
# DROID

## Universal CV Processor

### User manual

for firmware version magenta-5

June 12, 2022

DER MANN MIT DER MASCHINE

Contents

# 1 Quick start

Welcome to the DROID. The DROID is a very flexible generic CV processor. 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.

You tell your DROID what to do by means of a simple text file called "*DROID patch*". That file is always named **droid.ini** and is located on a micro SD card. No special software is required for writing that file. A simple text editor running on Windows, Linux, Mac or any other device will suffice. If you have an X7 expander attached to your master, you can access the memory card directly from your PC or Mac via a USB-C cable.

The building blocks of a DROID patch are called *circuits*. Every type of circuit performs some basic task. Just like a Eurorack module, each circuit has inputs and outputs. You can wire these either directly to some of the inputs and outputs of the DROID module or even connect them internally in order to create more complex networks of circuits.

### A first patch example - step by step

Let's do a first simple DROID patch!

1. Install your DROID master into your Eurorack system and power it on.
2. Remove the shipped micro SD card from your DROID master und put it into the micro SD card reader that also as been shipped with your DROID. Insert that reader into a free USB port of you Laptop or PC. (Alternatively, if you have an X7 attached, connect the X7 with a free USB port of your computer and put the switch on the X7 to the left).
3. Locate the file **droid.ini** on that card with your file browser (Explorer, Finder, whatever) and open it with a *text editor* (like Notepad, TextEdit, VIM, Notepad++, etc.).
4. Delete the current contents of that file and type the following:

```
[contour]
    gate        = I1
    decay       = 0.2
    sustain     = 0.5
    release     = 0.3
    output      = O1
```

5. Save the file back to the SD card, *eject* the card properly and then remove it from the reader.
6. Insert the card back into the DROID master - with the visible pins facing downwards.
7. Press the button left of the card slot

If you have an X7, simply put the switch back to its center position after "ejecting" the SD card with windows / Mac. It will then automatically load the new patch without need to press the button.

Your DROID now will read in its new patch. If everything goes well, a LED light goes in one circle around the "display". If not, please check your DROID patch and repeat the procedure. Refer to page 7 for how to find the root cause of a problem.

This first patch creates an ADSR type envelope that is triggered at input jack 1 (**I1**) and outputs its CV on output jack 1 (**O1**). For the parameters A, D, S and R fixed values are being set for the while.

Now within the DROID, *every* parameter can be controlled via CV. So instead of setting the release to a fixed value of **0.3** you can use the second input (**I2**) for CV controlling that. This is easy:

```
[contour]
    gate        = I1
    decay       = 0.2
    sustain     = 0.5
    release     = I2
    output      = O1
```

If you have a controller such as the P2B8, the P4B2 or the P10, you can use pots for controlling parameters. First of all - for each P2B8 you need one line with the contents **[p2b8]**. Likewise for a P4B2 you need the line **[p4b2]** and for a P10 the line **[p10]**. (Note: If you mix P2B8s, P4B2s, P10s and other controllers, the order of these controllers in your chain must match the order of the declaration in your DROID patch.) Now you can access the first pot of your first controller with **P1.1**, e.g. in order to control the sustain of the envelope via that pot:

```
[p2b8]

[contour]
    gate        = I1
    decay       = 0.2
    sustain     = P1.1
    release     = I2
    output      = O1
```

We didn't dig into details yet - but you get the idea! The rest of this manual will show you the wonderful world of the DROID. First, you will learn all general ideas and features. In chapter 10 there is a complete reference of all circuit types that your DROID offers.

# 2 Installation

## Controller connector

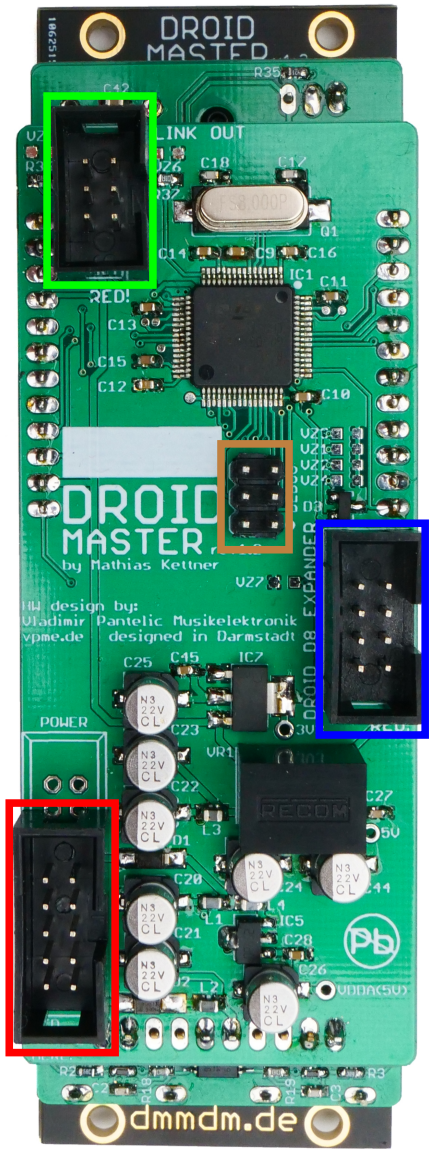
The connector for the controllers has 6 pins (two rows of three pins) and is used for connecting a chain of **P2B8**, **P4B2**, **B32**, **P10**, **M4** and other controllers. Please refer to page 21 for details.

## 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!**




## Expansion port for G8

The connector for the G8 expander has 8 pins (two rows of four pins). Here you can add **one** optional G8 expander for an additional 8 gate inputs/outputs. Please refer to page 25 for details.

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

# 3 Creating DROID patches

## 3.1 General procedure

Writing a DROID patch is what makes the DROID come to live. Without a patch your DROID is pretty useless. This is the general procedure of creating and loading a patch into your DROID:

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.

**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.

### 3.2 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:

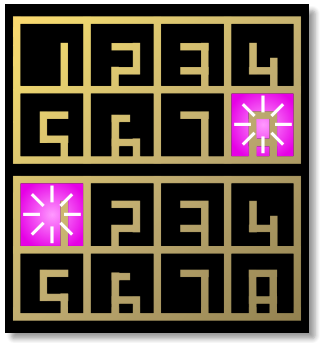
1. **General errors** concern the patch as a whole. The SD card is missing. You have misspelled