the current value of the four CC number that are defined with the inputs ccnumber1 ... ccnumber4 . CCs have a range from 0 to 127, but this is converted in the range 0.0 .. 1.0 here, in order to make it easier to use that as a CV. If you need the raw number, multiply the output with 127. Note: as long as no CC message with the selected number happened, this output will be set to 0.
notegate1 ... notegate16		Outputs a high gate whenever the corresponding note (which is selected by note1 through note16) is currently being played.
pitchbend		Outputs the current pitch bend value as a bipolar voltage. The range can be set with pitchbendrange .
programchange		Sends a trigger whenever a <i>MIDI program change</i> message arrives. Just before sending the trigger sets program to the new program number (something from 0 to 127). Note: This trigger is also being output when the program change messages sends the same program number as previously, i.e. if there is no actual <i>change</i> .
program	1 ◊ 2 ◊ 3	The number of the last program change. This starts at 0 .
bank	1 ◊ 2 ◊ 3	Outputs the number of the currently selected bank - from 0 to 16384. MIDI defines the MSB of the bank to be changed with CC#0 and the LSB with CC#32. That means if you just use CC#0, you will only be able to select the banks 0, 128, 256, and so on. As long as no bank select CC has been received, bank will output 0.
modwheel		Output the current state of the mod wheel level - within the range from 0.0 to 1.0 . The mod wheel is changed by MIDI control change 1.
volume		Outputs the current global volume as set by MIDI control change 7.

Output	Type	Description
portamento		This output gives you access to the current state of the “portamento pedal” (MIDI CC 65). You can use it to enable an external slew circuit for creating portamento effects.
soft		This output gives you access to the current state of the “soft pedal” (MIDI CC 67). It is 1 while the pedal is hold and 0 otherwise.

13.34 midiout - CV to MIDI converter

This circuit allows you to “play” notes via MIDI on an external hardware or software synth. You also can send all sorts of other MIDI events. You need the X7 expander for that to work (see page 49).



The MIDI implementation of **midiout** is very comprehensive. Please look at the table of input jacks for all features. Here I just want to show some basic examples to get you started quickly. Fun fact: This is the only circuit that does not have any outputs, because all output is done via MIDI!

Basic operation

Easy things should be easy and complex things should be possible. So we start with the easy things. Here is a patch that converts a CV / gate input from **I1** / **I2** into a stream of MIDI notes and sends them out via the 3.5 mm TRS jack on MIDI channel 1:

```
[midiout]  
pitch = I1  
gate = I2
```

Every time the gate input at **I2** goes from off to on, the current pitch (1V/Oct) is read from **I1**. Then one MIDI “note on” event is being created. The “velocity” of that note is set to the default value of 1.0, which is the maximum (every MIDI note event has a velocity, which is meant to reflect the speed at which the key of the keyboard has been pressed).

You can specify any velocity you like with the jack **velocity**. Let’s randomize that. Since the velocity jack

is just read just at the note starts, we don’t need a sample and hold here:

```
[random]  
minimum = 0.5 # minimum allowed velocity  
maximum = 1.0 # maximum allowed velocity  
output = _VELOCITY  
  
[midiout]  
pitch = I1  
gate = I2  
velocity = _VELOCITY
```

Note: the range of the velocity goes from 0.0 to 1.0 – just as all other parameters in **midiout** do. Internally MIDI uses the integer numbers 0 to 127.

Output selection

You can send your MIDI stream either via the 3.5 mm TRS jack of the X7 (TRS stands for “tip ring sleeve” – the structure of the stereo 3.5 mm plug) or via the USB-C port. This is controlled by the parameters **usb** and **trs**.

Per default the stream is sent via TRS. As soon as you use either **usb** or **trs** you set this explicitly. Here is a complete table of all possible usages of these inputs (empty cells mean that the parameter is not used):

		Uses TRS only (default)
usb = 1		Uses USB only
usb = 0		Uses TRS only (default)
	trs = 1	Uses TRS only (default)
	trs = 0	Uses USB only
usb = 0	trs = 1	Uses TRS only (default)
usb = 1	trs = 0	Uses USB only
usb = 1	trs = 1	Uses both TRS and USB
usb = 0	trs = 0	Mute! does not send MIDI.

Note: MIDI via USB has a much higher data rate than via TRS. If you use both USB and TRS at the same time, USB will run at the same (lower) data rate as TRS. This might lead to fewer updates for CCs and similar. The reason is that the **midiout** circuit does not make a separate book keeping for USB and TRS but creates just one common MIDI data stream. If that’s an issue for you, duplicate your **midiout** circuit and create one instance for TRS and one for USB. Then they create two separate MIDI streams that are optimized for the specific maximum data rates of their output ports.

Polyphonic patches

One great motivation for doing CV to MIDI at all is playing polyphonic music on hardware synths, because polyphony in Eurorack is quite costly and very time and space consuming. One **midiout** circuit can play up to eight notes at the same time and if that’s not enough, add a second **midiout** circuit. For each simultaneous note add one pair of **pitch** and **gate** jacks:

```
[midiout]
pitch1 = I1
pitch2 = I2
pitch3 = I3
gate1 = I5
gate2 = I6
gate3 = I7
```

If you work with velocity, each voice has its own velocity input:

```
[midiout]
pitch1 = I1
pitch2 = I2
pitch3 = I3
gate1 = I5
gate2 = I6
gate3 = I7
velocity1 = 0.6
velocity2 = 0.8
velocity3 = 1.0
```

CC and other controllers

There are several continuous values that you can change over time. The following example lets you control the MIDI CC number 17 via input **I3** (at a range from 0 V to 10 V) and the volume and modulation wheel with two pots:

```
[midiout]
pitch = I1
gate = I2
ccnumber1 = 17
cc1 = I3
volume = P1.1
modwheel = P1.2
```

Note gates

Note gates are a convenient way to directly trigger certain notes. Here you select up to eight notes and get one dedicated trigger for each. You select the note number with **note1**, **note2**, etc. These are MIDI note numbers from 0 to 127, where 0 is usually a C-2 (and 24 a C0). When you send a trigger into the corresponding **notegate** input, that note will be played.

```
[midiout]
note1 = 24
note2 = 25
notegate1 = I1
notegate2 = I2
```

This is sometimes convenient when triggering drum voices.

Creating a MIDI clock

If you want to simulate a MIDI sequencer, you need to provide a MIDI clock. This can be injected into the output either by sending a modular clock that is running on 16th notes into **clock**, or a raw MIDI clock into **midiclock**.

Example: You want your clock to run at 120 BPM. BPM means beats per minute. And a beat is meant to be a quarter note. 120 quarter notes a minute means two quarter notes a second and that means eight 16th notes a second, hence our clock needs to run at 8 Hz.

```
[lfo]
hz = 8 # 120 BPM
square = _CLOCK

[midiout]
clock = _CLOCK
```

Note: The input jack **clock** receives 16th clocks. The actual MIDI clock is derived from that by multiplying it by 6. This means that the circuit interpolates the clock by measuring its speed and introducing five artificial clock ticks inbetween the original ticks. While this works reasonably well for a steady clock, changes in clocks speed cannot be picked up very fast.

So if you work with a clock that can change the speed, better use the jack **midiclock** instead and directly supply the MIDI clock (at a six times higher speed). Here is the same example but now we directly create the MIDI clock:

```
[lfo]
hz = 48 # 120 BPM MIDI clock
square = _MIDICLOCK

[midiout]
midiclock = _MIDICLOCK
```

Start, Stop, Reset

MIDI sequencers also output “start” and “stop” messages. You can send them either via triggers into **start** and **stop** or use the input **running** for both. When running goes high, a “start” message is sent, when it goes low a “stop” message.

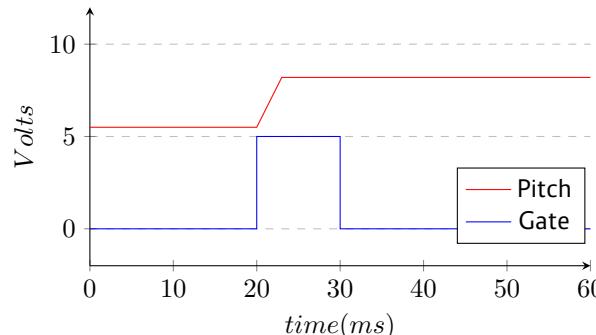
Pitch tracking

Pitch tracking is an advanced feature that works in monophonic setups. Here **midiclock** watches the input pitch all the time and adapts the pitch of the currently played note via MID pitchbend events in order to reflect the pitch changes. See the documentation of the **pitchtracking** jack for details.

Pitch stabilization

MIDI output appears simple to implement, but isn't when you look at the details. One tricky problem is that many modules that output pitch information are not very precise in timing. Sequencers often need a couple of milliseconds for the pitch CV to reach its final value and stabilize after the gate is being output.

The following diagram shows a gate signal going high (blue) and a pitch signal with a small ramp reaching its final destination shortly afterwards (red):



I've seen a very similar situation indeed when I attached an oscilloscope to the output of a very famous Eurorack sequencer.

Now when you would issue "note on" right at the beginning of the gate, you would obviously output the wrong pitch. What you need to do is to first *wait* for some time. You need to *delay* the note event until the pitch is stable. Of course this introduces some undesirable latency, so it is crucial to keep that as short as possible.

The **midout** circuit has two methods for doing this. The first one is enabled per default and called **pitchstabilization**. Here, as soon as the gate goes high, it watches how **pitch** evolves over time. And it delays the "note on" as long as the pitch is still mov-

ing. When it has stabilized - i.e. on the same level for at least some very short time - the note event is issued immediately. This keeps the latency at a minimum.

If that does not work out well for you, you can deactivate this algorithm. One reason could be that your pitch *never* stabilizes, since it is some ever evolving random data:

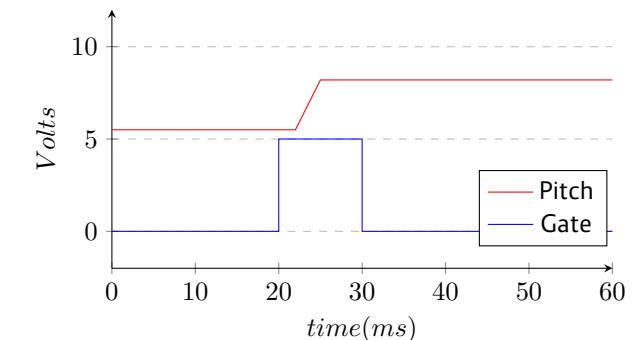
```
[midout]
pitch = I1
gate = I2
pitchstabilization = 0
```

The second method is introducing a fixed delay of the gate signal with the input **triggerdelay**. Using that parameter automatically disables pitch stabilization:

```
[midout]
pitch = I1
gate = I2
triggerdelay = 3.5 # delay gate by 3.5 ms
pitchstabilization = 1
```

Now the gate is delayed *exactly* 3.5 ms every time. You need to try out various useful values yourself. The best value depends on your sequencer (or whatever other source you are using).

You can also activate both methods at once. This makes sense in situations, where the pitch is stable for a very short time after the gate but afterwards begins to move, like in the following diagram:



As you can see, now after the gate comes high the pitch lingers on for 2 ms at its old value until the ramp starts. Here set the **triggerdelay** to 2 and explicitly set **pitchstabilization = 1**:

```
[midout]
pitch = I1
gate = I2
triggerdelay = 2
pitchstabilization = 1
```

Sending notes by number

If you are familiar with MIDI, you sometimes might want to send a certain note *number* rather than a pitch. MIDI knows notes from 0 (C-2) to 127. To do this, divide your number by 120 before sending it to **pitch**.

```
[midout]
pitch = _SOMENUMBER / 120
gate = _SOMEgate
```

Why not 127? Because the pitch input counts notes by semitones. And one semitone in modular is $\frac{1}{12}$ V, which in Droid means $\frac{1}{120}$. Dividing by 127 will be slightly off and send wrong note numbers.

Input	Type	Default	Description
channel	1 ⋅ 2 ⋅ 3	1	Selects the MIDI channel to send the events on. Default is to send on channel 1. There are 16 channels. Make sure that the receiving device listens to this (or to all) channels.
usb		0	Set usb = 1 if you want to send the MIDI output to the USB-C port. You can set trs = 1 , as well, for sending the data to both outputs. If you don't use usb nor trs , the output will be sent to the TRS output only.
trs		1	This controls whether the MIDI data is sent via the TRS output of the X7. If you just want the TRS output, you don't need this, because that is the default. If you want the output both on USB and TRS, you need to set usb = 1 and trs = 1 at the same time.
pitch1 ... pitch8	$\frac{1V}{Oct}$	0V	Pitch of the notes to be played in modular style (1 V/octave). The range is from -2 V (MIDI note 0, usually C-2) to 8.583 V (MIDI note 127, usually G9). You can use up to eight pitch inputs for playing up to eight notes in parallel. pitch1 can be abbreviated with just pitch .
gate1 ... gate8			A positive edge into the gate jacks trigger note on messages (starts the note at the pitch set by the corresponding pitch input). A negative edge ends the currently played note.
velocity1 ... velocity8	$0 \circlearrowleft 1$	1.0	The velocities for the up to eight notes. The velocity value is just picked up at the start of the note (at the positive edge of the corresponding gate inputs). It ranges from 0.0 to 1.0. A value of 0.0 is practically the same as "note off". The default velocity is 1.0.
noteoffvelocity1 ... noteoffvelocity8	$0 \circlearrowleft 1$		MIDI also sends a velocity at the <i>end</i> of a note. The idea is to model the speed with which a key is being <i>released</i> . This is rarely used. If you don't use these jacks, the velocity for "note off" events is the same as that for "note on" events.
pressure1 ... pressure8	$0 \circlearrowleft 1$		Sends key pressure events for individually played notes via the MIDI event "polyphonic key pressure" (this is not a CC!). These values are not processed at the time of note on/off events but all the time and can also change while a note is already being played. This corresponds to "aftertouch" key pressure on keyboards that have a pressure sensor <i>per key</i> . If nothing is patched here, no pressure events are sent.
channelpressure	$0 \circlearrowleft 1$		Whenever this CV changes, sends a MIDI channel pressure event, also known as "aftertouch". This corresponds to keyboards that just have one global pressure sensor and not one per key. If nothing is patched here, no channel pressure events are sent.
pitchstabilization		1	Enables or disables pitch stabilization. It is on by default and can be disabled by setting this jack to 0. Pitch stabilization fixes timing issues where the input pitch needs some time for reaching the target pitch after a gate.

Input	Type	Default	Description
triggerdelay		0.0	Introduces a delay between the incoming gate signal (just the positive edge) and the “note on” event. This can tackle the problem when your pitch input (sequencer etc.) needs some time after the gate in order to reach and stabilize the target pitch. The delay is specified in milliseconds, so a typical useful value would be 5 (5 ms). This is an alternative to the automatic pitchstabilization . Note: triggerdelay disables pitchstabilization , as long as that is not set to 1 explicitly. If both are used at the same time, the triggerdelay happens <i>before</i> the pitch stabilization. So it is a <i>minimum</i> delay.
lowestnote	1°2°3	0	With this input you can restrict the notes being played by setting a lower bound. In MIDI the notes range from 0 (C-2) to 127 (G9). By setting lowestnote to 24 (C0), all notes below this note are simply ignored. This allows for example for a keyboard split by using a second circuit with a highestnote of 23. Note gates are not being affected by this bound.
highestnote	1°2°3	127	Sets an upper limit to the note being played, similar to lowestnote . Note gates are not being affected by this bound.
notegate1 ... notegate16			You can define up to 16 notes that can be directly controlled with a dedicated gate. This is convenient for playing drum sounds directly from triggers and also for using DROID controllers as MIDI controllers. A trigger or gate to notegate1 will directly play the note whose pitch is set by note1 .
note1 ... note16	1°2°3		MIDI notes to be played via notegate . The range is from 0 to 127. Per default the notes are set to the MIDI notes 0, 1, 2 ... 15.
notegatevelocity1 ... notegatevelocity16		1.0	Here you can set the velocities used by the notegates. In order to keep it simple, this velocity is used for note on <i>and</i> note off events (nobody cares about the note off velocity anyway). If you do not use these jacks, the note gates will always use the maximum velocity.
modwheel		0.0	Sets the current value of the modulation wheel. Any change here sends a midi CC#1 with a new value for the modulation wheel. The input range is 0.0 ... 1.0 and will be converted into the MIDI range of 0 ... 127. Note: in future we might support CC#33, which is the LSB value of CC#1 and increases the resolution from 128 to 16384 different values, at the cost - however - of two additional bytes being sent.
volume		1.0	Sets the volume of the target device. This is done by sending the MIDI CC#7 (VOLUME MSB) and MIDI CC#39 (VOLUME LSB). Using these two CCs enables a 14 bit high resolution 16384 levels (not just 127). Some devices do not react to CC#39 and simply ignore the LSB (least significant byte). The volume CV ranges from 0.0 (silent) to 1.0 (the default).
pitchbend		0.0	Bends the pitches of <i>all</i> currently played notes up and down by a range that is configured or elsewhere defined by the device that plays our stuff. The range of this CV is -1.0 ... 1.0 for covering the maximum pitch bend range. Most times that range is two semitones up and down. This CV does <i>not</i> behave in a 1V/oct way!

Input	Type	Default	Description						
pitchtracking	1◦2◦3	0	<p>Pitch tracking is an advanced feature that allows you to track continuous changes in the incoming pitch CV <i>while the note is already playing</i>. It does this by listening to the input CV and converting any change into a MIDI “pitch bend” change.</p> <p>This feature has two limitations: First, there is just one global pitch bend value per channel, not one per note. So this feature only works in a monophonic situation. Only the value of pitch1 is being tracked. When you play more than one note per channel, funny things might probably happen. Also The maximum range is limited by the pitch bend range of your target device. That is usually preset to 2 semitones up and down. If you can increase it, please also adapt pitchbandrange so this circuit knows about it.</p> <p>Pitch tracking has two levels: pitchbandrange = 1 will alter the pitch of the current note within the maximum range of pitch bend and will clip any further changes. pitchbendrange = 2, in contrast, plays a new note if the current range is exceeded. Depending on your sound settings this “dent” might be audible or not.</p> <table border="1"> <tr> <td>0</td><td>pitch tracking is off</td></tr> <tr> <td>1</td><td>just use MIDI pitch bend</td></tr> <tr> <td>2</td><td>use new note on larger changes</td></tr> </table>	0	pitch tracking is off	1	just use MIDI pitch bend	2	use new note on larger changes
0	pitch tracking is off								
1	just use MIDI pitch bend								
2	use new note on larger changes								
			Note: When you use pitch tracking at the same time as pitchbend , both pitch alterations will add up.						
pitchbendrange	 $\frac{1V}{Oct}$	$\frac{1}{6}V$	<p>Defines the range of the effect of pitch bend at the target device on a 1V/oct base. Note: You cannot <i>change</i> that actual range here. You just can make sure that this circuit has the correct assumption of that range.</p> <p>If your target device has a configuration for extending the range, and you have set that for example to 1 octave, set pitchbendrange to 1 V. This allows pitchtracking to correctly adapt in-note pitch changes. Note: This has <i>no effect</i> on the pitchbend CV.</p>						
ccnumber1 ... ccnumber8	1◦2◦3	0	Specifies up to eight different CC numbers that can be continuously updated via the corresponding cc1 through cc8 inputs. The value needs to be an integer number from 0 to 127 .						
cc1 ... cc8			<p>The current value of the CCs that are specified with ccnumber1 through ccnumber8. The range is always from 0.0 to 1.0 (which is mapped to the number 0 to 127 on the MIDI wire).</p> <p>If you don't patch anything here, no CC events will be sent, of course.</p>						

Input	Type	Default	Description
cctrigger1 ... cctrigger8			<p>Usually midout will send out a new CC event every time the input value of a CC has changed (with some rate limit in order not to flood the MIDI stream).</p> <p>When you use these inputs, an alternative method is enabled. Now CC events are created whenever a trigger arrives here. No more updates will be sent automatically.</p> <p>This is useful for target devices that use CCs just as <i>messages</i>, i.e. as one time events and not for updating a continuous value.</p>
updateccs			<p>A trigger here sends an update for all CCs that you have in use (used ccX inputs). Normally an update is just sent once initially and then when the input CV at one of the cc inputs changes its value. With the trigger you can force updates. This might be necessary if the receiving device has lost memory of the current states of the CCs (e.g. due to a power cycle).</p> <p>Note: Other than the cctriggerX inputs, this trigger does <i>not</i> change the way the CC inputs work. It is just a hint for DROID that forces one additional update.</p>
delayinitialccs		1.0	<p>When the Droid starts it needs a short time until the X7 is operating and your PC / DAW is able to receive the MIDI events via USB. Initial CC updates during that short time period might get lost and you are missing the correct CC states (which are updated later only on changes).</p> <p>In order to avoid that, the Droid waits a short time after starting before it sends the first CC events. That delay can be tuned here. It is a time in seconds.</p>
bank	1 ◊ 2 ◊ 3		Selects the current “bank”. Some MIDI devices have more than 128 programs (i.e., patches, instruments, preset, etc). A MIDI Program Change message supports switching between only 128 programs. So, “Bank Select” (sometimes also called bank switch) is sometimes used to allow switching between groups of 128 programs. Bank select uses the MIDI CCs #0 (MSB) and #32 (LSB) together to form a number of 16384 different banks. The input value thus ranges from 1 to 16384. Most devices, however, restrict themselves to just 128 banks and just use the MSB (CC#0). If that is the case, you need to set bank to 128 for bank 2, 256 for bank 3 and so on. This can be done by simply multiplying the actual bank number with 128.
program	1 ◊ 2 ◊ 3		Select the current “program”. This is a number from 1 to 128 .
programchange			A trigger here will send out a “program change” MIDI message even if the value of bank or program has not changed.
start			If you send a trigger here, the MIDI message START will be emitted. Don’t use this jack if you also use running . Note: START/STOP messages are not bound to a specific channel.
stop			If you send a trigger here, the MIDI message STOP will be emitted. Don’t use this jack if you also use running . Note: START/STOP messages are not bound to a specific channel.

Input	Type	Default	Description
<code>running</code>			This is an alternative to the jacks <code>start</code> and <code>stop</code> . It combines both into one “running” state. When this gate input goes high, a START message is sent, when it goes low a STOP message. So you can work with a state rather than with state changes. Note: START/STOP messages are not bound to a specific channel.
<code>systemreset</code>			A trigger here will send the MIDI real-time message “RESET”, that is supposed to bring the device into some start state.
<code>allnotesoff</code>			A trigger here will send the MIDI CC#123 “ALL NOTES OFF”, which is essentially the same as releasing all currently held keys.
<code>allsoundoff</code>			A trigger here will send the MIDI CC#120 “ALL SOUND OFF”, which is supposed to make the device silent as soon as possible.
<code>damper</code>		<code>0</code>	This gate input simulates a hold or damper pedal. This is done via the CC#64. If the gate goes to high, a value of 127 is being sent, when it goes back to low, a value of 0. When the damper pedal is pressed, the device is supposed to hold all currently played notes and not react to any subsequent “NOTE OFF” of <i>those notes</i> as long as the pedal is held. When the pedal is released, all notes that had been held by the pedal should be released.
<code>portamento</code>		<code>0</code>	Controls the portamento pedal. The receiver is meant to activate some kind of glide effect as long as this gate is high.
<code>sostenuto</code>		<code>0</code>	This enables the sustain pedal. This is similar to but not exactly the same as the damper pedal as it just holds notes that are pressed while the pedal goes down.
<code>soft</code>		<code>0</code>	Controls the soft pedal. The receiving synth voice is meant to play notes softer while this pedal is held down.
<code>legato</code>		<code>0</code>	Controls the legato pedal, which ties subsequent notes together.
<code>clock</code>			If you feed a steady clock here, a MIDI clock signal will be derived from this and sent through the output wire. The <i>MIDI beat clock</i> or simply <i>MIDI clock</i> is defined to send pulses at 24 PPQN: 24 pulses per quarter note. One quarter note has four 16 th s, so the MIDI clock is running at 6 pulses per 16 th note, and in the modular environment it is very common to work with 16 th pulses as a master clock. So this <code>clock</code> jack is meant to retrieve a modular master clock, multiplies this by 6 and creates a MIDI clock from it.
<code>midiclock</code>			This is an alternative to <code>clock</code> : don't use both at the same time. Here you can directly send the MIDI clock in 24 PPQN.
<code>activesensing</code>		<code>1</code>	This is a switch that disables or enables active sensing . This is a MIDI feature where a MIDI sender emits one message of the type “active sensing” every 300 ms. The receiver can use this in order to detect if we are still connected and active and also immediately reset (and turn all sound off) if these messages stop. Active sensing is enabled per default. You can disable it here by setting <code>activesensing = 0</code> .

Input	Type	Default	Description
updaterate		50.0	<p>Specifies the maximum rate at which continuous controllers like the CCs, volume, pitchbend and channelpressure are updated. This limitation is necessary in order not to flood the MIDI interface with too many updates because of just minimal changes. This rate is specified in update per second and the default is 50. A zero or negative value will completely stop all updates.</p> <p>Note: depending on how many events are happening on your channel, fewer updates might be possible. MIDI over a classical cable is limited to 3125 bytes per second. Events typically need 1, 2 or 3 bytes each.</p>
select			<p>The select input allows you to overlay buttons and LEDs with multiple functions. If you use this input, the circuit will process the buttons and LEDs just as if select has a positive gate signal (usually you will select this to 1). Otherwise it won't touch them so they can be used by another circuit. Note: even if the circuit is currently not selected, it will nevertheless work and process all its other inputs and its outputs (those that do not deal with buttons or LEDs) in a normal way.</p>
selectat	1°2°3		<p>This input makes the select input more flexible. Here you specify at which value select should select this circuit. E.g. if selectat is 0, the circuit will be active if select is <i>exactly</i> 0 instead of a positive gate signal. In some cases this is more convenient.</p>

13.35 midithrough - MIDI routing through X7

Use this circuit for forwarding MIDI data from an input to an output. Here is an example:



[midithrough]

```
fromusb = 1 # TRUE, hence USB port for input  
tousb = 0 # FALSE, hence TRS jack for output
```

This will forward MIDI events from the USB port to the TRS output. Note: All **midin** (see page 183) and **midout** (see page 190) circuits still work, so the output stream on the TRS jack will both contain the original events from MIDI-USB and the events you create with your **midout** circuits.

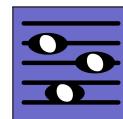
Notes:

- As of now, Sysex messages are not forwarded. Sorry for that. If that's becoming important we might add this feature.
- If you forward from USB to TRS make sure that you do not send more than 3125 bytes per second. TRS cannot output faster. It's limited by the MIDI standard. If you send MIDI data faster, some events will get lost.

Input	Type	Default	Description
fromusb	<input type="checkbox"/>	0	Set this to 0 if you want to receive data from the TRS/DIN jack and 1 if you want to receive via USB.
tousb	<input type="checkbox"/>	0	Set this to 0 if you want to send data to the TRS/DIN jack and 1 if you want to send via USB.

13.36 minifonion - Musical quantizer

This circuit is a very musical quantizer that gently moves any input CV (pitch information on a 1V/oct base) into selected notes of a musical scale. Typically the input CV is coming from a random source, LFO, melody generator or sequencer.



In fact the Minifonion is very similar to each of the three quantizer channels in the Audiophile Circuit League *Sinfonion* - just without the user interface and more flexible. It has Sinfonion compatible CVs for the root note and the scale selection so it can easily be combined with it as long as you control the Sinfonion via CV and stick to the first mode. But of course you do not need a Sinfonion in order to use this circuit!

If you want to mimick a Sinfonion with the **DROID** you might also be interested in the circuits **arpeggio** (see page 92) and **chord** (see page 115).

Here is the simplest possible application - a quantization of some (random) input pitch at **I1** to the seven notes of a C lydian major scale.

```
[minifonion]
  input = I1
  output = O2
```

Now let's change the root note to D (2 semitones above C) and the scale to natural minor, so that we now quantize to a D minor scale:

```
[minifonion]
  input = I1
  output = O2
  root = 2
  degree = 7
```

And here is the table of all 12 scales of the Minifonion. These are exactly the same scales as those in the first mode (called *Chords*) of the Sinfonion:

degree	Abbr.	Scale
0	lyd	Lydian major scale (it has a #4)
1	maj	Normal major scale (ionian)
2	X ⁷	Mixolydian (dominant seven chords)
3	sus	mixolydian with 3 rd /4 th swapped
4	alt	Altered scale
5	hm ⁵	Harmonic minor scale from the 5 th
6	dor	Dorian minor (minor with #13)
7	min	Natural minor (aeolian)
8	hm	Harmonic minor (b6 but #7)
9	phr	Phrygian minor scale (with b9)
10	dim	Diminished scale (whole/half tone)
11	aug	Augmented scale (just whole tones)

```
root = I2 * 120 # base on semitones
degree = I3 * 120 # base on semitones
```

Input	Type	Default	Description																										
input	 1V Oct	0V	Patch the unquantized input voltage here																										
trigger			This jack is optional. If you patch it, the Minifonion will work in triggered mode. Here the output pitch is always frozen until the next trigger happens.																										
bypass		off	If you set this gate input to 1 then quantization is bypassed and the input voltage is directly copied to the output.																										
noteshift	1◦2◦3	0	Shifts the output note after the quantization by this number of <i>scale</i> notes up or down (if negative). So the output note still is part of the scale but may be a note that is none of the selected ones. noteshift is applied when quantization takes places, so it also is sensible to the trigger input.																										
selectnoteshift	1◦2◦3	0	Shifts the output note after the quantization by this number of <i>selected</i> scale notes up or down (if negative). If you use noteshift at the same time, <i>first</i> selectnoteshift is applied, then noteshift . selectnoteshift is applied when quantization takes places, so it also is sensible to the trigger input.																										
root	1◦2◦3	0	Set the root note here. 0 means C, 1 means C#, 2 means D and so on. If you multiply the value of an input like I1 with 120, then you can use a 1V/Oct input for selecting the root note via a sequencer, MIDI keyboard or the like. Also then you are compatible with the ROOT CV input of the Sinfonion.																										
			<table border="1" data-bbox="819 722 932 1357"> <tr><td>0</td><td>C</td></tr> <tr><td>1</td><td>C#</td></tr> <tr><td>2</td><td>D</td></tr> <tr><td>3</td><td>D#</td></tr> <tr><td>4</td><td>E</td></tr> <tr><td>5</td><td>F</td></tr> <tr><td>6</td><td>F#</td></tr> <tr><td>7</td><td>G</td></tr> <tr><td>8</td><td>G#</td></tr> <tr><td>9</td><td>A</td></tr> <tr><td>10</td><td>A#</td></tr> <tr><td>11</td><td>B</td></tr> <tr><td>12</td><td>C</td></tr> </table>	0	C	1	C#	2	D	3	D#	4	E	5	F	6	F#	7	G	8	G#	9	A	10	A#	11	B	12	C
0	C																												
1	C#																												
2	D																												
3	D#																												
4	E																												
5	F																												
6	F#																												
7	G																												
8	G#																												
9	A																												
10	A#																												
11	B																												
12	C																												

Input	Type	Default	Description																								
degree	1 o 2 o 3	0	Set the musical scale. This is a number from 0 to 11 . At 12 this repeats over again. Please refer to the introduction for the list of scales. If you multiply an input like I1 with 120 , this will internally scale to one scale per semitone and you are compatible with the DEGREE CV input of the Sinfonion.																								
			<table border="1"> <tr><td>0</td><td>lyd - Lydian major scale (it has a $\sharp 4$)</td></tr> <tr><td>1</td><td>maj - Normal major scale (ionian)</td></tr> <tr><td>2</td><td>X7 - Mixolydian (dominant seven chords)</td></tr> <tr><td>3</td><td>sus - mixolydian with 3rd/4th swapped</td></tr> <tr><td>4</td><td>alt - Altered scale</td></tr> <tr><td>5</td><td>hm⁵ - Harmonic minor scale from the 5th</td></tr> <tr><td>6</td><td>dor - Dorian minor (minor with $\sharp 13$)</td></tr> <tr><td>7</td><td>min - Natural minor (aeolian)</td></tr> <tr><td>8</td><td>hm - Harmonic minor ($\flat 6$ but $\sharp 7$)</td></tr> <tr><td>9</td><td>phr - Phrygian minor scale (with $\flat 9$)</td></tr> <tr><td>10</td><td>dim - Diminished scale (whole/half tone)</td></tr> <tr><td>11</td><td>aug - Augmented scale (just whole tones)</td></tr> </table>	0	lyd - Lydian major scale (it has a $\sharp 4$)	1	maj - Normal major scale (ionian)	2	X7 - Mixolydian (dominant seven chords)	3	sus - mixolydian with 3 rd /4 th swapped	4	alt - Altered scale	5	hm ⁵ - Harmonic minor scale from the 5 th	6	dor - Dorian minor (minor with $\sharp 13$)	7	min - Natural minor (aeolian)	8	hm - Harmonic minor ($\flat 6$ but $\sharp 7$)	9	phr - Phrygian minor scale (with $\flat 9$)	10	dim - Diminished scale (whole/half tone)	11	aug - Augmented scale (just whole tones)
0	lyd - Lydian major scale (it has a $\sharp 4$)																										
1	maj - Normal major scale (ionian)																										
2	X7 - Mixolydian (dominant seven chords)																										
3	sus - mixolydian with 3 rd /4 th swapped																										
4	alt - Altered scale																										
5	hm ⁵ - Harmonic minor scale from the 5 th																										
6	dor - Dorian minor (minor with $\sharp 13$)																										
7	min - Natural minor (aeolian)																										
8	hm - Harmonic minor ($\flat 6$ but $\sharp 7$)																										
9	phr - Phrygian minor scale (with $\flat 9$)																										
10	dim - Diminished scale (whole/half tone)																										
11	aug - Augmented scale (just whole tones)																										
select1			<p>Gate input for selecting the <i>root note</i> as being an allowed interval. When you want to create a playing interface for live operation you can patch the output of a toggle button (made with the circuit [button]) here.</p> <p>Note: When all select and selectfill inputs are 0, automatically all seven scale notes are selected, i.e. select1 ... select13 will be set to one.</p>																								
select3			Gate input for selecting the 3 rd .																								
select5			Gate input for selecting the 5 th .																								
select7			Gate input for selecting the 7 th .																								
select9			Gate input for selecting the 9 th (which is the same as the 2 nd).																								
select11			Gate input for selecting the 11 th (which is the same as the 4 th).																								
select13			Gate input for selecting the 13 th (which is the same as the 6 th).																								

Input	Type	Default	Description
selectfill1		off	Selects the alternative 9 th (i.e. the 9 th that is <i>not</i> in the scale).
selectfill2		off	Selects the alternative 3 rd (i.e. the 3 rd that is <i>not</i> in the scale).
selectfill3		off	Selects the alternative 4 th or 5 th . In most cases this is the diminished 5 th .
selectfill4		off	Selects the alternative 13 th (i.e. the 1 st 3 that is <i>not</i> in the scale).
selectfill5		off	Selects the alternative 7 th (i.e. the 7 th that is <i>not</i> in the scale).
tuningmode		off	While this is 1 , the circuit will output the value set by tuningpitch instead of the actual pitch. This is meant to be a help for tuning your VCOs.
tuningpitch	$\frac{1V}{Oct}$	0V	This pitch CV will be output while the tuning mode is active.
transpose	$\frac{1V}{Oct}$	0V	This value is being added to the output pitch when not in tuning mode. It can be used for musical transposition or adding a vibrato.

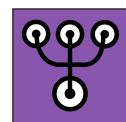
Output	Type	Description
output	$\frac{1V}{Oct}$	Here comes your quantized output voltage
notechange		Whenever the quantization changes to a new note a trigger with the duration 10 ms is output here. No trigger is output in bypass mode.

13.37 mixer - CV mixer

The main task of this circuit is simply adding up to eight inputs. Furthermore it can do simple operations like minimum, maximum and average. Please note that since every input can always be offset and attenuated, it's like a mixer with a CV controlled level and CV controlled offset per input channel.

Minimal example, mixing together two inputs:

`[mixer]`



```
input1 = I1
input2 = I2
output = O1
```

Since every input can add an offset, mixing four inputs can be done with two lines if you like:

```
[mixer]
input1 = I1 + I2
input2 = I3 + I4
output = O1
```

Please note that an unpatched input is (sometimes) not the same as an input where 0.0 is being sent. The difference arises if you use `minimum`, `maximum` and `average`, since these just consider the patched inputs.

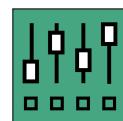
If eight inputs are not enough then you can simply create a mesh by mixing together the outputs of several submixers.

Input	Type	Default	Description
<code>input1 ... input8</code>		0.0	1 st ... 8 th mixing input

Output	Type	Description
<code>output</code>		Sum of all patched inputs
<code>maximum</code>		Maximum of all patched inputs of this circuit. This can e.g. be used for mixing together the envelopes from several sequencer tracks without making them "louder" or distorting them when two sequencers play a note at the same time.
<code>minimum</code>		Minimum of all patched inputs of this circuit.
<code>average</code>		Average of all patched inputs of this circuit.

13.38 motoquencer - Motor fader sequencer

This circuit allows you to build simple but also very complex performance sequencers based on motorized faders. It supports up to 32 steps and up to eight M4 controllers with up to 32 faders. The list of features is long and diverse and aims at supporting creative live performances.



You probably will fail to map all existing inputs to controls, so better don't try and rather experiment with just a fraction of those at a time.

Basic minimal example

Despite all the features, this sequencer is easy to get started with. Here is the smallest possible example. You always need a clock input. Here I get it from input **I1**. You need to have at least one M4 unit attached to your DROID (and declared with **[m4]** in your patch). The motor sequencer automatically configures all your available faders (up to 32) for the sequencer (you can change that with **firstfader** and **numfaders**):

```
[m4]
```

```
[motoquencer]
  clock = I1
  cv = 01
  gate = 02
```

As soon as your clock starts, you get a sequence with one step per available fader (which is four if you have just one **[m4]** declared). The faders select notes from a C lydian scale in two octaves. You will feel 15 notches. They correspond to the 15 notes in this range. The touch buttons below the faders switch on/off the gates.

The pitch is output at **01** and the gate at **02**. Well - this wouldn't have needed expensive motor faders, but it works and shows a minimal application of **motoquencer**.

Switching pages

Your sequence can have more steps than you have faders. This is done by switching *pages*. In the following example we assume that you have just one M4 but want a sequencer with 16 steps. Use the **page** input in order to set the current page (group of 4 steps) that you want to see and edit with your faders. These pages have the numbers **0**, **1**, **2** and **3**. That number can nicely be output by a **buttongroup** (see page 108) on a P2B8. Here is a fully functional example of a 16 step sequencer with just four faders:

```
[p2b8]
[m4]

[buttongroup]
  button1 = B1.1
  button2 = B1.2
  button3 = B1.3
  button4 = B1.4
  led1 = L1.1
  led2 = L1.2
  led3 = L1.3
  led4 = L1.4
  output = _PAGE
```

```
[lfo]
  hz = 20 * P1.1
  square = _CLOCK

[motoquencer]
  clock = _CLOCK
```

```
page = _PAGE
numsteps = 16
cv = 01
gate = 05
```

Repeats, Ratchets and Randomize

In the upper examples we just had two parameters per step of the sequence: The pitch / CV and the gate. There are some more. Altogether every step has the following eight parameters:

0	pitch / CV
1	randomize CV
2	gate propability
3	repeats (up to 16)
4	gate pattern
5	ratchets (up to 8)
6	gate
7	skip

Each of these parameters has a number from **0** to **7** and you can set the input **fadermode** to one of these in order to switch the faders to control that parameter. Here are some details about the various parameters:

Pitch / CV is the output pitch of each step. With the inputs **cvgbase** and **cvgrange** you can define a voltage range for those CVs. Per default, the CV is quantized to a musical scale, but you can change that with **quantize** (see below).

Randomize CV is a number from 0 (fader at the bottom) to 7 (fader at the top). 0 means randomization is off. The other 7 steps will increasingly modify the step's CV by adding a different random offset each time the step is played. At position 7 (the maximum), the offset is up to **cvrage**, so if your CV is at maximum, this could double up your CV range.

Gate probability also has 8 settings. Here the maximum (fader at top position) is the default and means: this step is always played, if the gate is on. The other seven settings will reduce the probability of this step being played. The lowest setting still leaves a small chance. Turn off the gate to silence a step completely.

But this probability is not simply a random chance. It has several very musical settings as you can see from the following table. Here you see the eight fader positions and their meaning - 8 being the top position and 1 the bottom position:

Pos.	Meaning	
8 (top)	played always	100%
7	random chance of 50%	50%
6	played every <i>even</i> turn	50%
5	played every <i>odd</i> turn	50%
4	random chance of 25%	25%
3	played every 4 th turn	25%
2	random chance of 12%	12%
1	played if last random was positive	-

The LEDs below the faders indicate the current setting with different color and blink codes:

- Gates that are played always are blue with a con-

stant light.

- Random gates for 50%, 25% and 12% are in the same blue but blink in various speeds.
- Gates of setting 1 (conditional random) are blinking fast.
- Gates depending on the turn (3, 5 and 6) are in cyan color and light *steadily* in the bars (turns) where they are *on* and blink in the other bars.

The position 6 and 5 are very musical and can transform a pattern of length 8 into an effective melody of 16 steps. A step in position 6 is just played every second run of the whole sequence. Position 5 is just the same but starts with the first run and will then be played on run 3, 5, and so on.

Position 4 is similar, but these steps will just be played every fourth sequence run, so you can use it for playing things like a pickup or break or the like. These "run counters" are reset by the **reset** input.

The bottom position of 1 is an addition for the true random positions 7, 4 and 2: A step in position 1 is played, whenever the *most recent* random decision of positions 7, 4 and 2 was *positive*. It allows you to create groups of notes that are either played completely or not at all: Set the first step of these to a random probability of 50, 25 or 12%. And the remaining notes to position 1. Now whenever fate decides that the first note is being played, so will all remaining ones. These steps do not need to be subsequent. You can have wholes.

Repeats changes the number of clock cycles one step will last. It is a number from 1 (fader at the bottom) to 16 (fader at the top). This setting changes the total duration of one sequence cycle. If you set repeats to 2 for one of 16 steps, your sequence will last 17 clock cycles.

The **Gate pattern** decides how gates are played when **repeats** is 2 or larger. There are four gate patterns, which

you can feel in the fader. In the first setting (fader down) just the first repetition of the step is "played" (i.e. a gate signal sent). Setting 2 will play one gate per repetition. Setting 3 plays one long gate. And setting 4 is like 3 but lets the gate open when the step ends. This ties this step to the next one. And this setting also has an effect when **repeats** is just 1.

Ratches can be set from 1 (normal) to 8. It divides the clock cycle of the step into equal time intervals in which the step is repeated. If you set ratchets to 2, for example, you will get two notes played at double time. Ratchets do *not* change the duration of the sequence.

The remaining two settings are usually set with the touch buttons, but you can also use the faders.

Gate decides whether the step is "played". If it is played, its CV will be sent to the **cv** output and the **gate** signal is set to high for half a clock cycle (you can change all this, no worries).

Steps with **Skip** enabled will be skipped. This shortens the duration of the sequence. Note: if *all* steps are set to skip, the sequencer repeats playing the most recent step over and over.

So let's now make an example where we use a button group for setting **fadermode**:

[p2b8]

[m4]

[buttongroup]

```
button1 = B1.1
button2 = B1.2
button3 = B1.3
button4 = B1.4
button5 = B1.5
button6 = B1.6
button7 = B1.7
```

```

button8 = B1.8
led1 = L1.1
led2 = L1.2
led3 = L1.3
led4 = L1.4
led5 = L1.5
led6 = L1.6
led7 = L1.7
led8 = L1.8
output = _FADERMODE

[lfo]
hz = 20 * P1.1
square = _CLOCK

[motoquencer]
clock = _CLOCK
fadermode = _FADERMODE
cv = 01
gate = 05

```

Button mode

Very similar to the faders, also the touch buttons have modes. These can be switched with **buttonmode** and here are the possible settings:

0	gates
1	start / end
2	gate pattern
3	skip

Three of these settings you already know from the **fadermode**. When the buttons are set to **gate pattern**, you cycle through the four steps each time you touch the button (and the LED cycles through four colors).

Fun fact: You can set **fadermode = 6** and **buttonmode =**

0. That way, both the button and the fader control the gates. Try this out and touch the buttons: the fader will move automatically.

The mode “start / end” cannot be set with the faders. They set a sub range of the sequence to be played. Here is what it means:

Start and end

Usually your sequence is played from the first to the last step. But you can change this by setting a start step and an end step. This can either be done manually (with **buttonmode = 1** or with the inputs **startstep** and **endstep**).

In **buttonmode = 1**, the start step has a green LED and the end step a red one. Both start and end can be at the same step (creating a one step sequence). The LED will then blink between red and green.

Touching a button changes the **end** step. You can set the start step by first setting an end step and **holding** that button and then - with a second finger - press another step. That will be the start step.

If the start step is after the end step, the play order is reversed.

Quantization, root and scale

Per default, the CVs are quantized to the notes of a lydian C major scale, as is the default for many other circuits, as well. This means that the faders have one artificial notch for each scale note. You can *feel* the notes. This makes it easy to change the note in exact steps without any display.

As with many other pitch-aware circuits, like for example **minifonion** (see page 200) or **chords** (see page ??), you can use **root** and **degree** for changing the scale. See in the table of inputs below for the different possible scales. Note: **root** has no effect on the lower CV boundary. It's just for the selection of the allowed notes. Use **cibase** for setting that.

Furthermore, there are the inputs **select1**, **select3**, ... You can use them to further restrict the possible notes - or even add notes that are not contained in the scale. Refer to the **minifonion** (see page 200) circuit for a broader discussion of these inputs.

Note: If you have set a melody with the faders and reduce the number of allowed notes afterwards, the faders will possibly move to new positions. But as long as you don't touch them, they will internally “remember” their original note. If you later re-add the missing notes, the faders will move back and your original melody is restored.

With the input **quantize** you can switch off the musical mode. **quantize = 0** disables quantization and the faders create a continuous CV (the internal resolution is 127 steps, just like in a MIDI CC). And **quantize = 1** will quantize to semitones ($\frac{1}{12}$ V steps).

Note: The maximum number of notches is 201. But if you select more than 25 notches, the force feedback is turned off as the notches would get too small to work. This number of 25 “real” notches nicely matches the 25 possible semitones of two octaves. If you increase that range, the notches are switched off.

Direction, ping pong, movement patterns

The Motoquencer has quite a bunch of interesting features for changing the order in which steps are being played. Some of them, like the playing direction or “ping

pong", are the usual suspects and common among sequencers. The "playing patterns" and "forms" go beyond this and create interesting creative possibilities.

direction defaults to **0**, which means "forwards". Set this to **1** (e.g. with a toggle button) to run the sequence backwards.

```
direction = 1 # backwards
```

pingpong is another switch. Setting it to **1** enables "ping pong mode". Here the direction switches back and forth. Depending on **direction**, the sequence starts at the start step or the end step, moves towards the other end and then turns around in order to come back. Note: Since the steps at the turning points are played just once, a sequence of 8 steps in ping pong mode has a duration of 14, not 16.

```
pingpong = 1 # enable ping pong
```

pattern changes the way how the sequencer steps through the sequence. Pattern **1** for example goes always two steps forwards (according to **direction** and **pingpong**) and then one step backwards. Assuming **direction = 0** and **pingpong = 0**, the step order would be 1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 5, 6 and so on. The available patterns are much the same as in the **arpeggio** (see page 92) circuit with the addition of pattern **6**, which goes forwards in small random steps.

```
pattern = 3 # set pattern 3
```

Forms like AAAB

Already confused? Then you probably won't like the "Forms" feature! Here we create longer sequences by first

dividing the steps into two (or three parts), and then playing these parts in certain orders.

The most useful form (except the trivial **0**) is probably **1**, which is AAAB. Here the steps are divided into a first half, which is called A, and a second half, which is called B. The A part is always played thrice and then once the B part. Assuming you have 8 steps (and all the other fancy stuff is off), the step order would be 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 5, 6, 7, 8.

The patterns with the three parts A, B, and C divide the steps into three equal sized parts. You better make sure that you have 6 or 12 or 24 steps in that case, or else your parts won't have equal size (which on the other hand could be funny anyway).

The forms can be combined with **direction**, **pingpong** and **pattern**. Here stepping modifications are always applied *within each individual part*.

The forms can also be combined with the start and end point. Here just the steps between start and end are divided into parts.

Autoreset

In contrast to the all the upper modifications of the step order, **autoreset** is super simple. It resets the whole sequence (including parts) to the very beginning after a specified number of clock ticks.

There are two typical applications: First, if you want to make sure that the pattern repeats in some regular way despite crazy modifications, set **autoreset = 16** and the sequence will restart exactly every 16th clock tick. If it is longer, it will be truncated. If it is shorter, it first repeats, but then the repetition is truncated.

On the other hand you can make a regular sequence irregular, if you set e.g. **autoreset = 7** in a sequence with usually 16 steps, thus forcing polymetric shifts with other parallel rhythms.

When you use the special gate "probabilities" odd and even in combination with autoreset, please note that after a reset the odd / even count always starts with odd.

The Metric Saver

The Metric Saver™ is a very musical feature that allows you to go bonkers with all start, end, direction, ping pong, pattern, form, repeats, autoreset and skips without losing the sync to the rest of your music.

If *The Metric Saver™* is turned on (which is the default), the **motoquencer** automatically keeps track of the original incoming clock count. As soon as – after a polymetric journey – you come back to "normal", it jumps to the step that *would* have been the current one without those alterations.

An example: You set **autoreset** to 7 in order to create polymetric tension. Later you set it back to **0**. Now the sequence immediately jumps to the step where it would have been without **autoreset** (this requires that none of the other step changing features are in use). You snap back to your original groove and are in sync again with the rest of your modular "band".

Note: *The Metric Saver™* is only activated when really *all* modifications to the normal step order are turned off. That also includes steps where "repeats" or "skip" is used, since they also introduce time shifts.

I Feel Lucky

The Motoquencer has a powerful system of *one time randomization*, which is called *I Feel Lucky™*. While setting random CVs or gate probabilities is quite common amongst sequencers, here we talk of something different. By sending a trigger to a certain input, some of your steps are randomly modified - and stay that way. If your faders currently show these steps, you will immediately see them moving around. And they stay there, so that you can manually modify the random decision if you like. Those triggers are most times sent by buttons, but also slowly running LFOs or using the **startofsequence** as a trigger are fine.

Let's make a simplified example:

```
[motoquencer]
... usual stuff goes here ...
luckychance = P1.1
luckyamount = P1.2
luckycvs = B1.1 # press to reroll CVs
```

All **lucky** operations honor the **luckychance** input. This sets the relative number of steps that is affected by the randomization. Setting it to **1** will affect all steps. At **0**, no step is affected. At **0.5** exactly half of the steps is affected, randomly chosen from all steps *between start and end*.

A trigger to **luckycvs** sets a new random CV value for each affected step. And with the pot **luckyamount** you control the maximum CV that's possible here.

You can use this mechanism also to reset things. A trigger at **luckycvs** with **luckyamount = 0** and **luckychance = 1** will bring all steps back to the CV set by **cvgbase**.

Please have a look at the table of inputs for all the other **lucky...** triggers and ... *feel lucky!*

Multiple tracks

Each **motoquencer** circuit has just one CV and one gate output. In many cases it is desirable to have several CVs and maybe also additional gate outputs as part of a sequence. Also you probably want more sequencers using the same faders, of course.

This is done by adding more instances of **motoquencer** to your patch. The easiest way is to use the **select** input of each of these, in order to make sure that at every time exactly **one motoquencer** is selected and gets access to the motor faders. You really shouldn't try selecting more than one at the same time, or your faders will get crazy!

Here is an example with the two buttons **B1.7** and **B1.8** selecting one of two sequencers:

```
[p2b8]
[m4]

[buttongroup]
button1 = B1.7
button2 = B1.8
led1 = L1.7
led2 = L1.8

[lfo]
hz = 20 * P1.1
square = _CLOCK

[motoquencer]
clock = _CLOCK
select = L1.7
cv = 01
gate = 05

[motoquencer]
clock = _CLOCK
select = L1.8
cv = 02
```

gate = 06

This simple patch is a fully functional two-track four-step sequencer. And as long as you don't run out of RAM, you can add as many tracks as you like.

One thing you have to have in mind: These sequencers can easily go out of sync. Just play around with the start or end step or skip or repeats. While that *can* be interesting, sometimes it is not desirable. Maybe you just want every step to have additional CV or gate values.

This can be done by **linking** two or more instances of **motoquencer** together. To do that, add the following line to the first instance:

```
linktonext = 1
```

At the next **motoquencer** in the patch, don't wire **clock** or **reset** or anything else that deals with stepping or direction or faders. Just connect the outputs. The linked sequencer is *remote controlled*.

Some inputs still apply for the linked sequencer. One example is **cvgbase** and **cvgrange**. Any parameter that has an influence on which step is played when, however, is ignored. That task is done by the main sequencer.

Here is a complete example that adds one additional CV and one gate to a sequencer. Note: The fader modes 10 and 16 give you access to the modes 0 and 6 of the linked sequencer. Simply add 10 for each sequencer in the chain.

```
[p2b8]
[m4]

[buttongroup]
button1 = B1.1
button2 = B1.2
```

```

button3 = B1.3
button4 = B1.4
button5 = B1.5
button6 = B1.6
button7 = B1.7
button8 = B1.8
led1 = L1.1
led2 = L1.2
led3 = L1.3
led4 = L1.4
led5 = L1.5
led6 = L1.6
led7 = L1.7
led8 = L1.8
output = _FADERMODE
value7 = 10 # CV of sequencer 2
value8 = 16 # gate of sequencer 2

```

```

[lfo]
hz = 20 * P1.1
square = _CLOCK

```

```

[motoquencer]
clock = _CLOCK
fademode = _FADERMODE
linktonext = 1
cv = 01
gate = 05

```

```

[motoquencer]
cv = 02
gate = 06

```

If you need more than two CVs, you can create even longer chains, for example:

```

[motoquencer]
clock = _CLOCK
fademode = _FADERMODE
linktonext = 1
cv = 01

```

```

gate = 05

[motoquencer]
linktonext = 1
cv = 02
gate = 06

[motoquencer]
cv = 03
gate = 07

```

Simply add a **linktonext** at every instance except the last. And add 10 to **fadermode** for every sequencer. For example **fadermode = 25** selects fader mode 5 on the third sequencer in the chain.

Here are some details, what linking exactly means for the linked sequencer:

- The linked sequencer does not react to **clock**, **reset**, **startstep**, **endstep**, **form**, **direction**, **pingpong**, **pattern**, **autoreset**, **shiftsteps** or any other potential means of influencing the play order of the steps. Instead the current step number of the linked sequencer will always be the same as the step number of the main sequencer.
- If you use **shiftsteps**, **luckyshuffle** or **luckyreverse** on the main sequencer, the exact same rearrangement of steps will happen at the linked sequencers.
- If the main sequencer plays repeats, so does the linked one. The “repeats” setting of the linked sequencer’s steps are ignored.
- If the main sequencer skips a step, so does the linked one. The “skip” property of steps in the linked sequencer are ignored, as well.
- Ratches still work independently, since they don’t change the step sequence.
- Also the gate pattern of the linked sequencer will be applied.

- In the linked sequencer, **holdcv** has one additional value: 2. If you set it to 2, the CV output of the linked sequencer is synchronized to the gate of the linked sequencer, not to that of the main sequencer.
- Don’t use **select**, **fadermode** and **buttonmode** on the linked sequencer. They are ignored. Instead, for accessing the parameters of the steps of the linked sequencer, add 10 to **fadermode** or **buttonmode**. So while **fadermode = 1** sets the fader to the CV randomization of the main sequencer, so does **fadermode = 11** for the linked sequencer.

The following parameters are still valid for the linked sequencer:

- cvbase**, **cvrage** and **quantize**
- gatelen**
- holdcv** (with the extra value 2)
- luckychance**, **luckyamount** and all of the other **lucky...** paramters, with the exception of **luckyskips**, **luckyrepeats**, **luckyshuffle** and **luckyreverse**.

Recording with a keyboard

You can use a keyboard to record sequences into your motoquencer. More precisely, you can attach a CV / gate input for that purpose. That *might* very well come from a keyboard attached to the X7, via the circuit **midinin** (see page 183). But any other source is possible, as well.

The first step is attaching your recording source to **keyboar**dcv and **keyboar**dgate. Here is an example:

```

[midinin]
cv = _CV

```

```

gate = _GATE

[motoquencer]
  keyboardcv = _CV
  keyboardgate = _GATE
  ...

```

After doing this, you should already be able to play with your keyboard directly to the voice that's attached to the motoquencer. While a key is pressed (**keyboardgate** is high), the **keyboardcv** has precedence over the sequence. But you can change that with the setting **keyboardmode**.

To record your keyboard into a sequence, you need to connect **recordmode**, maybe to a **button** (see page 103). While recording is active and the keyboard gate is high, the current sequencer step will be replaced with your keyboard note. Otherwise the steps are untouched. That way you play more and more notes into the sequence.

In order to get rid of existing notes, either clear the sequence before recording (using the **clear** trigger), or make use of the input **recordsilence**. Setting that to 1 will silence all steps when no key is pressed.

You also can route **recordsilence** to one key on your keyboard using the **notegate** outputs of **midiin**. That way you can actively "erase" notes by pressing that key.

While recording key presses the motoquencer tries to be tolerant with respect to your timing. So keys pressed slightly before or after the current clock tick are just fine.

Note: The sequencer can just record into its grid of steps and quantized notes. So it's not a free style MIDI recorder. You cannot record notes that are faster than your input clock. If you have enabled quantization, you can just play notes from the current scale. So it needs some time to get familiar with this way of recording. Nev-

ertheless it's a great tool for rapid composition. Especially because it's easy to modify your melodies with the faders after you have recorded them.

Recording & linked sequencers

When you have combined several **motoquencers** with **linktonext = 1**, recording also works in the linked sequencers. Here are some hints:

- **recordmode** can be (and must be) set individually on each of the motoquencers. If you want to record into a linked sequencer, make sure that you set **recordmode** there.
- Using the same value for **recordmode** for all sequencers means that they always record simultaneously.
- Also **keyboardcv** and **keyboardgate** are settings that each sequencer instance has on its own. That means that you can record different CVs with different gates on each sequencer at the same time.
- Using the same gate signal for the **keyboardgate** of all sequencers can make sense. E.g. if you want to record paraphonic chords or pitches together with modulation CVs.

Copy & paste

The copy & paste feature allows you to copy a part of your sequence from one page to another or from one preset to another. To do this, map the inputs **copy** and **paste** to two buttons (you don't need toggle buttons here, so no **button** circuit is needed).

A trigger to **copy** copies the current sequence into an internal clipboard. And **paste** copies the clipboard into the current sequence.

Use **copymode** to determine whether just the current page or the complete sequence shall be copied.

There are also two alternative triggers for pasting. **pastefaders** just pastes the faders of the currently selected mode. **pastebuttons** is likewise for the buttons. With that you can for example just copy the gate probabilities from one page to another while leaving the rest of the parameters as they are.

If you have linked sequencers, those will automatically be handled as well. Don't connect the **copy** and **paste** triggers there.

LED colors

Depending on the **buttonmode**, the LEDs below the faders have different colors. Here is an overview over all possible colors:

color	meaning	buttonmode
white	currently played step	always
blue	enabled gate	0
green	start step	1
red	end step	1
cyan	gate on the first repetition	2
pink	gate on each repetition	2
orange	hold gate over duration	2
yellow	tie the gate to the next step	2
violet	skip	3

Input	Type	Default	Description
firstfader	1◦2◦3	1	First M4 fader of the sequencer (starting with 1). If you omit this, it starts at the first fader of your first M4.
numfaders	1◦2◦3		Number of faders to use for your sequencer. The typical numbers are 4, 8, 16 and 32. 32 is the maximum (eight M4 units). If you omit this, all of your M4 faders will be used.
numsteps	1◦2◦3		<p>Number of steps your sequence consists of (at maximum). The number of steps can be greater than the number of faders. In that case use page for paging your faders so that you can edit all of the steps. Having the number of steps less than the faders, makes no sense - it's just a waste of faders. The maximum number of steps is 32.</p> <p>If you don't set this parameter, the number of steps will be set to the number of faders.</p> <p>Note: changing this setting dynamically can provoke various surprising behaviours. For example the number of pages (see parameter page) might be reduced. Or the end marker is forcibly moved around. If you want to change the length of the sequence via CV, better use endstep or autoreset.</p> <p>Another note: Setting numsteps will <i>not</i> restrict the number of faders. If you set numsteps = 4 but have eight faders available, the circuit will use all these, even if faders 5, 6, 7 and 8 will be useless. You need to set numfaders = 4 in this situation.</p>
page	1◦2◦3	0	Use this parameter, if you have less faders than steps. The first page is 0 , not 1 . For example if you have 4 faders but 16 steps, you can select between the four "pages" of four faders each, by settings bar to 0, 1, 2 or 3 . The output of a buttongroup (see page 108) with one button per page is a good match for this parameter.
clock			Patch an input clock here. If you want to use ratcheting, that clock needs to be stable and regular, because the sequencer needs to interpolate the clock and create evenly distributed new beats within two clock ticks. If you don't use ratching, you can use any rhythm you like here - may it be shuffled, euklidean, the output from another sequencer or whatever you like. Each clock tick will advance the sequence to the next step (or to the next repetition of the current step).
reset			A trigger here resets the sequencer to its start step. The next clock tick (or a tick that is roughly at the same time as the reset) will play step 1. Note: If there is a reset without a clock tick at the same time, the sequencer will go to "step 0", which is a special state where it waits for the clock to advance to the first step. Without that fancy logic, a reset plus clock would skip step 1 and start with step 2.

Input	Type	Default	Description
run		1	If you set this input to 0, the sequencer will ignore all incoming clock ticks. It stops. The default of 1 is normal operation, where it runs. This input is a better way to temporarily stop the sequencer than to stop the clock. The reason: for computing the gate length and ratchets a steady input clock is needed. If you stop the clock, the next gate length and ratchets right after the next start will have the wrong duration since at least two clock ticks are necessary for computing its speed. Note: This input is not a replacement for mute , since a muted sequencer leaves the clock running and advances steps. It just mutes the gate output.
composemode			Enabling “compose mode” makes it easier to find the right note in a step, when creating more complex melodies. When composemode is set to 1, the sequencer stops clocking. Instead - every time you change the CV of a step, it immediately jumps to that step, outputs the changed CV and opens the gate for a short time, so you can listen to the changed note.
mute			If you set this to 1, the gate output of the sequencer is muted (will always be 0). Any changes of the CV output still happen.
cibase		0.0	Lowest CV voltage the sequencer will output.
cvrage		0.2	CV range of the faders. So the resulting CV lies somewhere between cibase and cibase + cvrange .
quantize	1 ◊ 2 ◊ 3	2	Switches on quantization in two levels. At 0, the faders run freely and output a continuous CV. At 1, the output is quantized to semitones, which is $\frac{1}{12}$ V steps. Also the faders will get artificial notches - one for each semitone. That is, unless your range is too large. The maximum number of notches with force feedback is 25, so if your range exceeds two octaves (0.2), the notches are turned off. At 2, the output is quantized to the scale that root and degree define. Furthermore the individual scale notes can be switched on or off with the parameters select1 , select3 and so on. Note: the root input does not select the lowest note of the CV range. That is still set with cibase . It is just used for selecting the scale.

0	no quantization
1	quantize to semitones (1/12V steps)
2	quantize to the scale set by root and degree

Input	Type	Default	Description
cvnotches	1◦2◦3	0	<p>Usually the CVs of the steps are meant to be note pitches (when quantize is 1 or 2), or just free CVs (quantize = 0). There is an alternative mode, however, that allows you to assign integer values like 0, 1, 2 and so on to each step.</p> <p>To do this set cvnotches to a value of 2 or greater. This defines the number of discrete values (and hence notches) for each step and put CVs of the sequence into <i>notched mode</i>. 1 makes no sense, of course, since in this “pitch bend mode” the faders would always return to the neutral position.</p> <p>In notched mode the cv output does not output a pitch but a notch number starting from 0. cibase, cvrage and quantize are ignored.</p> <p>The maximum number of notches is 127, but the haptic force feedback of the motor faders is disabled starting at 26.</p>
shiftsteps	1◦2◦3	0	<p>Shifts all your steps by that number to the left (negative numbers shift to the right). So if shiftsteps is 1, right after a reset, the sequencer will not play step 1, but step 2. The shifting wraps around at the end of your sequence, so if you have 24 steps and shift is 1, the sequencer will play step 1 instead of step 24.</p> <p>Note: Other things like startstep, endstep, playmode, from and autoreset take place after shifting.</p>
startstep	1◦2◦3	1	<p>Sets the first step to be used. This means that after a reset or when the sequencer comes to the end of the sequence, it will begin at this step.</p> <p>There is also a way for settings start and end with buttons (see below at buttonmode). If you use the interactive mode, the startstep and endstep settings will be overridden. They are reactivated if you clear everything.</p> <p>Note: startstep and endstep take place after applying shiftsteps.</p>
endstep	1◦2◦3		<p>Sets the last of the steps to be played. The default is to play all steps. After playing the end step, the sequencer moves on to the start step at the next clock tick.</p> <p>If startstep is equal to endstep, the sequence just consists of one single step.</p> <p>Settings startstep larger than endstep is allowed and reverses the playing order.</p>

Input	Type	Default	Description														
form	1◦2◦3	0	<p>This is an advanced feature that allows you to slice your steps into two or three parts and create musical song forms like AAAB or ABAC. Each of the parts A, B or C are then played according to the playmode.</p> <p>The form AAAB, for example, creates a 32 step form from just 16 steps, by playing the first 8 steps three times and then the second 8 steps once.</p> <p>The following forms are available:</p> <table border="1"> <tr><td>0</td><td>A (forms are basically deactivated)</td></tr> <tr><td>1</td><td>AAAB</td></tr> <tr><td>2</td><td>AABB</td></tr> <tr><td>3</td><td>ABAC</td></tr> <tr><td>4</td><td>AAABAAAC</td></tr> <tr><td>5</td><td>AB</td></tr> <tr><td>6</td><td>AAB</td></tr> </table> <p>Notes:</p> <ul style="list-style-type: none"> The splitting of the steps into parts takes place <i>after</i> accounting for startstep and endstep. Forms with A, B and C split the pattern into three parts. These parts can only be of equal size if the number of steps is dividable by 3, of course. The pattern AB is really not the same as A, e.g when direction is set 1 (reverse). In that case each of the parts is played backwards, but the parts themselves move forwards on your steps. 	0	A (forms are basically deactivated)	1	AAAB	2	AABB	3	ABAC	4	AAABAAAC	5	AB	6	AAB
0	A (forms are basically deactivated)																
1	AAAB																
2	AABB																
3	ABAC																
4	AAABAAAC																
5	AB																
6	AAB																
direction		0	Sets the general direction in which the sequencer moves through the steps. 0 means forwards and 1 means backwards.														
pingpong		0	If set to 1, the sequencer will change the direction every time it reaches the start or end of the sequence.														

Input	Type	Default	Description
pattern	1◦2◦3	0	Selects one of a list of movement patterns. That way, the sequence steps are not played in linear order but in a more sophisticated movement. Available pattern are:
		0	go step by step to the sequence (normal)
		1	two steps forward, one step backward
		2	double step forward, one step backward
		3	double step forward, double step backward, single step forward
		4	double step forward, single step forward, double step backward, single step forward
		5	random single step forward or backward
		6	go forward by a small random number of steps
		7	random jump to any allowed (other) note
autoreset	1◦2◦3	0	If set to non-zero, automatically issues a reset (just like a trigger to reset) every N clock ticks.
metricsaver		1	<p><i>The Metric Saver</i>™ helps you to reliably come back to your original metric and time after playing around with all sorts of parameters that change the played number of steps in the sequence. These are: startstep, endstep (also when changed interactively), form, direction, pingpong, pattern, autoreset and repeats and skips of individual steps. Therefore it counts the actual number of clock cycles since the last external reset (or system start). And when all of these features are deactivated, it snaps back the clock to the position it <i>would</i> have been by now if you never had played around with all the funny stuff.</p> <p>That way, during a live performance, you can safely play around with all this polymetric and otherwise time disrupting stuff and as soon as you clean up your mess - voila: you are back on track and in sync with the rest of the "band".</p> <p>The metric saver is turned on by default. But you can disable it by setting the parameter to 0.</p>

Input	Type	Default	Description																
fadermode	1◦2◦3	0	Switches the current meaning of the motor faders. You probably want to connect the output of a buttongroup (see page 108) here. Here are the possible modes:																
			<table border="1"> <tr><td>0</td><td>pitch / CV</td></tr> <tr><td>1</td><td>randomize CV</td></tr> <tr><td>2</td><td>gate probability</td></tr> <tr><td>3</td><td>repeats (up to 16)</td></tr> <tr><td>4</td><td>gate pattern</td></tr> <tr><td>5</td><td>ratchets (up to 8)</td></tr> <tr><td>6</td><td>gate</td></tr> <tr><td>7</td><td>skip</td></tr> </table>	0	pitch / CV	1	randomize CV	2	gate probability	3	repeats (up to 16)	4	gate pattern	5	ratchets (up to 8)	6	gate	7	skip
0	pitch / CV																		
1	randomize CV																		
2	gate probability																		
3	repeats (up to 16)																		
4	gate pattern																		
5	ratchets (up to 8)																		
6	gate																		
7	skip																		
buttonmode	1◦2◦3	0	Switches the current meaning of the touch buttons below the faders. You probably want to connect the output of a buttongroup (see page 108) here. Here are the possible modes:																
			<table border="1"> <tr><td>0</td><td>gates</td></tr> <tr><td>1</td><td>start / end</td></tr> <tr><td>2</td><td>gate pattern</td></tr> <tr><td>3</td><td>skip</td></tr> </table>	0	gates	1	start / end	2	gate pattern	3	skip								
0	gates																		
1	start / end																		
2	gate pattern																		
3	skip																		
holdcv		1	<p>This setting determines whether the CV output changes every time the sequencer moves to the next step or just when that step is active (a gate is being played). The latter is the default. But if you set this to 0, the CV values of steps without gates will also influence the output CV.</p> <p>Note: regardless of this setting, the CV will never change inbetween. Any change of the CV faders, the cibase and cvrage and so on will only take effect when the next step is played. This also ensures that repeats or ratchets are always in the same pitch.</p>																
defaultcv		0.0	<p>Set the CV the steps should be set to on a trigger to clear. That value must be within the range set by cibase and cvrage, or it will be truncated to that range.</p> <p>If you have set cnotches, however, the value is expected to be an integer in the range 0 ... cnotches - 1.</p>																

Input	Type	Default	Description						
defaultgate		1	Here you set to which state (on / off) the gates should be set on a trigger to clear .						
clearskips			A trigger here removes the “skip” setting from all steps.						
clearrepeats			A trigger here resets the number of repeats to 1 for each step.						
clearstartend			A trigger here clears the manual settings of the start and end step. So the sequence will be played in its full length (again).						
gatelength		0.5	<p>The gate length in input clock cycles. A value of 0.5 thus means half a clock cycle. A steady input clock is needed for this to work. Please note that if the gate length is ≥ 1.0, two succeeding notes will get a steady gate, which essentially means legato.</p> <p>If you don't use a steady clock, set this parameter to 0. This will output a minimal gate length of about 10 ms (basically just a trigger).</p>						
keyboardmode	1 ◊ 2 ◊ 3	1	This input sets how a keyboard, that is hooked to keyboardcv , and keyboardgate should be used for directly playing notes. You can set it to 0 , 1 or 2 .						
			<table border="1"> <tr> <td>0</td><td>ignore the keyboard inputs</td></tr> <tr> <td>1</td><td>keyboard and sequencer play together, keyboard has precedence</td></tr> <tr> <td>2</td><td>mute sequencer, just play keyboard</td></tr> </table>	0	ignore the keyboard inputs	1	keyboard and sequencer play together, keyboard has precedence	2	mute sequencer, just play keyboard
0	ignore the keyboard inputs								
1	keyboard and sequencer play together, keyboard has precedence								
2	mute sequencer, just play keyboard								
keyboardcv			The pitch input of a keyboard. This is used for playing along with the keyboardmode or recording with recordmode .						
keyboardgate			The gate input of a keyboard. A positive gate enables play along (see keyboardmode) and also recording, if recordmode is set accordingly.						
recordmode	1 ◊ 2 ◊ 3	0	Use this input to record melodies played with a keyboard (namely keyboardcv and keyboardgate) into the sequencer. There are three possible settings:						
			<table border="1"> <tr> <td>0</td><td>don't record</td></tr> <tr> <td>1</td><td>record, notes longer than one step will automatically tie steps via the gate pattern</td></tr> <tr> <td>2</td><td>record, don't tie notes. Ignore the length of the input note</td></tr> </table>	0	don't record	1	record, notes longer than one step will automatically tie steps via the gate pattern	2	record, don't tie notes. Ignore the length of the input note
0	don't record								
1	record, notes longer than one step will automatically tie steps via the gate pattern								
2	record, don't tie notes. Ignore the length of the input note								
recordsilence		0	When this input is set to 1 while recording, silence will be recorded while keyboardgate is off. Otherwise you can just add notes to the sequence.						

Input	Type	Default	Description				
copy			A trigger here copies the complete sequence (if copymode = 0) or just the current page of the sequence (if copymode = 1) to an internal clipboard. The clipboard is not part of any preset and is also not saved to the SD card. It can be used later for pasting its data to another preset or another page of a sequence.				
copymode	1 o 2 o 3	1	Determines whether copy and paste works with the complete sequence or just with the current page (that part of the sequence that is currently shown on the faders).				
			<table border="1"> <tr> <td>0</td><td>copy and paste works on the sequence as a whole</td></tr> <tr> <td>1</td><td>copy and paste just works on the current page</td></tr> </table>	0	copy and paste works on the sequence as a whole	1	copy and paste just works on the current page
0	copy and paste works on the sequence as a whole						
1	copy and paste just works on the current page						
paste			A trigger here copies the steps from the clipboard either to the complete sequence (copymode = 0) or just to the current page (copymode = 1).				
pastefaders			This is like paste , but just the values of the faders of the current fadermode are copied.				
pastebuttons			This is like paste , but just the values of the faders of the current buttonmode are copied. Note: the button mode "start / end" is not supported by copy and paste.				
linktonext		0	<p>This setting allows you to create motoquencer tracks that have more than one CV or gate output for each step. If you set this to 1, the next motoquencer circuit in your patch will be synchronized to this one. This means that it always plays the same step number - including all fancy operating like shiftsteps, startstep, endstep, form, pattern and autoreset. All those inputs and also clock and reset are ignored by the next motoquencer.</p> <p>The same holds for the "repeats" and "skip" setting of the steps.</p> <p>fadermode and buttonmode are extended to the next motoquencers by adding 10 for each motoquencer to follow. So fadermode = 10 will show the CV of next motoquencer in the faders. fadermode = 11 the CV randomization of the next motoquencer. fadermode = 20 show the CV of the third linked motoquencer and so on.</p> <p>Don't set fadermode or buttonmode on the linked motoquencers. They will be ignored there.</p> <p>Note: The linktonext setting cannot be dynamically changed. It needs to be fixed 0 or 1. You cannot use any button or internal cable or other methods to change it while the patch is running.</p>				
luckychance		1.0	Sets the chance for a step to be affected by the next "lucky" operation (see triggers below).				

Input	Type	Default	Description								
luckyscope	1 o 2 o 3	0	Determines which part of the sequence is affected by the “lucky” operations. Depending on this setting the following steps are affected:								
			<table border="1"> <tr> <td>0</td><td>All steps between the current start and end step</td></tr> <tr> <td>1</td><td>All steps</td></tr> <tr> <td>2</td><td>All steps between start and end on the current page</td></tr> <tr> <td>3</td><td>All steps on the current page</td></tr> </table>	0	All steps between the current start and end step	1	All steps	2	All steps between start and end on the current page	3	All steps on the current page
0	All steps between the current start and end step										
1	All steps										
2	All steps between start and end on the current page										
3	All steps on the current page										
luckyamount	o ↕ 1	1.0	Sets the amount of change that a “lucky” operation does to a step. The meaning depends on the operation. See the parameters below.								
luckyfaders	█		Moves the currently selected faders (according to fadermode) to new random positions. luckyamount sets the maximum value of the fader, where 1 allows the maximum.								
luckybuttons	█		Randomly toggles the currently selected buttons (according to buttonmode). luckyamount only has an effect when the gate patterns are selected, since here, four different values are possible. luckyamount restricts them if it is lower than 1 .								
lucky cvs	█		Replaces the affected steps’ CVs with a new random CVs. The lowest possible CV is cibase . If luckyamount is 1 , the highest possible CV is cibase + cvrange , otherwise it is cibase + luckyamount × cvrange .								
lucky cvdrift	█		Modifies the affected steps’ CV randomly up or down. They will stay in the CV range set by cibase and cvrange . luckyamount controls the amount of change.								
luckyspread	█		First computes the average CV of all steps. Then changes the CV values of the affected steps such that their distance to the average increases or decreases. If luckyamount is greater than 0.5 , the distance is increased. Otherwise it is decreased.								
luckyinvert	█		Inverts the CVs of the affected steps within the allowed CV range. luckyamount has no influence.								
luckyrandomizecv	█		Sets the “randomize CV” values of the affected steps to random values (yes, this is double randomization). The luckyamount sets the maximum randomization value that will be set.								
luckygates	█		Sets the gates of the affected steps randomly to on or off. The chance for on is determined by luckyamount . So with luckyamount = 0 you clear all gates and with luckyamount = 1 you set all gates.								
luckyskips	█		Sets the “skip this step” setting of the affected steps randomly to skip or normal. The chance for skip is determined by luckyamount .								

Input	Type	Default	Description
<code>luckyties</code>			Sets the “tie this step to the next” setting of the affected steps randomly to tie or normal. This is the same as setting the gate pattern to the upper most position. The chance for tie is determined by <code>luckyamount</code> .
<code>luckygatepattern</code>			Randomizes the gate pattern of the selected steps (there are four different values: once, all, hold and tie). Use <code>luckyamount</code> to reduce that set.
<code>luckygateprob</code>			Sets the “randomize gate” values of the affected steps to random values (yes, this is double randomization). The <code>luckyamount</code> sets the minimum randomization value that will be set (yes, this is inverted). So with <code>luckyamount = 1</code> you disable randomization and make the gates play always. With <code>luckyamount = 0</code> you set the gate probability to the lowest possible value (still not 0).
<code>luckyrepeats</code>			Randomly sets the number of repeats of the affected steps to something between 1 and 16 (the maximum). The <code>luckyamount</code> determines the maximum repetition number, where 1 stands for a maximum of 16 repetitions.
<code>luckyratchets</code>			Randomly sets the number of ratches of the affected steps to something between 1 and 8 (the maximum). The <code>luckyamount</code> determines the maximum ratchet number, where 1 stands for a maximum of 8 ratchets.
<code>luckyshuffle</code>			Randomly swaps all affected affected steps (their playing order) together will all their attributes. <code>luckyamount</code> has no influence.
<code>luckyreverse</code>			Reverses the playin gorder of the affected steps. <code>luckyamount</code> has not influence.

Input	Type	Default	Description																										
root	1◦2◦3	0	<p>Set the root note here. 0 means C, 1 means C#, 2 means D and so on. If you multiply the value of an input like I1 with 120, then you can use a 1V/Oct input for selecting the root note via a sequencer, MIDI keyboard or the like. Also then you are compatible with the ROOT CV input of the Sinfonion.</p> <table border="1"> <tr><td>0</td><td>C</td></tr> <tr><td>1</td><td>C#</td></tr> <tr><td>2</td><td>D</td></tr> <tr><td>3</td><td>D#</td></tr> <tr><td>4</td><td>E</td></tr> <tr><td>5</td><td>F</td></tr> <tr><td>6</td><td>F#</td></tr> <tr><td>7</td><td>G</td></tr> <tr><td>8</td><td>G#</td></tr> <tr><td>9</td><td>A</td></tr> <tr><td>10</td><td>A#</td></tr> <tr><td>11</td><td>B</td></tr> <tr><td>12</td><td>C</td></tr> </table>	0	C	1	C#	2	D	3	D#	4	E	5	F	6	F#	7	G	8	G#	9	A	10	A#	11	B	12	C
0	C																												
1	C#																												
2	D																												
3	D#																												
4	E																												
5	F																												
6	F#																												
7	G																												
8	G#																												
9	A																												
10	A#																												
11	B																												
12	C																												

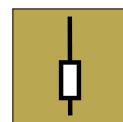
Input	Type	Default	Description																								
degree	1 o 2 o 3	0	Set the musical scale. This is a number from 0 to 11 . At 12 this repeats over again. Please refer to the introduction for the list of scales. If you multiply an input like I1 with 120 , this will internally scale to one scale per semitone and you are compatible with the DEGREE CV input of the Sinfonion.																								
			<table border="1"> <tr><td>0</td><td>lyd - Lydian major scale (it has a #4)</td></tr> <tr><td>1</td><td>maj - Normal major scale (ionian)</td></tr> <tr><td>2</td><td>X7 - Mixolydian (dominant seven chords)</td></tr> <tr><td>3</td><td>sus - mixolydian with 3rd/4th swapped</td></tr> <tr><td>4</td><td>alt - Altered scale</td></tr> <tr><td>5</td><td>hm⁵ - Harmonic minor scale from the 5th</td></tr> <tr><td>6</td><td>dor - Dorian minor (minor with #13)</td></tr> <tr><td>7</td><td>min - Natural minor (aeolian)</td></tr> <tr><td>8</td><td>hm - Harmonic minor (b6 but #7)</td></tr> <tr><td>9</td><td>phr - Phrygian minor scale (with b9)</td></tr> <tr><td>10</td><td>dim - Diminished scale (whole/half tone)</td></tr> <tr><td>11</td><td>aug - Augmented scale (just whole tones)</td></tr> </table>	0	lyd - Lydian major scale (it has a #4)	1	maj - Normal major scale (ionian)	2	X7 - Mixolydian (dominant seven chords)	3	sus - mixolydian with 3 rd /4 th swapped	4	alt - Altered scale	5	hm ⁵ - Harmonic minor scale from the 5 th	6	dor - Dorian minor (minor with #13)	7	min - Natural minor (aeolian)	8	hm - Harmonic minor (b6 but #7)	9	phr - Phrygian minor scale (with b9)	10	dim - Diminished scale (whole/half tone)	11	aug - Augmented scale (just whole tones)
0	lyd - Lydian major scale (it has a #4)																										
1	maj - Normal major scale (ionian)																										
2	X7 - Mixolydian (dominant seven chords)																										
3	sus - mixolydian with 3 rd /4 th swapped																										
4	alt - Altered scale																										
5	hm ⁵ - Harmonic minor scale from the 5 th																										
6	dor - Dorian minor (minor with #13)																										
7	min - Natural minor (aeolian)																										
8	hm - Harmonic minor (b6 but #7)																										
9	phr - Phrygian minor scale (with b9)																										
10	dim - Diminished scale (whole/half tone)																										
11	aug - Augmented scale (just whole tones)																										
select1			<p>Gate input for selecting the <i>root note</i> as being an allowed interval. When you want to create a playing interface for live operation you can patch the output of a toggle button (made with the circuit [button]) here.</p> <p>Note: When all select and selectfill inputs are 0, automatically all seven scale notes are selected, i.e. select1 ... select13 will be set to one.</p>																								
select3			Gate input for selecting the 3 rd .																								
select5			Gate input for selecting the 5 th .																								
select7			Gate input for selecting the 7 th .																								
select9			Gate input for selecting the 9 th (which is the same as the 2 nd).																								
select11			Gate input for selecting the 11 th (which is the same as the 4 th).																								
select13			Gate input for selecting the 13 th (which is the same as the 6 th).																								

Input	Type	Default	Description
<code>selectfill1</code>		off	Selects the alternative 9 th (i.e. the 9 th that is <i>not</i> in the scale).
<code>selectfill2</code>		off	Selects the alternative 3 rd (i.e. the 3 rd that is <i>not</i> in the scale).
<code>selectfill3</code>		off	Selects the alternative 4 th or 5 th . In most cases this is the diminished 5 th .
<code>selectfill4</code>		off	Selects the alternative 13 th (i.e. the 1 st 3 that is <i>not</i> in the scale).
<code>selectfill5</code>		off	Selects the alternative 7 th (i.e. the 7 th that is <i>not</i> in the scale).
<code>tuningmode</code>		off	While this is 1 , the circuit will output the value set by <code>tuningpitch</code> instead of the actual pitch. This is meant to be a help for tuning your VCOs.
<code>tuningpitch</code>	$\frac{1V}{Oct}$	0V	This pitch CV will be output while the tuning mode is active.
<code>transpose</code>	$\frac{1V}{Oct}$	0V	This value is being added to the output pitch when not in tuning mode. It can be used for musical transposition or adding a vibrato.
<code>select</code>			The <code>select</code> input allows you to overlay buttons and LEDs with multiple functions. If you use this input, the circuit will process the buttons and LEDs just as if <code>select</code> has a positive gate signal (usually you will select this to 1). Otherwise it won't touch them so they can be used by another circuit. Note: even if the circuit is currently not selected, it will nevertheless work and process all its other inputs and its outputs (those that do not deal with buttons or LEDs) in a normal way.
<code>selectat</code>	1 \circ 2 \circ 3		This input makes the <code>select</code> input more flexible. Here you specify at which value <code>select</code> should select this circuit. E.g. if <code>selectat</code> is 0 , the circuit will be active if <code>select</code> is <i>exactly</i> 0 instead of a positive gate signal. In some cases this is more convenient.
<code>preset</code>	1 \circ 2 \circ 3		This is the preset number to save or to load. Note: the first preset has the number 0 , not 1 ! For the whole story on presets please refer to page 19. This circuit has 4 presets, so this number ranges from 0 to 3 .
<code>loadpreset</code>			A trigger here loads a preset. As a speciality you can use the trigger for selecting a preset at the same time.
<code>savepreset</code>			A trigger here saves a preset.
<code>clear</code>			A trigger here loads the default start state into the circuit. The presets are not affected, unless you use direct preset switching with the <code>preset</code> input and without triggers. And that case the current preset is also cleared.
<code>clearall</code>			A trigger here loads the default start state into the circuit and into all of its presets.
<code>dontsave</code>		0	If you set this to 1 , the state of the circuit will not be saved to the SD card and not loaded from the SD card when the Droid starts.

Output	Type	Description
<code>cv</code>		The CV output (or pitch output, if you use <code>quantize</code>).
<code>gate</code>		The gate output.
<code>startofsequence</code>		Outputs a trigger whenever the sequencer starts playing from the beginning. This can be used for synchronizing with other sequencers. An external <code>reset</code> will also cause this output to trigger.
<code>startofpart</code>		Outputs a trigger whenever a form part starts again. This is only interesting when you use <code>form</code> .
<code>startstepout</code>	<code>1◦2◦3</code>	Outputs the current start step. This is useful in case it has been interactively set with the buttons and you need that information for another circuit.
<code>endstepout</code>	<code>1◦2◦3</code>	Outputs the current end step. This is useful in case it has been interactively set with the buttons and you need that information for another circuit.
<code>currentstep</code>	<code>1◦2◦3</code>	Outputs the number of the step that is currently being played (starting from 0).
<code>currentpage</code>	<code>1◦2◦3</code>	Outputs the number of the fader page that is currently played, i.e. the value you would have to feed into <code>page</code> in order to see the currently being played step.

13.39 motorfader - Create virtual fader in M4 controller

The circuit provides the most basic access to motor faders and supports switching between presets, overlayed functions and force feedback.



For the basics about these ideas and the M4 in general, please read the introduction to the M4 on page [43](#).

Presets

Let's start with presets and make a simple example with one P2B8 and one M4 controller. First we need to declare both in our patch:

[p2b8]
[m4]

Let's use the first fader as a simple CV source to be output on **01**. And four buttons should select four different presets of that fader. Those are grouped into a button with the circuit **buttongroup** (see page [108](#)):

```
[buttongroup]
  button1 = B1.1
  button2 = B1.2
  button3 = B1.3
  button4 = B1.4
  led1 = L1.1
  led2 = L1.2
  led3 = L1.3
  led4 = L1.4
  output = _PRESET
```

This circuit will switch between the values **0**, **1**, **2** and **3** and output that number to the intercal cable **_PRESET**. Now let's add the fader definition:

```
[motorfader]
  fader = 1
  preset = _PRESET
  output = 01
```

That's really all. **fader = 1** selects the first motor fader in your setup. All faders are simply enumerated, so **fader = 7** would select the third fader on the second M4.

The output **01** now always outputs the current setting of the fader. The range is 0 V ... 10 V - just like with pots of the controllers.

Hitting the buttons will switch to one of the four presets and move the fader to the position corresponding to current value of that preset.

Faders with multiple functions

The second way to use the motor faders is to assign multiple functions to one fader and then switch between those functions. The crucial difference to the presets is, that for every function there is a *dedicated output*.

Let's now change our example so that we use one fader controlling *four* CV sources, but without any presets for the while. The start is the same (just we renamed the internal cable to **_FUNCTION**):

```
[buttongroup]
  button1 = B1.1
```

```
button2 = B1.2
button3 = B1.3
button4 = B1.4
led1 = L1.1
led2 = L1.2
led3 = L1.3
led4 = L1.4
output = _FUNCTION
```

No we need a separate **motorfader** circuit for each function. And instead of choosing a preset, we need to **select** each circuit when the active button selects its function:

```
[motorfader]
  fader = 1
  select = _FUNCTION
  selectat = 0
  output = 01
```

```
[motorfader]
  fader = 1
  select = _FUNCTION
  selectat = 1
  output = 02
```

```
[motorfader]
  fader = 1
  select = _FUNCTION
  selectat = 2
  output = 03
```

```
[motorfader]
  fader = 1
  select = _FUNCTION
  selectat = 3
  output = 04
```

As you can see: each fader has a **selectat** input match-

ing one of the buttons of the buttongroup. And each fader also sends its output to one of the main outputs of the master.

There is one possible simplification: Instead of using **_FUNCTION** and **selectat**, we also could use the LED outputs of the button group directly:

```
[buttongroup]
button1 = B1.1
button2 = B1.2
button3 = B1.3
button4 = B1.4
led1 = L1.1
led2 = L1.2
led3 = L1.3
led4 = L1.4

[motorfader]
fader = 1
select = L1.1
output = 01

[motorfader]
fader = 1
select = L1.2
output = 02

[motorfader]
fader = 1
select = L1.3
output = 03

[motorfader]
fader = 1
select = L1.4
output = 04
```

Notches

Maybe the coolest feature of the M4 is the haptic feedback. The M4 uses its motors in order to give you force feedback. This is done in various forms.

The most useful form is to use artificial “notches” or “dents”. Try that out by setting **notches** to a number, e.g. 8:

```
notches = 8
```

This changes the behaviour of the fader in two ways:

1. The output value is now a discrete whole number from **0** up to **7**.
2. When you move the fader you feel eight artificial dents. That's really hard to explain. Try it out!

These notches are super helpful especially in live performances. You instantly *feel* where your are. You don't need any visual feedback. You can very reliably set a value without looking.

The maximum number of notches is **201**. But if you select more than 25 notches, the force feedback is turned off as the notches would get too small to work.

There are also two other variants of force feed back:

Binary switch

If you set **notches = 2**, you turn the fader into a binary switch. The output will be **0** if the fader is in the bottom position and **1** on the top. Just move the fader away from its position and it will immediately snap to the other side.

Pitch bend wheel

Setting **notches = 1** will convert the fader into a kind of pitch bend wheel. It always wants to stay in the middle, where it outputs a value of **0.5**. If you move it away from the center position, it creates a force back to the center that is the greater the nearer you are to the top or bottom. As soon as you release it, it snaps back to the middle.

Modifying one value with two virtual faders

The sharing of virtual faders is a bit more tricky to explain and you probably won't need it. It means that you use two **motorfader** circuits for controlling the same output value. Why would you do this?

I have added that feature when building a motor fader based MIDI control for my audio interface. I have one mode where every of eight faders controls the main volume of one of eight voices.

And then I have a “drill down” for each voice, where the first fader is the main volume, the second fader the head phone, the third the volume of an aux channel and so on.

So now I can control the volume of voice 3 either with the third fader in the “global” volume control or with the first fader the drill down of voice 3. This leads to an output collision since two circuits would try to modify the same output, even if always just one of the two motor fader circuits is selected.

The solution to this problem is the **sharewithnext** input. Put the two **motorfader** circuits next to each other into your patch. Put a **sharewithnext = 1** into the first one. Don't use the **output** there. Now both virtual faders will control the output that is defined in the second **motorfader** circuit:

```
[motorfader]
  fader = 1
  select = _GLOBAL
  sharewithnext = 1
```

```
[motorfader]
  fader = 3
  select = _DRILLDOWN_3
```

```
output = _VOLUME_3
```

Note: if you are using **notches**, make sure that both **motorfader** circuits have the same number of notches!

Input	Type	Default	Description
fader	1°2°3	1	The number of the motor fader to use, starting with 1 for the first fader in the first M4. 5 selects the first fader in the second M4 and so on.
startvalue		0.0	This sets the value the fader gets when you start this circuit this first time or when a trigger to clear happens.
notches	1°2°3	0	Number of artificial notches. 0 disables the notches. 1 creates a pitch bend wheel. 2 creates a binary switch with the output values 0 and 1 . Higher number create that number of notches. E.g. 8 creates eight notches and output will output one of the value 0, 1, ... 8 . The maximum allowed number is 25 .
ledvalue			When you use this input, it will override the brightness of the LED below the fader, but just when this circuit is selected.
ledcolor			When you use this input, it will set the color of the LED below the fader, when the circuit is selected. If the LED is off, this setting has now impact.
sharewithnext		0	If set to 1 , the output output will not be used but the circuit shares its output with the next motorfader circuit.
select			The select input allows you to overlay buttons and LEDs with multiple functions. If you use this input, the circuit will process the buttons and LEDs just as if select has a positive gate signal (usually you will select this to 1). Otherwise it won't touch them so they can be used by another circuit. Note: even if the circuit is currently not selected, it will nevertheless work and process all its other inputs and its outputs (those that do not deal with buttons or LEDs) in a normal way.
selectat	1°2°3		This input makes the select input more flexible. Here you specify at which value select should select this circuit. E.g. if selectat is 0 , the circuit will be active if select is <i>exactly</i> 0 instead of a positive gate signal. In some cases this is more convenient.
preset	1°2°3		This is the preset number to save or to load. Note: the first preset has the number 0 , not 1 ! For the whole story on presets please refer to page 19 . This circuit has 8 presets, so this number ranges from 0 to 7 .
loadpreset			A trigger here loads a preset. As a speciality you can use the trigger for selecting a preset at the same time.
savepreset			A trigger here saves a preset.
clear			A trigger here loads the default start state into the circuit. The presets are not affected, unless you use direct preset switching with the preset input and without triggers. And that case the current preset is also cleared.
clearall			A trigger here loads the default start state into the circuit and into all of its presets.

Input	Type	Default	Description
<code>dontsave</code>		<code>0</code>	If you set this to <code>1</code> , the state of the circuit will not saved to the SD card and not loaded from the SD card when the Droid starts.

Output	Type	Description
<code>output</code>		Output the current value if the virtual motor fader (don't use this if you are using <code>sharewithnext</code>).

13.40 notchedpot - Helper circuit for pots (OBSOLETE)

This circuit has been superseded by the new circuit pot (see page 242). It will be removed in the next firmware version. If you use it in your patch, better replace it.



pot can do all notchedpot can do and much more. So notchedpot will be removed soon.

This little circuit simulates a potentiometer with a notch at the center. It helps you exactly selecting the center position by defining a range that is considered to be the center. This range is called "notch" and defaults to 10% of the

available range. You can set the size of the notch via the notch input. Here is an example:

```
[notchedpot]
pot      = P1.1
notch   = 15%
output  = _ACTIVITY

[algoquencer]
activity = _ACTIVITY
...
```

For a second use case there is the output bipolar. That converts a normal pot into one with range from -1.0 to 1.0. This example also shows how to disable the notch, if you do not need it here:

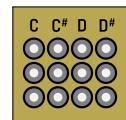
```
[notchedpot]
pot      = P1.1
notch   = 0
bipolar = 01 # Send -10V ... +10V to 01
```

Input	Type	Default	Description
pot	o ↕ 1		Wire your pot here, e.g. P1.1
notch	~~~	0.1	Optionally set the notch size, if you do not like the default of 0.1. The maximum allowed value is 0.5. Greater values will be reduced to that.

Output	Type	Description
output	o ↕ 1	Your pot output comes here. It still goes from 0.0 to 1.0.
bipolar	~~~	Optional output with a range from -1.0 to 1.0, where the center notch is at 0.0.
absbipolar	~~~	A variation of bipolar that always outputs a positive value, i.e. the pot will go 1 ... 0.5 ... 0 ... 0.5 ... 1
lefthalf	~~~	This output allows you to split the pot into two hemispheres. Here you get 1.0 ... 0.0 while the pot is in the left half. In the middle and right of it you always get 0.
righthalf	~~~	This is the same but for the right half. It outputs 0 while the pot is in the left half and 0.0 ... 1.0 from the middle to the fully right position.
lefthalfinv	~~~	This outputs 1.0 - lefthalf, i.e. the value range 0.0 ... 1.0 ... 1.0 when the pot moves left → mid → right.
righthalfinv	~~~	This outputs 1.0 - righthalf, i.e. the value range 1.0 ... 1.0 ... 0.0 when the pot moves left → mid → right.

13.41 notebuttons - Note Selection Buttons

This simple utility combines 12 buttons, just like radio buttons, into a selector for a note such as C, C[#], D, D[#] and so on. It is similar to **buttongroup**, but much simpler.



And it allows 12 buttons. The output is either a number from **0** to **11** - or alternatively on a $\frac{1}{12}$ V per semitone base. The latter one is ideal for sending it to external sequencers or quantizers as they often adopt that scheme.

The following example uses all eight buttons of the first controller plus the first column of the second controller for selecting the twelve notes. It sends the currently selected note to **07** in a 1 V per octave scheme:

```
[notebuttons]
button1 = B1.1
button2 = B1.2
button3 = B2.1
button4 = B1.3
button5 = B1.4
button6 = B2.3
button7 = B1.5
button8 = B1.6
button9 = B2.5
button10 = B1.7
button11 = B1.8
button12 = B2.7
led1    = L1.1
led2    = L1.2
led3    = L2.1
led4    = L1.3
led5    = L1.4
led6    = L2.3
led7    = L1.5
led8    = L1.6
led9    = L2.5
led10   = L1.7
```

```
led11    = L1.8
led12    = L2.7
semitone = 07
```

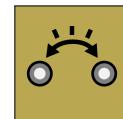
Input	Type	Default	Description
button1 ... button12			Wire 12 buttons to these 12 inputs.
clock			When you use this jack, all button presses are quantized in time to the next clock pulse arriving here. That makes it easier to switch the note exactly in time.
startnote	1 ◊ 2 ◊ 3		Specify the note that should be selected when the Droid starts and no state is loaded, or when a trigger to clear or clearall happened. This is an integer number from 0 to 11 .
select			The select input allows you to overlay buttons and LEDs with multiple functions. If you use this input, the circuit will process the buttons and LEDs just as if select has a positive gate signal (usually you will select this to 1). Otherwise it won't touch them so they can be used by another circuit. Note: even if the circuit is currently not selected, it will nevertheless work and process all its other inputs and its outputs (those that do not deal with buttons or LEDs) in a normal way.
selectat	1 ◊ 2 ◊ 3		This input makes the select input more flexible. Here you specify at which value select should select this circuit. E.g. if selectat is 0 , the circuit will be active if select is <i>exactly</i> 0 instead of a positive gate signal. In some cases this is more convenient.
preset	1 ◊ 2 ◊ 3		This is the preset number to save or to load. Note: the first preset has the number 0 , not 1 ! For the whole story on presets please refer to page 19. This circuit has 16 presets, so this number ranges from 0 to 15 .
loadpreset			A trigger here loads a preset. As a speciality you can use the trigger for selecting a preset at the same time.
savepreset			A trigger here saves a preset.
clear			A trigger here loads the default start state into the circuit. The presets are not affected, unless you use direct preset switching with the preset input and without triggers. And that case the current preset is also cleared.
clearall			A trigger here loads the default start state into the circuit and into all of its presets.
dontsave		0	If you set this to 1 , the state of the circuit will not be saved to the SD card and not loaded from the SD card when the Droid starts.

Output	Type	Description
led1 ... led12		Wire the LEDs in the buttons to these 12 outputs.
output	1 ◊ 2 ◊ 3	Here you get a number from 0 to 11 , according to the currently selected button.
semitone	$\frac{1V}{Oct}$	Here you get the same as output , but divided by 120. When you patch this output to a CV output of the DROID, like 01 , it will output the note as a semitone on a 1 V per octave scheme.

Output	Type	Description
gate		This output is 1 as long as one of the buttons is held. You can use that together with the semitone output to use the notebuttons as a CV/gate keyboard with 12 keys.

13.42 nudge - Modify a value in steps using two buttons

This small utility allows you to modify a value up and down in fixed steps using two buttons. This value can be persistent so it survives a power cycle.



Here is an example for a simple CV source that outputs a value between -2 V and 2 V:

```
[nudge]
minimum = -2V
maximum = 2V
amount = 1V
buttonup = B1.1
buttondown = B1.3
ledup = L1.1
leddown = L1.3
output = 01
```

Note: If you press both buttons at the same time, the value will be reset to its start value.

You can extend this into an octave switch by using the input **offset**, which will be added to the output:

```
[nudge]
minimum = -2V
maximum = 2V
amount = 1V
buttonup = B1.1
buttondown = B1.3
ledup = L1.1
leddown = L1.3
output = 01
offset = I1
```

If you now feed some V/Oct source, such as the pitch output of a sequencer, to **I1**, it will be shifted up and down for up to two octaves.

Another application might be to fine tune an oscillator. Here you set the nudge steps (set by **amount**) a lot smaller. Also it is allowed to leave out **minimum** and **maximum** and thus make the possible range unrestricted. Note: **1V / 1200** means essentially a step size of $\frac{1}{1200}$ of an octave, which is $\frac{1}{100}$ of a semitone, which is also known as *one cent*:

```
[nudge]
amount = 1V / 1200
buttonup = B1.1
buttondown = B1.3
ledup = L1.1
leddown = L1.3
output = 01
offset = I1
```

A third application could be a button for selecting a certain input number for - let's say - an euclidean rhythm pattern:

```
[nudge]
amount = 1
buttonup = B1.1
ledup = L1.1
minimum = 3
maximum = 7
wrap = 1
output = _BEATS

[euklid]
clock = G1
length = 16
beats = _BEATS
output = G3
```

Note: Here only one button is wired. In addition **wrap** is set to **1**, which means that after reaching the maximum value, the next value will be the minimum value. Here each press of the button **B1.1** forwards the number of beats in the matter $3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 3$ and so on...

Understanding the LEDs

By nudging the value below the center value the buttonup LED will be off and the brightness of the buttondown LED will gradually increase indicating how much the value is set below this center value. It remains maximally bright at the minimum.

Vice versa by nudging the value above the center value the buttondown LED will be off and the brightness of the buttonup LED will gradually increase indicating how much the value is set above this center value. It remains maximally bright at the maximum.

And if the value is exactly in the middle between **maximum** and **minimum**, both LEDs are maximally bright. Here you have to have in mind that this must be **exactly** in the middle. Of course, this only works if the distance between **maximum** and **minimum** is an exact odd number of **amounts**.

Input	Type	Default	Description
buttonup			Button for nudging the value up by one step
buttondown			Button for nudging the value down by one step
amount		0.1	Amount to modify the value by on each press. This must be a value > 0
startvalue		0.0	The value this circuit starts with or is being reset to if you use the clear input.
minimum			The minimum possible value. If you do not wire this, the value can go down infinitely.
maximum			the maximum possible value. If you do not wire this, the value can go up infinitely.
wrap		0	Set this to 1 in order to have the value wrap around if the minimum or the maximum has been exceeded. Note: wrap does only work if you set minimum and maximum .
offset		0.0	This value is being added to the output.
select			The select input allows you to overlay buttons and LEDs with multiple functions. If you use this input, the circuit will process the buttons and LEDs just as if select has a positive gate signal (usually you will select this to 1). Otherwise it won't touch them so they can be used by another circuit. Note: even if the circuit is currently not selected, it will nevertheless work and process all its other inputs and its outputs (those that do not deal with buttons or LEDs) in a normal way.
selectat	1 o 2 o 3		This input makes the select input more flexible. Here you specify at which value select should select this circuit. E.g. if selectat is 0 , the circuit will be active if select is <i>exactly</i> 0 instead of a positive gate signal. In some cases this is more convenient.
preset	1 o 2 o 3		This is the preset number to save or to load. Note: the first preset has the number 0 , not 1 ! For the whole story on presets please refer to page 19. This circuit has 16 presets, so this number ranges from 0 to 15 .
loadpreset			A trigger here loads a preset. As a speciality you can use the trigger for selecting a preset at the same time.
savepreset			A trigger here saves a preset.
clear			A trigger here loads the default start state into the circuit. The presets are not affected, unless you use direct preset switching with the preset input and without triggers. And that case the current preset is also cleared.
clearall			A trigger here loads the default start state into the circuit and into all of its presets.
dontsave		0	If you set this to 1 , the state of the circuit will not be saved to the SD card and not loaded from the SD card when the Droid starts.

Output	Type	Description
<code>ledup</code>		Wire this to the LED in the button for nudging up. It will indicate the current value.
<code>leddown</code>		Wire this to the LED in the button for nudging down. It will indicate the current value.
<code>output</code>		The output of the current value plus value if offset .

13.43 octave - Multi-VCO octave animator

This circuit is used to control the pitches of three oscillators by octave or even fifths. It also allows a linear detune in order to make the common sound of the VCOs sound fatter.



Here is an example for a setup where the octave spreading and the detune is controlled with two pots:

[octave]

```
input    = I1
output1 = 01
output2 = 02
output3 = 03
spread   = P1.1
detune   = P1.2
```

Patch the 1 V / octave inputs of three VCOs at **01**, **02** and **03**. Tune all VCOs at exactly the same pitch. Patch the pitch output from your sequencer, quantizer or whatever to **I1**.

Now with the pot **P1.1** turned fully left nothing changes. All VCOs will get exactly the same pitch. As you turn up the pot the pitches of the VCOs 2 and 3 will start to get octaved up more and more until VCO 2 is two octaves above VCO 1 and VCO 3 is four octaves above VCO 1.

If you add **fifths = on** then intermediate steps shift the pitch by perfect fifths.

Note: The output **output1** was implemented just for sake of completeness. It passes through the input to **output1**, since the pitch of VCO 1 is never detuned nor pitched up. If you are running low in outputs then some use a passive multiple or stacked cable and connect VCO 1 externally the pitch and thus save one output.

Detune

In the example, if you turn **P1.2**, VCO 2 will be detuned up and VCO 3 down. A very slight turn will get you the nice fat classical detune sound. The speciality here is: the detune is *linear*. This means that the detune is always done by the same number of *Hertz* - regardless of the current pitch. This is done by automatically adapting the detune voltage to be less in higher pitches and greater in lower pitches. The result is a beating independent of pitch.

Animation

Since everything in DROID is CV'able so is **spread**. A nice application is to use a sequencer or clocked random generator for *animating* the octaving. Here is an example:

```
[random]
trigger  = I1
output   = _RANDOM
```

[octave]

```
input    = I1
output1 = 01
output2 = 02
output3 = 03
spread   = _RANDOM * P1.1
```

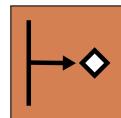
Now **P1.1** controls the depth of random octave animation.

Input	Type	Default	Description
input	 $\frac{1V}{Oct}$	0V	The general pitch information on a 1 V / octave base to be used for the three VCOs.
spread		0	The amount of octave spread between output1 and output3 . At a value of 1.0 the spread is four octaves.
detune		0.0	The amount of linear detune of VCO 2 and 3. This is <i>not</i> on a 1 V / octave base but corresponds to an absolute frequency difference in Hertz. The exact frequency difference cannot be set here, since that depends on how you have tuned your VCOs. But the rule is the following: If input is a 0 V and detune is 1.0 , the detune is by four semitones. And for an input of 1 V (one octave higher) it is just two semitones, because that results in the same frequency difference. For 2 V (two octaves up) it is just one semitone and for 3 V half a semitone (and so on). Best thing is to simply try out and listen!
fifths		off	Set this to 1 or on if you want to include perfect fifths as intermediate steps.

Output	Type	Description
output1 ... output3	 $\frac{1V}{Oct}$	Outputs for the 1 V / octave of the three VCOs. output1 is an exact copy of input so you could omit that and rather patch VCO 1 to the original pitch CV.

13.44 once - Output one trigger after the Droid has started

This circuit outputs exactly one trigger after the Droid module has started. You can set a delay for that to happen.



Example:

```
[once]
delay = 0.2 # 200 ms
trigger = _DO_ONCE
```

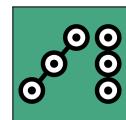
The applications are up to you. Maybe you want to automatically start something when the Droid starts or update some MIDI data or whatever weird other idea you have in mind.

Input	Type	Default	Description
delay		0.01	Set a delay in seconds after the Droid's start before the trigger triggers. Note: the default value is 10 ms, not zero. This allows all attached controllers to have sent at least one update before and the real pot values etc. are available at the circuits.
onlycoldstart		0	If you set this input to 1, once just sends a trigger after a cold start, <i>only</i> . A cold start means that the Droid has been powered up. Pressing the button for loading a new patch and does a warm start.

Output	Type	Description
trigger		The trigger is output here.

13.45 polytool - Change number of voices in polyphonic setups

The polytool is an intelligent “transformer” that can map melodies with N parallel notes to synth voices with M parallel voices and can thus change the polyphony of a melody.



This functionality is inspired by MIDI to CV interfaces (such as `midiiin` (see page 183)), which need to deal with the almost unlimited possible polyphony of MIDI, where 127 parallel notes are possible and where the interface needs to suffice with a fixed limited number of CV/gate outputs.

The usage is very simple: patch your input voices (CV/gate pairs) into `pitchinputX` and `gateinputX`. And patch your output voices into `pitchoutputX` and `gateoutputX`.

Here is an example for converting a three-fold polyphony

into a single voice. That voice is controlled by `01` and `02`:

```
[polytool]
pitchinput1 = _PITCH_1
pitchinput2 = _PITCH_2
pitchinput3 = _PITCH_3
gateinput1 = _GATE_1
gateinput2 = _GATE_2
gateinput3 = _GATE_3
pitchoutput1 = 01
gateoutput1 = 02
```

See how the parameter `voiceallocation` determines, which note should be played if there is more than one at a time.

The `polytool` can also do the opposite: You input a serial melody with just one note at a time and have that mapped to multiple output voices that make the actual

audible sound of the notes overlap. This can even be used to convert fast short arpeggios into chord pads.

The next example shows this. At `pitchinput1` and `gateinput1` there is a melody, for example from a sequencer or from the circuit `arpeggio` (see page 92). That is then played on two output voices:

```
[polytool]
pitchinput1 = _PITCH
gateinput1 = _GATE
pitchoutput1 = 01
pitchoutput2 = 02
gateoutput1 = 03
gateoutput2 = 04
```

Here the parameter `roundrobin` decides how the notes will be distributed onto the two output voices.

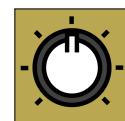
Input	Type	Default	Description
<code>pitchinput1</code> ... <code>pitchinput16</code>	?		The pitches of up to 16 input voices.
<code>gateinput1</code> ... <code>gateinput16</code>			The gates of up to 16 input voices.

Input	Type	Default	Description								
roundrobin		0	<p>Normally when looking for a free output for playing the next note, this circuit will start from pitchoutput1 in its search. This way, if there are not more notes than outputs at any time, the notes played first will always be played at the lowest numbered outputs. This leads to a deterministic behaviour when it comes to playing things like chords. The same voice will always be used for the first note in the stream of MIDI events.</p> <p>When you switch roundrobin to 1, this changes. Now the outputs are scanned in a round-robin fashion, like in a rotating switch. That way every output has the same chance to get a new note. Here it can even make sense to define multiple voices even if the track is monophonic. When you use envelopes with longer release times, you can transform such a melody into chords with simultaneous notes.</p> <p>Note: When all outputs are currently used by a note, roundrobin has no influence. Here voiceallocation selects which of the notes will be dropped.</p>								
voiceallocation	1 • 2 • 3	0	<p>When from the pitch inputs, at any given time, more voice are active than you have outputs assigned, normally the "oldest" notes would be cancelled. This behaviour can be configured here by setting voiceallocation to one of the following values:</p> <table border="1"> <tr> <td>0</td><td>The oldest note will be cancelled (default)</td></tr> <tr> <td>1</td><td>The new note will not be played and simply be omitted</td></tr> <tr> <td>2</td><td>The lowest note will be cancelled</td></tr> <tr> <td>3</td><td>The highest note will be cancelled</td></tr> </table>	0	The oldest note will be cancelled (default)	1	The new note will not be played and simply be omitted	2	The lowest note will be cancelled	3	The highest note will be cancelled
0	The oldest note will be cancelled (default)										
1	The new note will not be played and simply be omitted										
2	The lowest note will be cancelled										
3	The highest note will be cancelled										

Output	Type	Description
pitchoutput1 ... pitchoutput16	?	The pitches of up to 16 output voices.
gateoutput1 ... gateoutput16		The gates of up to 16 output voices.

13.46 pot - Helper circuit for pots

This circuit adds plenty of functionality to the controller pots in one circuit. It helps with various tasks. It replaces the former circuits **notchedpot** and **switchedpot** and these are also the main applications of **pot**: the simulation a precise center dent (notch) and the sharing of one pot for several different functions.



Convert a knob to bipolar output voltage

Let's start with some simple features. There are a couple of useful outputs, all of which you could do externally by use of some math. The following example converts a pot (which is ranging from 0 to 1) to a bipolar pot ranging from -1 to +1 (or -10 V to +10 V if you send it to an output):

```
[pot]
pot      = P1.1
bipolar  = 01 # Send -10V ... +10V to 01
```

Have a look into the table of jacks below about further useful things like splitting the pot's way in two halves.

Center notch

pot can simulate a potentiometer with a notch at the center. It helps to exactly select the center position by defining a "range of tolerance" that is considered to be the center. This range is called "notch" and is given in a percentage of the available range. I suggest using 10% so you don't lose too much pot resolution, but it's still easy enough to hit the center reliably. Here is an example:

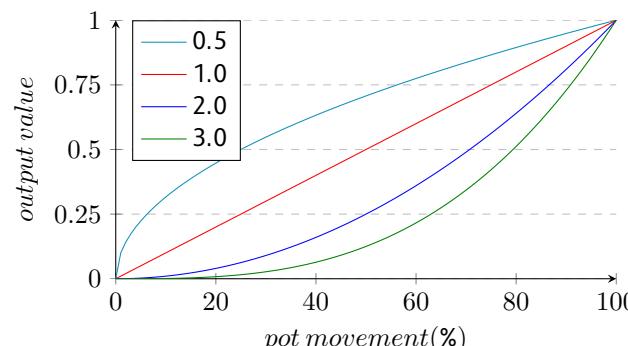
```
[pot]
pot      = P1.1
notch   = 10%
output   = _ACTIVITY

[algoquencer]
activity = _ACTIVITY
...
```

Slope

Sometimes you want a bit more resolution at the smaller values of the pot range. Maybe the pot controls a time from 0.0 to 1.0 seconds. And in the low range, say about 0.1 seconds, you need finer control.

You can change the slope of the pot in a way that either small values or values near 1.0 are "stretched out". The default is **slope = 1.0**. Look at the following diagram for the impact of different slope values:



As slope value of 0.0 does not make sense, because the pot would stick to 0.0 all the time, a minimum value of 0.001 is enforced.

If you are curious about the algorithm: This operation is just x^{slope} . So it's not "logarithmic" or "exponential" but polynomial.

Splitting the pot into two hemispheres

The jacks **lefthalf**, **righthalf**, **lefthalfinv** and **righthalfinv** allow you to split the pot in the middle into two ranges and use them for something completely different. Let's make an example:

```
[pot]
pot = P1.1
lefthalf = 01
righthalf = 02
```

Now let's start with the pot in the center position. Both outputs will be at **0.0**. If you now turn the pot to the left, just **lefthalf** (at **01**) is going to rise until it reaches **1.0** at the left end of the pot range. **righthalf** is staying at **0** all the time.

At the right half of the pot range, likewise **lefthalf** stays zero and **righthalf** will raise from **0** to **1**.

The jacks **lefthalfinv** and **righthalfinv** are similar, but are **1.0** in the neutral position in the center and fall to **0.0** at the edges.

Virtual pots

This circuit can handle so called "virtual pots". This is a situation where the physical position of the potentiometer does not match its output value. There are three situa-

tions where the **pot** circuit automatically switches to this virtual mode:

- When you share (overlay) pots using the **select** input
- When you enable presets (using **preset** or **Loadpreset**)
- When you send a trigger to **clear**

If course you can even use combinations of this: Overlay a pot with multiple functions, work with presets and set a start value at the same time.

If none of these three feature are used, there is not virtual pot and the physical position always counts.

In virtual mode, the last virtual value of the pot is always saved to the SD card and restored the next time your start your Droid.

In addition, in virtual mode the LED gauge is automatically activated. That displays the current virtual value of a pot using the 16 LEDs of the Droid master.

Sharing / overlaying pots

Potentiometers are valuable ressources and sooner or later you will run into a situation where you wish you had more pots. So you come up with the idea of using one pot for more than one function and switch between those with a button.

Previously **DROID** offered the circuit **switchedpot** for that task but that had certain limitations and also was not consistent with other circuits.

Let's make an example: Our task is to share pot **P1.1** so it sets *individual* release values for four different envelopes. First we need something to switch between these four.

We do this with a **buttongroup** (see page 108):

[p2b8]

```
[buttongroup]
    button1 = B1.1
    button2 = B1.2
    button3 = B1.3
    button4 = B1.4
    led1 = L1.1
    led2 = L1.2
    led3 = L1.3
    led4 = L1.4
```

Now at any given time, exactly one of the four buttons (i.e. their LEDs) is active. Now we add four **pot** circuits using the *same pot*. The trick is the **select** input. Each of these four should be selected just if one specific button is active. The output of each is being sent to one of the envelopes:

[pot]

```
    pot = P1.1
    select = L1.1
    output = _RELEASE1
```

[pot]

```
    pot = P1.1
    select = L1.2
    output = _RELEASE2
```

[pot]

```
    pot = P1.1
    select = L1.3
    output = _RELEASE3
```

[pot]

```
    pot = P1.1
    select = L1.4
    output = _RELEASE4
```

Finally we can add the four envelopes:

[contour]

```
    trigger = I1
    release = _RELEASE1
    output = 01
```

[contour]

```
    trigger = I2
    release = _RELEASE2
    output = 02
```

[contour]

```
    trigger = I3
    release = _RELEASE3
    output = 03
```

[contour]

```
    trigger = I4
    release = _RELEASE4
    output = 04
```

Now you can switch between the four envelopes with the buttons and use the pot to adjust the release time of the selected envelope.

Hints:

- Don't mix up **B1.1** and **L1.1**. If you would use **B1.1** for the switching, you would need to *hold* the button down while turning the knob. In which case you wouldn't need the **buttongroup** circuit.
- It is supported (and maybe useful) to select *several* of the "virtual" pots at the same time. In such a situation the turning of the real knob will adjust all of the selected values at the same time.
- Pots are no motorized faders. So they cannot show the current value correctly after switching. See below for details.
- In certain cases the **selectat** input might come handy: if you do the switching with *one* number

that changes, not a bunch of gate signals. See the jack table below for details.

Working with presets

The **pot** circuit supports up to 16 presets. With the use of the **preset** input you can select one of these. Set a number from **0** to **15** there to switch between presets. A change of that number immediately switches to another preset.

As an alternative you can work in a triggered mode by patching **loadpreset** and **savepreset** in addition. Switching presets happens just on these triggers. In triggered mode it's like have one more preset: the current "working" position of the pot.

On page 19 there is a whole chapter about presets. You find examples and more hints there.

Using a start value

A trigger to **clear** will set the virtual position of the pot to a defined start value (which you can adapt with **startvalue**). This means that now the physical position of the pot is not anymore identical with the virtual position. For that reason the pot runs in virtual mode as soon as you connect the **clear** input.

In virtual mode the state of the virtual pot is saved to the SD card, the pickup procedure (as described below) is applied and the LED gauge is active per default.

Picking up the pots

When you use overlaying, presets or a start value, your pots run in virtual mode. It means that the physical value of the pot might not be identical with its output value.

As an example let's assume that - using the upper example with overlaying - you first press **B1.1** and set decay fully CW **1.0**. Now you select **B1.2**. Because **0.5** is the start position of every virtual pot that is the current value of the second virtual pot. But the physical pot is at **1.0**.

This is solved in the following way:

- If you turn the physical pot *right*, the value of the virtual pot is always increased until both reach **1.0** at the same time.
- If the physical pot is already at **1.0** when you select a virtual pot, it cannot be increased further. You first have to turn the pot left a bit and then right again.
- If you turn the physical pot *left*, then the value of the virtual pot is always *decreased* until both reach **0.0** at the same time.
- If the physical pot is already at **0.0** when you select a virtual pot, it cannot be decreased further. You first have to turn the pot right and then left again.

If you really want even more details - here we go: Let's assume that the virtual pot is at **0.4** when you select it. And let's further assume that the physical pot is at position **0.8**. When you turn it *left*, the physical pot has a way of **0.8** to go until **0.0** and the virtual just **0.4**. So the virtual pot is moving with half of the speed, for both to reach **0.0** at the same time. When you turn the pot *right*, the virtual pot has **0.6** to go until maximum, while the physical pot has just **0.2** left until it reaches its maximum. So now the virtual pot moves three times faster than the physical.

This algorithm is different than the common "picking up" of pots that you see in Eurorack land quite a lot in such situations. I preferred my solution because it seems to be more convenient - especially if you want to change a value *a little bit*. Also it allows to have multiple virtual pots to be selected at the same time without having their values immediately snap to the same value.

By the way: it is also possible to select *none* of the pots. Which is a convenient way to reset the physical pot to the middle position so that you always have headroom for movement left *and* right, before selecting one of the virtual pots.

Input	Type	Default	Description
pot		0.0	Wire your pot here, e.g. P1.1
outputscale		1.0	The final output is multiplied with this value. It's a convenient method for scaling up and down the pot range.
notch		0.0	By setting this parameter to a positive number you create an artificial "notch" of that size. We suggest using 0.1 (or 10%). The maximum allowed value is 0.5 . Greater values will be reduced to that. Note: Using this in combination with outputscale also moves the notching point. E.g. with outputscale = 2 the notch will be at 1.0 .
discrete	 1 2 3		<p>Setting this value to 1 or larger switches the pot over to select a discrete integer number, rather than a continuous value. For example discrete = 5 makes the pot output one of the <i>exact</i> values 0, 1, 2, 3 or 4. This is ideal for selecting presets and similar. If you enable ledgauge (highly recommended), it shows you the value by using the LEDs of the master in an adapted way.</p> <p>The maximum allowed number is 16.</p> <p>When using discrete, the startvalue input is interpreted as a discrete number. So for example if you have discrete = 5, you can use startvalue = 3 to set the selected value to the number 3 after a trigger to clear. A potential outputscale is applied afterwards.</p> <p>Notes: The options notch and slope do not work in discrete mode. outputscale is still applied, though. All outputs other than output are dead and output 0.0. discrete = 1 does not really make sense, since there is just one value to select from and the output will always be 0.0.</p>
slope		1.0	Changes the resolution of the pot in lower or higher ranges. Set slope to 2 or more, if you want small values near 0.0 to be "zoomed in". Set slope to 0.5 or 0.3 if you want to zoom in value nears 1.0.

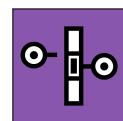
Input	Type	Default	Description												
<code>ledgauge</code>	 		<p>The “LED gauge” uses the 16 LEDs of the DROID master in order to indicate the current value of the pot. This is especially useful for “virtual” pots – i.e. those pots that you get when you use <code>select</code> in order to layer several different functions onto one pot. In that situation the position of the physical pot can be different than that of the virtual one, so the gauge shows you the effective virtual value.</p> <p>Furthermore, by illuminating the inner four LEDs, the gauge shows when the pot hits <i>exactly</i> 0.5. This can only happen if you use the <code>notch</code> parameter. Otherwise its practically impossible to hit exactly.</p> <p>The LED gauge is automatically activated if you use <code>select</code>. If you don’t like the LED gauge, you can turn it off with <code>ledgauge = off</code>. Otherwise <code>ledgauge</code> set’s the color of the indicator in the same way as the R-registers do and at the same time <i>enables</i> the gauge even if you don’t use <code>select</code>.</p> <p>Here are some color examples that you can use for the value of <code>ledgauge</code>:</p> <table border="1" data-bbox="819 627 1021 913"> <tr><td>0.2</td><td>magenta</td></tr> <tr><td>0.4</td><td>red</td></tr> <tr><td>0.6</td><td>yellow</td></tr> <tr><td>0.8</td><td>green</td></tr> <tr><td>1.0</td><td>cyan</td></tr> <tr><td>1.2</td><td>blue</td></tr> </table> <p>The colors repeat over in a kind of wheel at 1.2, so e.g. 1.4 creates the same color as 0.2.</p>	0.2	magenta	0.4	red	0.6	yellow	0.8	green	1.0	cyan	1.2	blue
0.2	magenta														
0.4	red														
0.6	yellow														
0.8	green														
1.0	cyan														
1.2	blue														
<code>startvalue</code>	 	0.5	<p>This parameter defines the value your pot will get when there is a trigger to <code>clear</code>. This is the value <i>before outputscale</i> is applied.</p> <p>If you use <code>discrete</code>, the parameter does not expect a fraction but a discrete number in the range of the discrete values (0, 1, 2, etc.).</p>												
<code>select</code>	 		<p>The <code>select</code> input allows you to overlay buttons and LEDs with multiple functions. If you use this input, the circuit will process the buttons and LEDs just as if <code>select</code> has a positive gate signal (usually you will select this to 1). Otherwise it won’t touch them so they can be used by another circuit. Note: even if the circuit is currently not selected, it will nevertheless work and process all its other inputs and its outputs (those that do not deal with buttons or LEDs) in a normal way.</p>												

Input	Type	Default	Description
selectat	1 o 2 o 3		This input makes the select input more flexible. Here you specify at which value select should select this circuit. E.g. if selectat is 0, the circuit will be active if select is <i>exactly</i> 0 instead of a positive gate signal. In some cases this is more convenient.
preset	1 o 2 o 3		This is the preset number to save or to load. Note: the first preset has the number 0, not 1! For the whole story on presets please refer to page 19. This circuit has 16 presets, so this number ranges from 0 to 15.
loadpreset			A trigger here loads a preset. As a speciality you can use the trigger for selecting a preset at the same time.
savepreset			A trigger here saves a preset.
clear			A trigger here loads the default start state into the circuit. The presets are not affected, unless you use direct preset switching with the preset input and without triggers. And that case the current preset is also cleared.
clearall			A trigger here loads the default start state into the circuit and into all of its presets.
dontsave		0	If you set this to 1, the state of the circuit will not be saved to the SD card and not loaded from the SD card when the Droid starts.

Output	Type	Description
output		Your pot output comes here.
bipolar		Optional output with a range from -1.0 to 1.0, where the center notch is at 0.0 (or from -outputscale to +outputscale if that is used).
absbipolar		A variation of bipolar that always outputs a positive value, i.e. the pot will go 1 ... 0.5 ... 0 ... 0.5 ... 1 (if outputscale is not used).
lefthalf		This output allows you to split the pot into two hemispheres. Here you get outputscale ... 0.0 while the pot is in the left half. In the middle and right of it you always get 0.
righthalf		This is the same but for the right half. It outputs 0 while the pot is in the left half and 0.0 ... outputscale from the middle to the fully right position.
lefthalfinv		This outputs 1.0 - lefthalf , i.e. the value range 0.0 ... 1.0 ... 1.0 when the pot moves left → mid → right (and the scaled by outputscale).
righthalfinv		This outputs 1.0 - righthalf , i.e. the value range 1.0 ... 1.0 ... 0.0 when the pot moves left → mid → right (and the scaled by outputscale).
onchange		This output emits a trigger whenever the pot is turned in either direction.

13.47 quantizer - Non-musical quantizer

This quantizer circuit is very simple. It reads an input voltage, quantizes it to the next discrete step that you configured and outputs it.



You can use it for musical purposes by setting the number of steps to 12 per Volt (which is default). It will quantize the input to semitones.

The following example scales down a pot P1.1 to 1 V (i.e. one octave) and then quantizes it to semitones. Since 12 is the default value for steps this parameter can be omitted:

ted here:

```
[quantizer]
    input = P1.1 * 1V
    output = 01
```

Note¹: In fact you can select 13 semitones here because if you turn the pot fully CW it will output 1, which will be scaled to 1 V and then quantized to 1 V - which is the 13th semitone above the lowest possible note.

Note²: if you are looking for a more musical quantizer then have a look at the Minifonion circuit.

You can use the Quantizer circuit as a sample & hold circuit if you set steps to 0 and use the trigger input:

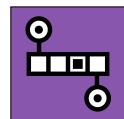
```
[quantizer]
    input = I1
    steps = 0
    trigger = I2
    output = 01
```

Input	Type	Default	Description
input		0.0	Patch the unquantized input voltage here
trigger			This jack is optional. If you patch it, the quantizer will work in triggered mode. Here the output pitch is always frozen until the next trigger happens.
steps	1 • 2 • 3	12	Number of steps that one Volt should be divided in. The default is 12 and will quantize the input voltage to semitones. The number of steps is related to a value of 1 V which means 0.1. It is allowed to use a fractional number here. E.g. the value 1.2 will quantize to 12 steps per 10 V (which means 12 steps per 1.0, which can make sense). A value of 0.0 (or lower) will basically mean an infinite number of steps and thus practically disable quantization.
bypass		0	If you set this gate input to 1 then quantization is bypassed and the input voltage is directly copied to the output.

Output	Type	Description
output		Here comes your quantized output voltage
changed		Whenever the quantization changes to a new output value a trigger with the duration 10 ms is output here. No trigger is output in bypass mode.

13.48 queue - Clocked CV shift register

This circuit implements a shift register (a queue) with 64 cells. Each cell contains one CV value. At each clock impulse the CVs each move one cell forwards. The last CV is dropped. And the current input value is copied to the first cell.



There are eight outputs, which you can place at any of the 64 cells you like. If you do not specify any placement, the outputs are placed at the first eight cells - and thus the information in the remaining 56 cells is not being used.

The following example reads CVs from the input **I1**. **04** always shows the CV value that was seen at the input four cycles previously:

```
[queue]
  input = I1
  clock = I2
  output4 = 04
```

The next example places three outputs at the positions **3**, **24** and **64**:

```
[queue]
  input = I1
  clock = I2
  outputpos1 = 3
  outputpos2 = 24
  outputpos3 = 64
```

```
output1 = 01
output2 = 02
output3 = 03
```

Please note:

- Since the DROID is very precise in processing CV voltages you can use the **queue** in order to delay melodies from sequencers etc.
- As always also the inputs **outputpos1** ... **outputpos8** may be CV controlled and change in time.

Input	Type	Default	Description
input		0.0	This CV will be pushed into the first cell of the shift register whenever a clock occurs.
clock			Each clock signal at this jack will move the CV content from every cell of the shift register to the next cell. The CV in the last cell will be dropped.
outputpos1 ... outputpos8			Specifies the position of each of the eight outputs - i.e. which cell of the shift register it should output. Allowed are values from 1 up to 64. These jacks default to 1 , 2 , ... 8 , so if you do not wire them the eight outputs reflect the first eight positions of the shift register.

Output	Type	Description
output1 ... output8		Eight outputs for eight different positions of the register. If you do not wire outputpos1 ... outputpos8 , these outputs show the content of the 1 st , 2 nd , ... 8 th cell.

13.49 random - Random number generator

A random number generator with clocked and unclocked mode, that can either create voltages at discrete steps and completely free values.



This circuit creates random numbers between two tunable levels **minimum** and **maximum**. In clocked mode each clock creates and holds a new random value. In unclocked mode the random values change at

the maximum possible speed (about 6000 times per second).

Simple example for clocked random numbers between **0.0** and **1.0** (**1.0** translates into 10 V at the output):

```
[random]
clock = I1
output = O1
minimum = 1V
maximum = 3V
```

Example for creating random output voltages between 1 V and 3 V:

```
[random]
clock = I1
output = O1
minimum = 1V
maximum = 3V
```

Input	Type	Default	Description
clock			Optional trigger: if this input is used then the output holds the current random number until the next clock impulse (sample & hold)
minimum		0.0	Minimum possible random number
maximum		1.0	Maximum possible random number
steps	1..2..3	0	Number of different voltage levels. If this is set to 0 (default), any voltage can appear, there is no limit. If this is 1, then there is no random any more since there is only one allowed step (which is the average between minimum and maximum). At 2 the only two possible output values are minimum and maximum . At 3 the possible levels are minimum , $\frac{\text{minimum} + \text{maximum}}{2}$ and maximum and so on...

Output	Type	Description
output		Output of the random number / voltage

13.50 recorder - Record and playback CVs und gates

Record and playback the movement of one CVs, eight gates and one integer numbers in the range 0 to 255, with permanent storage on the SD card.



Note: This circuit is still experimental. In a future firmware version it might be changed or removed. Also the file format on the SD card for the saved recordings might change and a new version might not be able to load old recordings.

Basic usage

The typical interface to the recorder is to use the three buttons "Record", "Play" and "Stop". The stop button is optional if you are low in buttons. Here is a simple example patch for recording a CV:

[p2b8]

```
[recorder]
  cvin = I1
  cout = 01
  recordbutton = B1.1
  playbutton = B1.2
  stopbutton = B1.3
  recordled = L1.1
  playled = L1.2
  stopled = L1.3
```

Now feed some CV into **I1**. The circuit starts in idle / stopped mode and **L1.1** is lit. In that mode the input is bypassed to the output, so that you can "hear" the effects of the CV at **I1**.

When you press the record button (**B1.1**), the recording starts and **L1.1** becomes lit. The input is still bypassed to

the output but at the same time written to the tape. Stop the recording either by pressing the stop button **B1.3** or record again.

Note: For your first experiments you might want to use the value of a pot as input CV. Then you can record your pot movements:

```
[recorder]
  cvin = P1.1 # record pot P1.1
```

You can now play the recording by hitting **B1.2**. The LED in that button is lit to indicate that the playback is going on. During playback the signal at **I1** is ignored and instead the tape's content is sent to **01**. The playback stops when the recording has played completely or when you hit the stop button. Hitting the play button during playback does not stop it but immediately restarts it from the beginning.

Sharing the three buttons with other circuits can be done with the **select** input – just as usual.

Pausing

The **pause** input allows you to pause the tape. This input is different from the three buttons as it does not expect a trigger but a gate (a state). You can use a **button** (see page 103) circuit for that:

[p2b8]

```
[button]
  button = B1.4
  led = L1.4
```

```
[recorder]
  cvin = I1
  cout = 01
  record = B1.1
  play = B1.2
  stop = B1.3
  recordled = L1.1
  playled = L1.2
  stopled = L1.3
  pause = L1.4
```

When you enable pause during playback, the playback is hold and the output sticks at the current CV. Disable pause to go on with the playback.

When you enable pause while recording, the tape stops and the input CV is no longer recorded. But you can resume the recording later by disabling pause.

Looping

The recorder has a simple loop function builtin. When you set the input **loop** to **1**, a playback immediately starts again when it's finished.

If looping is your main objective, please have a look at **cvlooper** (see page 133). That circuit has some very useful features for a real performance looper.

Playback speed

With the parameter **playbackspeed** you can alter the speed of the playback. The default value is **1**. A value of **2** doubles the speed speed. The fun part: you even can use

a negative speed for running the tape backwards. In that case a press to the play button starts the playback at the tape end.

The following example maps the speed to a pot that's scaled to a range from -5 to 5 (five times speed backwards to five times speed forwards). The center position sets the speed to 0 and stops the tape.

```
[recorder]
playbackspeed = P1.1 * 10 - 5
...
```

Scrubbing

Scrubbing is a special playback mode that's enabled by **scrub = 1**. During scrubbing no linear playback is done. Instead, you select a position on the tape with the input CV **scrubposition**. Example:

```
[button]
button = B1.5
led = L1.5
```

```
[recorder]
scrub = L1.5
scrubposition = P1.1
...
```

While the button **B1.5** is enabled, the **recorder** outputs the CV that's at the position that **P1.1** selects. The left position of the pot (or the value **0**) selects the start of the recording, the right position (**1**) the end.

While **scrub** is **1**, the current state (play, record, stop) of the recorder is ignored. It is in scrub mode. The **playled** output is **1**, the other LED outputs are **0**.

Trimming the start and end

The two inputs **trimstart** and **trimend** range from **0** to **1** and limit of the portion of the recording that is used for playback or scrubbing. For example **trimstart = 0.1** and **trimend = 0.8** disables the first 10% and the last 20% of the recording.

If you map the trimming positions to two pots you can manually select a portion. Just make sure that you start with the **timstart** pot fully left and **trimright** fully right:

```
[recorder]
trimstart = P1.1
trimend = P1.2
...
```

This limitation is not permanent. The recording itself is not modified by using trimming.

Recording gates and numbers

Along the CV, the recorder also records the state of up to eight input gates. You could record the output of a multi-track drum sequencer or even a manually tapped button pattern with that:

```
[recorder]
gatein1 = I1
gatein2 = I2
gatein3 = I3
gatein4 = I4
gateout1 = O1
gateout2 = O2
gateout3 = O3
gateout4 = O4
...
```

Other than **cvin** and **cfout** the gate tracks on the tape just distinguish between **0** and **1**.

In addition you can record one discrete integer number from **0** to **255**:

```
[recorder]
numberin = I1
numberout = O1
...
```

Other than with the CV, no linear interpolation is done. Every time the input number changes a new sample is created.

Applications for recording a number could be chord progressions or melodies that are represented by note numbers rather than pitch CVs.

Technical background and limitations

The two circuits **recorder** (see page 251) and **delay** (see page 138) are based on the same implementation of a virtual tape. This virtual tape has three tracks with three recording and playback heads:

1. One head for recording a continuous CV in the range $-1 \dots +1$ (which is $-10\text{ V} \dots 10\text{ V}$)
2. One head for recording eight gate tracks in parallel (CVs where just 0 and 1 is recorded)
3. One head for recording a discrete integer number in the range $0 \dots 255$

All these are recorded in parallel, so for example it's easy to record a CV/gate signal with just one **cvrecorder**. The discrete number is useful for recording the outputs of **buttongroup** (see page 108) circuits or the switches on the S10 similar things.

Note: The dynamich range of CV signal on the tape is just -1 ...+1 (or -10 V ...+10V). Any “too hot” signal is clipped to that range. The internal resolution of the CV is 16 bit (precisely: one Volt is divided in 3200 steps). If you need a larger range, you need to divide the input signal and multiply the output signal by some factor, but loose a bit precision that way.

The track with the eight gates records just **0** and **1**. Any other value will be squeezed into that format: values below 0.1 (1 V) are considered **0**, all other values **1**.

In order to use the RAM of the **DROID** as efficient as possible (and allow for many multiple instances of these circuits), the tape uses just 256 samples. Each time the state of one of the gates or the value of the number changes, a new sample is created. A change in the input CV is handled more intelligently as the CV values of the samples or *interpolated* linearly. The maximum error between the interpolated value and the actual stored CV is limited to 0.0001 (which is 1 mV).

If the input CV is more chaotic, however, the number of samples per time is limited to an average of one sample every 20 ms, while short periods with up to 10 samples without this limitations are allowed. This ensures that the minimum recordable tape length is $256 \times 20 \text{ ms}$, which is 5.12 seconds. Usually CVs are not so chaotic but either stepped or moving smoothly, so the recordings can be much longer.

If you have the special case of a stepped input CV - such as the output from a sequencer or from a CV/gate keyboard - you can switch to an alternative mode. Patch the gate output of the sequencer or keyboard into the **sample** input of the circuit. This enables the “triggered mode”. Here a new sample is just and only created at each positive gate edge of the **sample** input. So the recordings can be as long as 256 notes.

Note: That way you would loose the gate length, since the end of the gate does not trigger a new sample. Use the **gatetool** (see page [160](#)) with the **inputgate** and **outputedge** to get one trigger at each edge and feed that into **sample**.

Saving the tape to disk

The **recorder** does not support presets because of memory limitations. But you can save the current contents of the tape to your SD card. This is done by the two trigger inputs **save** and **load**, which are usually mapped to some buttons. Here is a simple example.

```
[recorder]
  save = B1.5
  load = B1.6
  ...
```

If you hit button **B1.5**, the file **tape0001.bin** is created on your SD card. Button **B1.6** loads that file into the circuit.

You can use any file number from **1** to **9999** by using the parameter **filenumber**. You might want to map that to a rotary switch of an S10:

```
[recorder]
  save = B1.5
  load = B1.6
  filenumber = S2.1
  ...
```

Note: Loading and saving is done in real time from/to your SD card. The files are very small, but the operation can take a small number of milliseconds. During that time

no circuit will do its job. And if your SD card is missing, things lag a bit more due to timeouts.

One important difference to presets is that these files can be share among circuits and even among different patches. A recording of the **recorer** circuit can be loaded with every **recorder** or delay circuit.

Input	Type	Default	Description						
playbutton			A trigger here starts or restarts the playback.						
recordbutton			A trigger here starts or stops the recording.						
stopbutton			A trigger here stops and ongoing playback or recording.						
loop		off	Set this to 1 to enable loop mode. In loop mode the playback is restarted immediately when it's finished.						
pause		off	While this is 1 , playback or recording is halted (the tape stops moving for the while).						
mode	1°2°3		Using this input is an alternative to using the three button inputs. If you patch mode , the three buttons (and LED outputs) are ignored. Instead you set the mode with this input:						
			<table border="1"> <tr> <td>0</td><td>Idle / stopped</td></tr> <tr> <td>1</td><td>Playback</td></tr> <tr> <td>2</td><td>Recording</td></tr> </table>	0	Idle / stopped	1	Playback	2	Recording
0	Idle / stopped								
1	Playback								
2	Recording								
			Since you set the mode from "outside", the recorder cannot switch it by itself. So if the mode is set to 1 (playback) and the playback is finished, it stays in playback mode and continues outputting the last sample.						
playbackspeed		1.0	Sets the speed of the tape during playback. 1 is normal speed, 0.5 half speed, 2 double speed, and so on. Negative speeds are allowed and move the tape backwards. The speed 0 stops the tape.						
scrub		off	If 1 enables scrubbing. Now the outputs reflect the tape position that is set with scrubposition .						
scrubposition		0.0	Position of the tape to play when scrubbing is enabled.						
trimstart		0.0	Omits a fraction of the recording at the beginning during playback and scrubbing. 0.1 omits the first 10%.						
trimend		1.0	Omits a fraction of the recording at the end during playback and scrubbing. 0.8 omits the last 20% (not 80%!).						
cvin		0.0	Continuous input CV						
numberin	1°2°3		Discrete input number in the range 0 ... 255						
gatein1 ... gatein8			Input gates						
clock			If you use this clock input, all time inputs are measured in clock ticks instead of seconds.						
sample			If you patch this input, "triggered" mode is enabled. In this mode, the virtual tape just records a new CV on each trigger at sample . So it just records stepped CVs, no slopes and no CV changes between the triggers.						

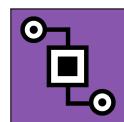
Input	Type	Default	Description
timewindow		0.0	<p>When in triggered mode, this optional parameter helps tackling a problem that many hardware sequencers show: often their pitch CV is not at its final destination value at the time their gate is being output. Often you see a very short "slew" ramp of say 5 ms after the gate. During that time the pitch CV moves from its former to the new value.</p> <p>Now if you trigger the cvtape circuit with the sequencer's gate you will essentially sample the <i>previous</i> pitch CV instead of the new one. Or maybe something in between.</p> <p>The timewindow parameter configures a short time window after the trigger to trigger. During that time period the tape will constantly adapt the last sample to a changed input CV. When that time is over, the input is finally frozen on the tape.</p> <p>The timewindow parameter is in seconds. So when you set timewindow to say 0.005 (which means 5 ms), you give the input CV 5 ms time for settling to its final value after a trigger to sample before freezing it.</p>
bypass		off	Setting bypass to on copies the input signals directly to the outputs, regardless of any other stuff going on.
save			Send a trigger here to save the current contents of the tape to a file on the SD card. The filename is tapeXXXX.bin , where XXXX is replaced by the number set by filenumber .
load			Send a trigger here to load a previously saved file into the tape. Use filenumber so specify which file to load.
filenumber	1..2..3	1	Number of the file to load or save. The range is 0 - 9999. If filenumber is 123, the name on the SD card is tape0123.bin . These files are shared between all recorder and delay circuits.
select			The select input allows you to overlay buttons and LEDs with multiple functions. If you use this input, the circuit will process the buttons and LEDs just as if select has a positive gate signal (usually you will select this to 1). Otherwise it won't touch them so they can be used by another circuit. Note: even if the circuit is currently not selected, it will nevertheless work and process all its other inputs and its outputs (those that do not deal with buttons or LEDs) in a normal way.
selectat	1..2..3		This input makes the select input more flexible. Here you specify at which value select should select this circuit. E.g. if selectat is 0 , the circuit will be active if select is <i>exactly</i> 0 instead of a positive gate signal. In some cases this is more convenient.

Output	Type	Description
recordled		Is 1 during recordings.
playled		Is 1 during playback or scrubbing.
stopled		Is 1 when no playback, recording or scrubbing is going on.

Output	Type	Description
cvout		Output of the continous input CV
numberout	1 ◊ 2 ◊ 3	Output of the discrete number
gateout1 ... gateout8		Output of the gates
overflow		When the internal memory of the tape is exceeded and data got lost, this gate goes to 1 for 0.5 seconds. If you are suspecting this situation, you can wire this output to an LED and observe the memory status that way.

13.51 sample - Sample & Hold Circuit

This is a simple sample & hold circuit. Each time a positive trigger is seen at the jack **sample** a new value is sampled from **input** and sent to the **output**.



Example:

[sample]
 input = I1

sample = I2
output = 01

Input	Type	Default	Description
input		0.0	Input signal to be sampled
sample			A positive trigger here will read the current value from input and store it internally.
gate			This is an alternative way of making the circuit take a sample from the input. Here it is sampling all the time while the gate is high. In that way it is a bit like bypass . But as soon as the gate goes low again, the output sticks to the last sample value just before that.
timewindow		0.0	<p>This optional parameter helps tackling a problem that many (non-analog) sequencers show: often their pitch CV is not at its final destination value at the time their gate is being output. Often you see a very short "slew" ramp of say 5 ms after the gate. During that time the pitch CV moves from its former to the new value.</p> <p>Now if you trigger the sample circuit with the sequencer's gate you will essentially sample the <i>previous</i> pitch CV instead of the new one. Or maybe something in between.</p> <p>Now the timewindow parameter introduces a short time window after the sample trigger. During that time period the sample & hold circuit will constantly adapt to a changed input CV (is essentially in bypass mode). When that time is over, the input is finally frozen.</p> <p>The timewindow parameter is in seconds. So when you set timewindow to say 0.005 (which means 5 ms), you give the input CV 5 ms time for settling to its final value after a trigger to sample before freezing it.</p>
bypass			While this gate input is high, the circuit is bypassed and input is copied to output .

Output	Type	Description
output		The most recently sampled value is sent here.

13.52 select - Copy a signal if selected

Copies a value just when the circuit is selected via **select**.

This solves the problem of having an LED displaying something, but just when a certain “menu page” or similar is active. Simply setting the LED with **copy** (see page 131) or some other circuit’s output will *always* set it. Checking some



select state and sending **0** does not help, since it will override any other circuit’s values for the LED even when those are selected.

Here is an example of letting the LED L1.1 flash when **_SELECTED** is high, and otherwise **don’t copy anything** to the LED:

```
[lfo]
output = _FLASH
```

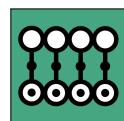
```
[select]
select = _SELECT
input = _FLASH
output = L1.1
```

Input	Type	Default	Description
input		0.0	Connect the signal you want to copy.
select			The select input allows you to overlay buttons and LEDs with multiple functions. If you use this input, the circuit will process the buttons and LEDs just as if select has a positive gate signal (usually you will select this to 1). Otherwise it won’t touch them so they can be used by another circuit. Note: even if the circuit is currently not selected, it will nevertheless work and process all its other inputs and its outputs (those that do not deal with buttons or LEDs) in a normal way.
selectat			This input makes the select input more flexible. Here you specify at which value select should select this circuit. E.g. if selectat is 0 , the circuit will be active if select is <i>exactly</i> 0 instead of a positive gate signal. In some cases this is more convenient.

Output	Type	Description
output		The input will be copied here, but just when the circuit is selected via select .

13.53 sequencer - Simple eight step sequencer

This circuit implements a sequencer that is a bit similar to the widely known Metropolis sequencer by Intellijel. It lacks a couple of its features - but most of these can be patched externally by use of other circuits.



On the other hand it is not limited to 8 stages since you can chain multiple instances of this sequencer together to form one large sequencer very easily.

Since *everything* in the **DROID** is controllable via CV, of course pitch and gate signals are included, which makes the circuit much more versatile than it may seem at a first look.

Here is a small example of a CV sequencer that is playing four voltages in a turn (it needs a clock into **I1**):

```
[sequencer]
  clock      = I1
  pitchoutput = 01
  pitch1     = 1V
  pitch2     = 3.5V
  pitch3     = 8V
  pitch4     = -2V
```

If you set the **outputscale** parameter to $\frac{1}{12}$ V (which is the same as the number $\frac{1}{120}$, you can specify pitches directly in *semitones*:

```
[sequencer]
  clock      = I1
  pitchoutput = 01
  outputscale = 1/120
  pitch1     = 0
  pitch2     = 12
  pitch3     = 10
  pitch4     = 7
```

```
pitch5      = 5
pitch6      = 3
pitch7      = 5
pitch8      = 7
```

The following example uses four expander buttons for turning the steps on or off and four pots, which are scaled down to a range of 0V ... 3V.

```
[p2b8]
[p2b8]

[lfo]
  hz = 4
  square = _CLOCK

[button]
  button = B1.1
  led    = L1.1

[button]
  button = B1.2
  led    = L1.2

[button]
  button = B1.3
  led    = L1.3

[button]
  button = B1.4
  led    = L1.4

[sequencer]
  clock      = _CLOCK
  pitchoutput = 01
  gateoutput = 02
  pitch1     = P1.1 * 3V
  pitch2     = P1.2 * 3V
```

```
pitch3      = P2.1 * 3V
pitch4      = P2.2 * 3V
gate1       = L1.1
gate2       = L1.2
gate3       = L1.3
gate4       = L1.4
```

Note: the pitch values you dial in with the pots are not quantized, so it's a bit hard to hit a musical pitch. Please have a look at the circuits **quantizer** (page 248) and **minifonion** (page 200) for how to quantize pitch values.

Making longer sequences

The **sequencer** circuit is limited to 8 steps. But: you can easily chain a large number of these circuits together to form longer sequences. This is super easy. Just set the jack **chaintonext** to **1** and place another **sequencer** circuit with more steps after that. Here is an example for a 12 step sequencer:

```
[p2b8]

[lfo]
  hz = P1.1 * 30
  output = _CLOCK

[sequencer]
  clock = _CLOCK
  reset = B1.1
  pitchoutput = 01
  gateoutput = 02
  outputscale = 1/120
  pitch1 = 1
  pitch2 = 8
  pitch3 = 13
```

```

pitch4 = 25
pitch5 = 4
pitch6 = 11
pitch7 = 7
pitch8 = 21
chaintonext = 1 # continue at next sequencer

[sequencer]
pitch1 = 2
pitch2 = 9
pitch3 = 14
pitch4 = 26

```

You can make the chain longer by adding more **sequencer** circuits. All but the last must have **chaintonext** set to **1**. Here comes a 19 step sequencer:

```

[p2b8]

[lfo]
hz = P1.1 * 30
output = _CLOCK

```

```

[sequencer]
clock = _CLOCK
reset = B1.1
pitchoutput = 01
gateoutput = 02
outputscaled = 1/120
pitch1 = 1
pitch2 = 8
pitch3 = 13
pitch4 = 25
pitch5 = 4
pitch6 = 11
pitch7 = 7
pitch8 = 21
chaintonext = 1 # continue at next sequencer

```

```

[sequencer]
pitch1 = 2

```

```

pitch2 = 9
pitch3 = 14
pitch4 = 26
pitch5 = 2
pitch6 = 9
pitch7 = 14
pitch8 = 26
chaintonext = 1 # continue at next sequencer

```

```

[sequencer]
pitch1 = 3
pitch2 = 10
pitch3 = 15

```

Notes:

- Define all the input and output jacks like **clock**, **pitchoutput** etc. just for the first sequencer. All subsequent ones just have **pitch**, **gate**, **repeat**, **slew** and **cv** definitions.
- The parameter **chaintonext** is *dynamic*. You could make or break the chain with a toggle **button** or something else if you like.

Input	Type	Default	Description
clock			Each trigger into this jack advances the sequence by one step.
reset			A trigger here resets the sequence to the first step
stages	1..2..3		Number of inputs of pitch... , gate... , slew... , cv and repeats that should be used. If you set stages to a number higher than the number of used inputs, all inputs will be used. If you omit this parameter, all used inputs will be used.
steps	1..2..3	0	With this input you can force the sequencer to begin from start after a certain number of clock cycles. If you omit the parameter or if it is set to 0, the sequencer will play all stages with all repeats until it resets to the beginning.
transpose		0.0	This voltage is added to the pitch output.
outputsscaling		1.0	The output pitch is multiplied by this parameter.
gatelen	0..1		The length of the output gates. If it is unpatched, the original input clock is fed through 1:1 (with its own duty cycle). When used, it is a ratio from 0.0 to 1.0 and relative to the cycle of the input clock. Setting the gatelen to 1.0 merges two adjacent gates together since there is not time left for a low gate before the next step begins.
pitch1 ... pitch8		0.0	These are the pitches of the various steps. You can put fixed numbers here but also of course pots or variable inputs. Note: The number of <i>used</i> input jacks defines the length of the sequence, unless you override that with stages .
cv1 ... cv8		0.0	Each step has an optional CV assigned. You can use that CV for modulating something or even outputting a second pitch information.
gate1 ... gate8		1	The gate inputs should be 0 (off) or 1 (on) . For stages with a 0-gate no output gate is produced and the pitch information is kept at the previous state. Unpatched gates are considered to be on!
slew1 ... slew8		0.0	Enables slew limiting for that stage. The input is not binary but you can set the amount of slew here – individually for each step. 0.0 switches the slew off, higher values create slower slews.
repeat1 ... repeat8		1.0	Set this to a positive integer number like 1 , 2 , and so on. It sets the number of times this stage should be repeated until the next stage will be approached. It is currently not allowed to have 0 repeats – although this would make sense in a future version.
chaintonext			If you set this input to 1 , the next sequencer circuit's pitch and other step inputs will be added to this sequencer. See the general circuit notes for details.

Output	Type	Description
pitchoutput		The pitch output. It is unquantized.
cvoutput		The optional CV output, in case you use the cv1 ... cv8 inputs.

Output	Type	Description
gateoutput		The gate output.

13.54 slew - Slew limiter

This is a CV controllable slew limiter for CVs. Special about it is that it implements three alternative algorithms. The traditional exponential algorithm (as is commonly implemented in analog circuits), a linear algorithm and a special S-shaped curve.

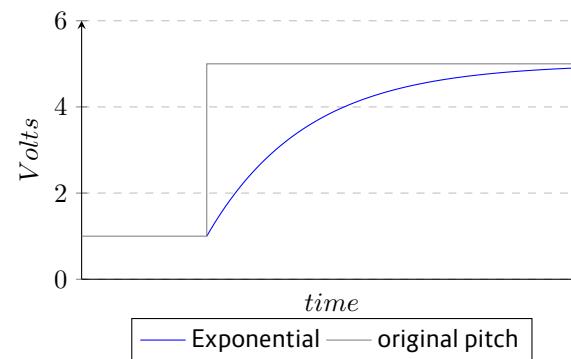


Here is a simple example for a slew limiting on **I1** → **01** which is controlled with the pot **P1.1**:

```
[slew]
  input      = I1
  slew       = P1.1
  exponential = 01
```

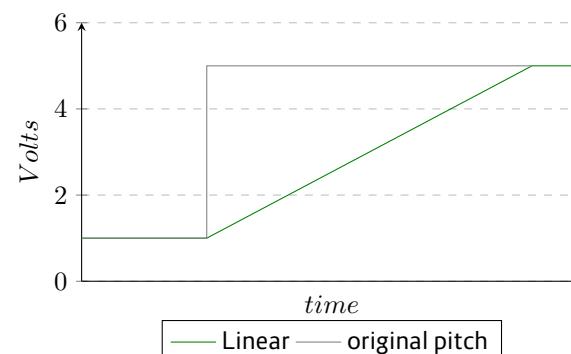
Exponential shape

This is the “classical” slew limit shape, which originates from the (negative) exponential loading current of a capacitor. It is also the shape of a low pass filter that is used for slew limiting. The slope is proportional to the distance between the current and the target voltage. Or in other words the voltage changes fast at the beginning and slower at the end:



Linear shape

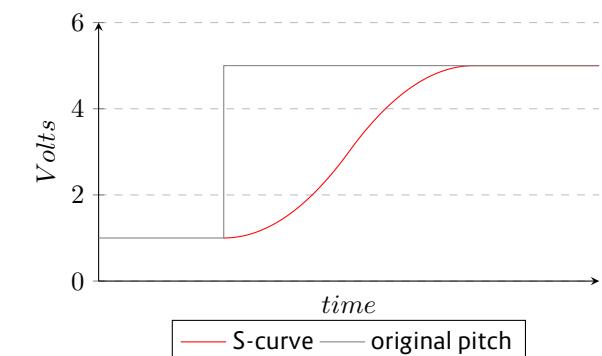
The *linear* algorithm simply limits the voltage change per time to a certain change rate, e.g. to 10 V per second. If the input voltage changes faster (for example suddenly jumps up), the output voltage follows that with that maximum rate. At a pot position of **0.5** the maximum slew is 120 V per second.



S-Curve shape

The S-curve – when applied to pitches – sounds different than an exponential curve since it more reflects the way e.g. a trombone player accelerates and deaccelerates his arm in order to move to another pitch. In our algorithm we assume that in the first half of the time the arm accelerates at a constant rate (which is controlled by the **slew** parameter) and at the second half of the time it deaccelerates (again at that rate, just negative), until it exactly reaches the target pitch.

There is one audible difference to a real trombone player, however. The real musician would start to move his arm *before* the new note begins, in order to be at the target position right in time. But here the movement is initiated by the pitch change it self so it is delayed by the slew limiting.



Input	Type	Default	Description
input			Wire the CV that you wish to slew limit here.
slew		1.0	This controls the slew rate. A value of 0.0 disables slew limiting. The output immediately follows the input without any delay. A value of for example 2.0 in linear mode means that 2.0 seconds are needed for a change of 1 V (which is a value of 0.1 or one octave if used as pitch). In the other two modes the slew time is tuned to sound similar. Negative values of this parameter are treated as 0.0 .
slewup		1.0	This allows a special handling when the voltage moves <i>upwards</i> . The slew limiting for upwards is slew multiplied with slewup . Since slew defaults to 1.0 you can just use slewup and slewdown if you want to control both directions separately.
slewdown		1.0	Sets the slew rate for downwards movement.
gate			If this jack is patched, the slew limiting is only active while this gate is high. Otherwise it's like setting the slew parameter to zero.

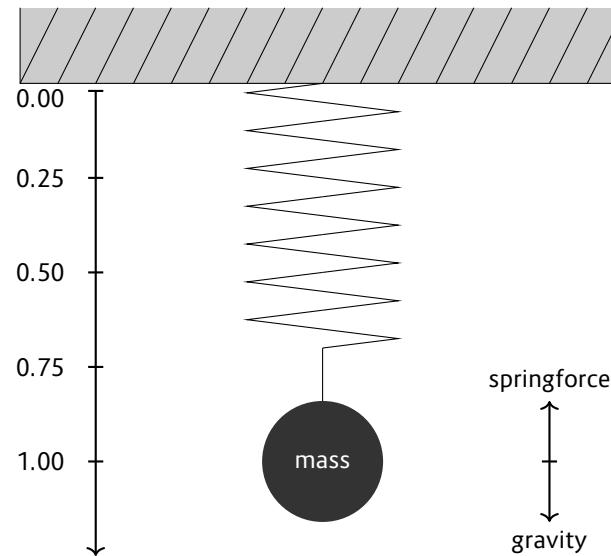
Output	Type	Description
exponential		Output for the resulting CV with the exponential (classical) slew algorithm applied
linear		Output for linear slew limiting
scurve		Output with the slew limitation according to the S-curve algorithm.

13.55 spring - Physical spring simulation

A physical simulation of a mass hanging from on an ideal spring which can create interesting "bouncing" CV sources.



Consider the following drawing:



Without any further parameters the mass starts at position **0.00** and velocity **0.00** and is accelerating downwards until the force of the spring equals the gravity. At this point it decelerates until the velocity is zero. Now the mass is being accelerated *upwards* until it reaches the top position at **0.00** again. This results, in essence, to a damped sine wave.

The **position** and **velocity** are available at their respective outputs ready to be used for modulation.

```
[spring]  
position = 01  
velocity = 02
```

Now, this could be done more easily with the LFO circuit (see page 163). But it's getting interesting when you look at the other parameters and the modulation possibilities. Please look at the table of jacks for details.

Friction

Per default the motion is without any friction and thus the mass will move up and down forever. You can apply two different types of friction. **flowresistance** is the type of friction a body has in a liquid or gas. Its force is relative to its velocity. Whereas the normal **friction** force is constant.

When you use any type of friction, the spring will finally stop swinging. You need to either *shove* it from time to time or reset it to its start with the **reset** trigger input.

The following example will create a slowly decaying sine wave, which is restarted whenever a trigger is sent to **reset**:

```
[spring]  
flowresistance = 0.5  
reset = I1  
position = 01  
velocity = 02
```

Shoving

You also can *shove* the mass downwards or upwards. As long as you send a gate signal into **shove** the mass will be shoved downwards. The exact force can be set with **shoveforce** and defaults to being the same as the gravity. A negative value will lift the mass upwards.

Setting **shove** to a constant **1** value will steadily apply **shoveforce**, which can be interesting as that is itself a changing CV (some LFO, feedback loop or whatever).

The physical model

Please note that the physical model is normalized in a way such that every parameter is 1. For example the mass is 1kg and the gravity is $1\frac{\text{N}}{\text{kg}}$. The force of the spring is $1\frac{\text{N}}{\text{m}}$.

In order to avoid anomalies or infinities, the velocity of the mass is limited to $\pm 10\frac{\text{m}}{\text{s}}$ and the position is limited to the range of $\pm 10\text{ m}$.

Input	Type	Default	Description
mass		1.0	The mass of the object on the spring. The heavier it is, the farther the spring will move up and down.
gravity		1.0	The gravity of the simulated planet the spring is mounted at. If you set the gravity to zero, the mass will move exactly around the zero position from positive to negative and back. But you need to shove it or set a start position other than 0, in order to get it started.
springforce		1.0	The force of the string per m it is stretched. In an ideal spring the force is proportional to the current elongation.
flowresistance		0.0	Setting this to a value > 0 will dampen the oscillation in a way, that higher velocities will be damped more then slower ones. This means that impact of the friction will get less and less as time goes by and the movement slows down.
friction		0.0	Setting this to a value > 0 will also dampen the oscillation, but in a way that is independent of the current speed of the mass.
speed		0.0	This parameter speeds up or slows down the perceived time. It works on a 1V/Oct base. Every positive 1V (or 0.1) doubles the speed. So if you set speed to 2V or 0.2 it will speed up the movement by a factor of 4. An input of -1V will slow down the movement to the half.
shove		0	While this gate input is logical 1, an extra force of 1 N is applied to the mass pointing downwards. You can change that force with shoveforce .
shoveforce		1.0	This is the force being applied to the mass while shove is active
reset			Resets the whole system to its start position.
startvelocity		0.0	Sets the velocity the mass has which DROID starts of a reset is triggered
startposition		0.0	Sets the position the spring has which DROID starts of a reset is triggered

Output	Type	Description
velocity		Outputs the current velocity of the mass
position		Output the current length of the string. If the string goes upwards (which is possible with certain modulations), this can be negative.

13.56 superjust - Perfect intonation of up to eight voices

This circuit automatically creates a perfect pure intonation for up to eight input pitches.



Introduction

This means that all pitches are in just intervals, which correspond to small whole number ratios such as $\frac{3}{2}$ or $\frac{5}{4}$. Assuming that you have perfectly tuned and calibrated VCOs, If these pitches are used to play a chord, there will be no or just minimal audible beatings and the chord will sound very pure.

In normal equal temperament intonation all intervals are a multiple of $\sqrt[12]{2}$ and thus there is no pure interval at all, with the exception of the octave. So all chords will sound impure.

The problem about pure or just intonation is, that you need to decide for just one scale, e.g. C major, and then tune all 12 notes in a way that chords from that scale sound good. But as soon as you change the scale, the intervals will sound ugly.

What makes the **superjust** unique is that fact, that it automatically creates a pure intonation in a *dynamic* way. It constantly “listens” to the notes that are *currently* being played and creates a perfect intonation just for those, not for a scale or so. As soon as at least one note changes, all notes are retuned in order to find a new perfect tuning. This is a bit like a well-trained string ensemble or choir, where each musician listens and adjusts his or her pitch in relation to all others.

Usage

The nice thing is: you don't need any configuration. You need not specify any information about the root note, the scale or anything else. Neither need the inputs be quantized so some scale or tuned to 440 Hz. The circuit will simply analyse all input pitches, apply its algorithm (patent pending) and then just slightly raises or lowers each note so that at the end each pair of frequencies have a rational oscillation ratio with small numerator and denominator. This is done in a way that the average pitch does not change. Just pipe your pitches through that circuit and you are done. And if you want to use a quantizer, use **superjust** *after* quantization.

Here an example for three voices:

```
[superjust]
  input1 = I1
  input2 = I2
  input3 = I3
  output1 = O1
  output2 = O2
  output3 = O3
```

Tuning

Of course, an exact tuning of your VCOs is crucial, since the pitch differences between a normal tempered intonation and a perfect intonation are quite small. The circuit helps you in the process of tuning with the inputs **tuningmode**, which you can map to a toggle button:

```
[button]
  button = B1.1
```

```
led = L1.1

[superjust]
  input1 = I1
  input2 = I2
  input3 = I3
  output1 = O1
  output2 = O2
  output3 = O3
  tuningmode = L1.1
```

Now when the button **B1.1** is active, all outputs will output zero volts. Tuning with 0 V is not optimal in some cases. You should tune your VCOs always roughly in the average pitch you play them. So you can set the tuning voltage with the parameter **tuningpitch**. Here it is set to 2 V (2 octaves higher than 0 V):

```
[button]
  button = B1.1
  led = L1.1
```

```
[superjust]
  input1 = I1
  input2 = I2
  input3 = I3
  output1 = O1
  output2 = O2
  output3 = O3
  tuningmode = L1.1
  tuningpitch = 2V
```

Sometimes it is desirable to change the tuning pitch to other octaves on the fly. This example uses pot **P1.1** for going through several octaves, and uses a quantizer for creating steps of 1 V each:

```

[button]
  button = B1.1
  led = L1.1

[quantizer]
  input = P1.1
  steps = 1 # 1 step per octave
  output = _TUNINGPITCH

[superjust]
  input1 = I1
  input2 = I2
  input3 = I3
  output1 = 01
  output2 = 02
  output3 = 03
  tuningmode = L1.1
  tuningpitch = _TUNINGPITCH

```

Perfect VCO calibration

If you *really* want to eliminate all beatings in your chords while using analog VCOs, you probably need something to correct tracking deviations. Here I strongly recommend using the circuit **calibrator** (see page 111). Here is an example with three voices, where buttons of a P2B8 are used for fine tuning the VCO tracking in each octave:

```

[superjust]
  input1 = I1
  input2 = I2
  input3 = I3
  output1 = _01
  output2 = _02
  output3 = _03

```

[calibrator]

```

input = _01
output = 01
nudgeup = B1.1
nudgedown = B1.3

```

```

[calibrator]
  input = _02
  output = 02
  nudgeup = B1.2
  nudgedown = B1.4

```

```

[calibrator]
  input = _03
  output = 03
  nudgeup = B1.5
  nudgedown = B1.7

```

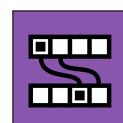
The number of pitch inputs and pitch outputs you patch should be identical.

Input	Type	Default	Description
input1 ... input8	 $\frac{1V}{Oct}$		1 st ... 8 th pitch input
tuningmode		0	While this is 1 , all outputs output the value set by tuningpitch . This is for tuning all outputs. Since perfect tuning is crucial for perfect intonation, this is quite useful.
tuningpitch	 $\frac{1V}{Oct}$	0V	This pitch CV will be output while the tuning mode is active.
bypass		0	While this is 1 , all inputs are passed through to the outputs without changes.
transpose	 $\frac{1V}{Oct}$	0V	This value is being added to all outputs, but not in tuning or bypass mode. It can e.g. be used for making a vibrato on a chord.

Output	Type	Description
output1 ... output8	 $\frac{1V}{Oct}$	1 st ... 8 th pitch output

13.57 switch - Adressable/clockable switch

This circuit supports a set of various switching operations. It can switch several inputs to one output either by means of addressing the input via CV or by stepping forward and backward. You can do the same vice versa: connecting one input to one of several outputs while setting the inactive outputs to 0 V.



You can even use several inputs *and* outputs at the same time and thus create an $n \times m$ switch with the option of rotating the outputs against the inputs by means of addressing or stepping.

At minimum you need to patch two inputs and one output (or vice versa), plus a switch like **forward**, **backward** or **offset**.

The first example switches four inputs **I1** ... **I4** to one output **01** by means of a trigger at **forward**. At the beginning **I1** is wired to **01**. Each time a trigger is seen at **forward** the switch switches to the next input and at the end starts over at **I1** again. So it cycles through **I1** → **I2** → **I3** → **I4** → **I1**:

```
[switch]
  input1 = I1
  input2 = I2
  input3 = I3
  input4 = I4
  output = 01
  forward = I8
```

Please note, that **output** and **output1** are synonyms here. You can use either way you like. Just the same is **input** just a shorthand for **input1**.

Now Let's do the opposite thing: distribute one input to four different outputs:

```
[switch]
  input = I1
  output1 = 01
  output2 = 02
  output3 = 03
  output4 = 04
  forward = I8
```

Now, if you try this out, you might notice that a trigger to **forward** moves the selected output *backwards!* This is no bug but very logical. The reason will get more clear if we build a switch with several inputs *and* outputs. Let's make a 3×3 switch:

```
[switch]
  input1 = I1
  input2 = I2
  input3 = I3
  output1 = 01
  output2 = 02
  output3 = 03
  forward = I8
```

Now a trigger to **forward** moves each output forward to the next input. That is the same as saying each input moves *backward* to the previous output. Of course you can change the direction by using **backward** instead of **forward**.

Instead of moving the switch with a trigger you also can *address* it by using a CV at the input **offset**. In this example we use a steady CV being either 0 (for selecting **01**) or 1 (10 V) for selecting **02**:

```
[switch]
  input = I1
  output1 = 01
```

```
  output2 = 02
  offset = I7
```

Using two inputs and two outputs creates a switch that can swap these two. Here with offset 0 **input1** is connected to **output1** and **input2** to **output2**. If **offset** is 1, **input1** will be connected to **output2** and **input2** to **output1**.

```
[switch]
  input1 = I1
  input2 = I2
  output1 = 01
  output2 = 02
  offset = I7
```

Now let's make another example for a CV addressable switch. The CV is read from **I7**. At a voltage of 0 V **output1** is connected to **input1**, at 1 V to **input2**, at 2 V to **input3**, at 3 V to **input4**, at 4 V to **input1** again, at 5 V to **input2** and so on:

```
[switch]
  input1 = I1
  input2 = I2
  input3 = I3
  input4 = I4
  output1 = 01
  offset = I7 * 10 # 1 V per switch step
```

Generally speaking, if you connect less inputs than outputs, the unconnected inputs are regarded as getting a 0 V input. If you connect less outputs then inputs, the unconnected outputs send their values into the black horrible void.

Input	Type	Default	Description
<code>input1 ... input16</code>		<code>0.0</code>	1 st ... 16 th input. Use these inputs in order and don't leave gaps.
<code>forward</code>			If a trigger or gate is received here, the switch adds one to the current internal switch offset. So every output moves to the next input and every input moves to the previous output.
<code>backward</code>			Similar then <code>forward</code> , but switches backwards
<code>reset</code>			Resets the switch to its initial position. Assuming <code>offset</code> is at <code>0</code> , <code>input1</code> is connected to <code>output1</code> , <code>input2</code> to <code>output2</code> etc. If <code>reset</code> and a trigger at <code>forward / backward</code> happen at the same time (within 5 ms), the reset will win and the switch is being reset to offset 0. This avoids problems with unprecise timing of external sequencers.
<code>offset</code>	<code>1°2°3</code>	<code>0</code>	This allows CV addressable switching. The number read here is being used a shifting offset and is always added to the internal offset. For example if you send 5 here, it is like you have triggered <code>forward</code> five times after the last reset. Please note, then 5 would mean 50 Volts, not 5 Volts. So if you patch an external CV like <code>I1</code> here, you probably want to multiply with some useful number.

Output	Type	Description
<code>output1 ... output16</code>		1 st ... 16 th output. Use these outputs in order and don't leave gaps.

13.58 switchedpot - Overlay pot with multiple functions (OBSOLETE)

This circuit has been superseded by the new circuit pot (see page 242). pot can do all switchedpot can do and much more. switchedpot will be removed soon.



This circuit allows you to use one of your potentiometers on your controllers for up to eight different functions. It is like creating up to eight *virtual* pots. With the inputs **switch1** ... **switch8** you select, which of these virtual pots are currently active. When you turn the (physical) pot, all active virtual pots are being changed.

The values of all virtual pots start at center position (**0.5**).

The current values of all virtual pots are saved in the DROID's internal flash memory, so next time you power on you have all settings of the virtual pots reserved.

Here is an example, where one pot is used to control both decay and release of an envelope.

```
[switchedpot]
pot      = P1.1
switch1 = B1.1
switch2 = B1.2
output1 = _DECAY
output2 = _RELEASE
```

```
[contour]
gate    = I1
decay   = _DECAY
release = _RELEASE
output  = 01
```

Now - while you press *and hold* button **B1.1** and turn the knob, the decay parameter will change. Holding **B1.2** will change release. Holding *both* at the same time is also

possible and will change decay and release at the same time.

Hints:

- If you do not like to hold the buttons then you might want to use the **button** circuit for converting the buttons into toggle buttons.
- If you want one button per function and want always one pot to be selected, you can use the **buttongroup** circuit for combining the buttons into a group.

Picking up the pots

Pots are no encoders. So when reusing a pot for more than one function at a time there is always the problem that when you switch to one pot function the pot probably currently is not set to the current value of the function. As an example let's assume that - using the upper example - you first press **B1.1** and set decay fully CW **1.0**. Now you select release. Because **0.5** is the start position of every virtual pot that is the current value of release. But the physical pot is at **1.0**.

DROID solves this in the following way:

- If you turn the physical pot *right*, then the value of the virtual pot is always increased until both pots reach **1.0** at the same time.
- If the physical pot is already at **1.0** when you select a virtual pot, it cannot be increased further. You first have to turn the pot left a bit and then right again.
- If you turn the physical pot *left*, then the value of the virtual pot is always *decreased* until both pots

reach **0.0** at the same time.

- If the physical pot is already at **0.0** when you select a virtual pot, it cannot be decreased further. You first have to turn the pot right a bit and then left again.

Let's assume that the virtual pot is at **0.4** when you select it. And let's further assume that the physical pot is at position **0.8**. When you turn it *left* the physical pot as a way of **0.8** go until **0.0** and the virtual just **0.4**. So the virtual pot is moving with half of the speed, so that both reach **0.0** at the same time. When you turn the pot *right*, on the other hand, the virtual pot has **0.6** to go until maximum while the physical pot has just **0.2** left until it reaches its maximum. So now the virtual pot moves three times faster than the physical.

This algorithm is different than the common "picking up" up pots that you see in Eurorack land quite a lot in such situations. We preferred our solution over that because it seems to be more convenient - especially if you just want to change a value just a little bit. Also it allows to have multiple virtual pots to be selected at the same time.

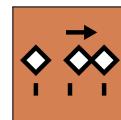
By the way: in the upper example it is possible to select *none* of the pots. That is a convenient way to reset the physical pot to the middle position so that you always have headroom for movement left *and* right, before selecting one of the virtual pots.

Input	Type	Default	Description
pot		$0 \dots 1$	The pot that you want to overlay, e.g. P1.1
bipolar			If this input is set to 1, the usual pot range of 0 ... 1 will be mapped to -1 ... +1, which converts this to a bipolar potentiometer. This is done by multiplying the output with 2.0 and subtracting 1.0 afterwards.
switch1 ... switch8			These inputs select which of the virtual pots should be changed when the physical pot is being turned. These should be set to 0 or 1 (or off and on).

Output	Type	Description
output1 ... output8		The output of the up to eight virtual pots.

13.59 timing - Shuffle/swing and complex timing generator

This circuit converts a steady input clock into an output clock with flexible timing modifications. The most common use is a "swing" feeling where every second note is delayed. But this circuit is much more flexible.



The length of a timing pattern can be up to eight steps. That means that you can set a different relative time shift for each clock pulse in a sequence of up to eight.

Let's start with a simple swing pattern, which is just a sequence of two. We assume an external input clock at G1 and output the resulting modified clock to G2:

```
[timing]
clock = G1
output = G2
timing1 = 0.0
timing2 = 0.3
```

In this example every second clock pulse is delayed by 30% of one clock tick's duration - which gives a standard swing pattern.

Creating a *reverse swing*, where every second pulse is *early* is as easy as using a negative number for **timing2**:

```
[timing]
clock = G1
output = G2
timing1 = 0.0
timing2 = -0.3
```

Creating a sequence with an odd number of steps can create rather weird groove patterns. Look at the following example:

```
[timing]
clock = G1
output = G2
timing1 = 0.0
timing2 = 0.2
timing3 = 0.1
```

Now every second note of *three* is delayed by 20% and every third note by 10%.

Of course, you can use **timing** in order to create a simple clock shift by creating a pattern with just one timing, as well. The following example will shift the input clock *forwards*, so that it always comes a bit earlier. This can be used for compensating a slight delay of a master clock:

```
[timing]
clock = G1
output = G2
timing1 = -0.03
```

Notes:

- This circuit needs a steady and stable input clock.
- In order to get a synchronized start together with the rest of your patch, it is advisable also to make use of the **reset** input.
- You cannot shift a beat forward or backward by more than 99.99% of a clock tick.
- When you set your timings in a way that two beats happen at the same time, just one trigger is output for these two beats.
- When you set your timings in a way that a later beat would come before an earlier beat, the later beat is not played.
- For each input beat there is at max one output beat. If for any input beat the corresponding output beat has already been played, it will not be replayed even if you suddenly shift it into the future.
- If an output beat has not yet played because it is delayed and then you suddenly reduce the delay by an amount that would shift that beat into the past, it is played immediately (so it is not lost).

Input	Type	Default	Description
clock			Patch a steady clock here for this circuit to be of any use
reset			A trigger here resets the internal step counter and restart at step 1.

Input	Type	Default	Description
<code>timing1 ... timing8</code>			<p>Specifies a <i>relative</i> timing for each step in relation to the input clock. A <code>timing</code> of 0.3 will shift the respective beat 30% of a clock cycle behind, while -0.3 will make it 30% early.</p> <p>The timing values are clipped into the range -0.9999 ... 0.9999.</p>

Output	Type	Description
<code>output</code>		Here comes the modified output clock

13.60 togglebutton - Create on/off buttons (OBSOLETE)

This circuit has been superseded by the new circuit **button** (see page 103). **button** can do all **togglebutton** can do and much more. So **togglebutton** will be removed soon.



This small utility circuit converts a normal push button into a toggle button that is either **on** or **off**. It toggles its state every time the button is being pressed. It even can persist the current state of the button in the DROID's internal flash memory, so at the next time you start your modular the button will have the same state as just before you switched it off.

Typically you will wire **button** to one of your controllers' buttons like **B1.1** and **led** to the LED in that button (**L1.1**). LED will then always visualise the current state of the button. As a side effect the LED register **L1.1** will store the button state as a value **0** or **1** and hence can be used by some other DROID as an input.

Here is a typical example. The button is being used for enabling the loop in the CV looper:

```
[togglebutton]
  button      = B1.4
  led        = L1.4

[cvlooper]
  loop       = L1.4
```

If you do not want the state of the button to be persisted in the DROID's flash memory then use **startvalue** for setting a start value. This make sense for the CV looper since the loop is apparently empty anyway if you start your DROID. By the way: **off** is a synonym for **0**.

```
[togglebutton]
  button      = B1.4
  led        = L1.4
  startvalue = off
```

```
[cvlooper]
  loop       = L1.4
```

Since a multiplication with **0** or **1** can switch off or on a signal you can use the LED register directly for enabling a signal. The next example uses a button for switching between 0 V and the output of an LFO:

```
[togglebutton]
  button      = B1.4
  led        = L1.4
```

```
[lfo]
  level      = L1.4 # 0 or 1
  sine        = 01
```

Usually the toggle button switches between the two values **0** and **1**. Sometimes you need different values. Therefore there are the two inputs **offvalue** and **onvalue** for two alternative values for these two states and the output **output1** where you can fetch that value (since **led** will continue to send **0** or **1** in order for the LED to work properly). Here is an example for a toggle button that switches a clock divider between 2 and 4:

```
[togglebutton]
  button      = B1.4
  led        = L1.4
  offvalue   = 2
  onvalue    = 4
  output     = _CLOCK_DIV
```

```
[clocktool]
  input      = G1 # external clock
  output    = G2
  divide    = _CLOCK_DIV
```

Of course **offvalue** and **onvalue** are CV controllable. How can make this sense? Well - as they can take variable inputs you can use a togglebutton for directly switching between two different input CV signals. The following example will send two different wave forms of an LFO to **01**. The button **B3.1** switches between sawtooth and sine:

```
[lfo]
  hz         = 2
  sawtooth   = _SAWTOOTH
  sine       = _SINE
```

```
[togglebutton]
  button      = B3.1
  led        = L3.1
  offvalue   = _SAWTOOTH
  onvalue    = _SINE
  output     = 01
```

Hint: if you need to have not only two but three or four different states for your button then have a look at the circuit **button**.

Buttons with up to four layers

The toggle button can overloaded with up to four functions. For switching between these layers you need a CV. This example assigned three different layers to one button. Each layer has its own state.

```
[togglebutton]
button    = B1.4
led       = L1.4
output1   = _ENABLE_LOOP
output2   = _FANCY_STUFF
```

```
output3   = _FOO_BAR
switch    = I1 * 2
```

Now if **I1** is near zero volts, then the button behaves like in the previous example. But when you set it to 5 V (re-

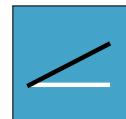
sulting in a number of **0.5** which is multiplied by **2** and thus evaluates to **1**), then a second copy of the button is activated with its own state. The LED now shows the state of that second button which **output** will outputs the value of the first button.

Input	Type	Default	Description
button			The actual push button. Usually you want to wire this to B1.1 , B1.2 and so on: to one of the push buttons of your controllers. Each time that input goes from low to high the state of the push button will toggle.
reset			A positive trigger edge here will reset the button into the state "not pressed" - regardless of its current state
onvalue		1.0	Value sent to output when the push button is on. Setting this to a different value than the default value saves you attenuating its value later on when you use it as a CV.
offvalue		0.0	Value sent to output when the push button is off.
doubleclickmode		off	This input can enable a <i>double click mode</i> when set to 1 . In that mode the button only toggles its constant state if you double press it in a short time. Otherwise it behaves like a momentary button, that inverts the persisted state (which you toggle with the double click).
startvalue			State of the push button when you switch on your system. Setting this to on or off will force the button into that state. Using this jack disables the persistence of the state! In switched mode this will be used for the other button layers as well.

Output	Type	Description
led		When the button's state is on a value of 1.0 will be sent to that output - regardless of the values in onvalue and offvalue . Usually you will wire this jack to the LED within the button, e.g. to L1.1 , L1.2 and so on
output		This jack will output either onvalue or offvalue depending on the state of the 1 st ... 4 th button. If you have not wired those inputs then this is the same as the led output.
inverted		The same as output1 , but sends onvalue when the button is off and offvalue when the button is on. Note: there is no inverted version of output2 ... output4 .
negated		Similar to inverted , but always sends 1 when the button is off and 0 when the button is on - independent of the values of onvalue and offvalue .

13.61 transient - Transient generator

This circuit creates (possibly very slow) linear transients from a defined start value to an end value. The duration of that transition is either set in seconds or specified as a number of clock ticks. This circuit is built in a way that very long transients are possible, even several days, weeks, months, years or whatever you like.



Here is a simple example:

```
[transient]
  start = 1V
  end = 3V
  duration = 600
  output = 01
```

Here the duration is meant to be 600 seconds (10 minutes). So at the beginning **01** will be at 1 V. Then it rises slowly until after ten minutes it reaches 3 V. There it stays forever.

There are two ways of restarting it again. Either you send a trigger to **reset** or you set **loop** to 1. When **loop** is active, the transient will start over at **start** immediately when it reaches **end**:

```
[transient]
  start = 1V
  end = 3V
  duration = 600
  output = 01
  reset = G1
  loop = 1
```

As an alternative to seconds you can specify the length in terms of clock ticks. This needs a steady clock signal patched into the **clock** input.

```
[transient]
  start = 0.2
  end = 0.7
  duration = 32
  clock = I1
  output = 01
```

Here the duration of one transient is exactly 32 clock ticks. This makes it simpler to exactly align a transient with a musical structure of a song or the like.

Changes while in the air

As **start**, **end** and **duration** are CV inputs, they might change while the transient is running. This is how **transient** behaves in such situations:

The **start** value is just taken into account whenever the transient starts. this is:

- When the **DROID** starts
- When there is a trigger at **reset**
- When the transient reaches the end and **loop** is on.

Whenever that happens, the current output level is set to **start**. Also the output **phase** is set to 0. Phase is a kind of internal clock that measures which part of the transient has been run through already.

At any given time **transient** assumes that the **phase** times the duration equals the time left. And the distance to go in the remaining time is the current distance from the current output level to the end. These two values directly translate into a slope. This slope now determines how fast the output level is moving and into which direction.

From this follows:

- When you make the duration longer in-flight, the speed of change will get slower.
- When you change **start** in-flight, nothing happens.
- When you change **end** in-flight to a value that is “farther” away from the current level, the speed of change increases.
- If you change **end** to be the current level of the transient, it seems to stop, but in fact the slope is just zero and it still lasts until the duration is over.
- The output level is always smooth. No sudden steps. With one exception: When the transient resets to its start value.

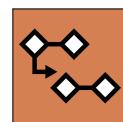
In pingpong mode (see the table of inputs for details) this changes accordingly. While the transient is on its way back, consider **start** and **end** exchanged.

Input	Type	Default	Description
start		0.0	Start value of the transient
end		1.0	Target value of the transient
duration		1.0	Duration: if the clock input is used, it is in clock ticks. Otherwise it is in seconds. A negative duration will be treated as zero. And a zero duration will make the output always be at end level.
loop		0	If this is set to 1 , the transient will start over again as soon as it reaches the end.
pingpong		0	If this set to 1 , the transient will start moving backwards towards the start when it has reached end. It will swing back and forth, in fact looping infinitely.
freeze		0	while this is set to 1 , the transient it frozen at its current position.
reset			A trigger here will immediately set the transient back to its start value.
clock			If you patch a clock here, the duration will be set in terms of clock ticks, not of seconds. This needs to be a steady clock in order to get predictable results.

Output	Type	Description
output		Here comes the current value of the transient.
phase		This output reflects the current phase of the transient. It behaves as if start would be 0 and end would be 1.
endoftransient		When loop and pingpong is off, this output goes to 1 when the transient has reached the end - and stays there. In loop mode just a short trigger is sent. In pingpong mode that trigger is not sent when the transient has reach the end -value, but when it is back at start (i.e. after one full cycle).

13.62 triggerdelay - Trigger Delay with multi tap and optional clocking

This circuit implements a CV controllable delay for a trigger or gate signal. It listens for triggers at **input** and sends the same triggers *later* to the **output**. It does *not* look at the voltage level of the inputs. The output triggers are always sent with 10 V (**I1** ... **I8**) or 5 V (on the G8 expander).



As a difference to an analog trigger delay this circuit is capable of keeping memory of up to 16 triggers. This means it is able to process further incoming triggers while previous triggers are still in the delay. This allows you to delay complex rhythmic patterns, e.g. in order to reuse the output of one track of a trigger sequencer shifted in time for another instrument.

Furthermore, it is able to retain the gate length of the original input signal and output the delayed gate with exactly the same length.

Here is the simplest possible example, which delays an incoming gates / triggers by exactly one second:

[triggerdelay]

```
input      = G1
output     = G2
```

You can set the delay in seconds via the **delay** jack. And if you patch **gatelength**, the original gate length is being ignored and overridden by this value (also in seconds):

[triggerdelay]

```
input      = G1
output     = G2
delay     = 0.1 # 0.1 seconds
gatelength = 0.05 # 50 ms
```

Clocked mode

triggerdelay supports a clocked mode, in which all timing is relative to an input clock. You enable clocked mode by simply patching a steady clock into **clock**. Now **delay** and **gatelength** are relative to *one clock cycle*.

The following example delays all input triggers by one clock cycle (which is the default):

[triggerdelay]

```
input      = G1
output     = G2
clock     = G3
```

If you specify **delay** and/or **gatelength** they are now measured in clock cycles:

[triggerdelay]

```
input      = G1
output     = G2
clock     = G3
delay     = 16 # clock cycles
gatelength = 0.5 # half a clock cycle
```

Input	Type	Default	Description
input		0	Patch triggers or gates to be delayed here.
delay		1.0	Amount of time the incoming triggers are being delayed. When clock is patched, this is in relation to one clock cycle, so a delay of 4 will delay the input pattern by exactly 4 beats. Fractions are allowed also. If clock is not patched, this parameter is in <i>seconds</i> . So for example in order to delay by 100 ms you need a delay of 0.1 .
gatelength			Unless you patch this jack the length of the output gates is exactly the length of the input gates. By use of this parameter you override that length and set a fixed length in <i>seconds</i> - or if clock is being used - in clock cycles.
repeats	1 ⋅ 2 ⋅ 3	1	Number of times the delayed trigger is being repeated. Each further repetition is with the same delay.
mute		0	A high gate signal suppresses any further output gates. However, the current gate is finished normally.

Input	Type	Default	Description
<code>clock</code>			When you patch this input, the trigger delay runs in clocked mode. In this mode <code>delay</code> is relative to one clock cycle. I.e. a delay of <code>0.5</code> will delay the trigger by half a clock cycle. The same holds for <code>gatelength</code> . That is measured in clock cycles, too.

Output	Type	Description
<code>output</code>		Outputs the delayed triggers/gates, while keeping the gate length - unless you have changed that
<code>overflow</code>		Whenever there are more input triggers than this circuit can keep memory of, this output outputs a gate of 0.5 sec length. You can wire this to an LED in order to know when this happens.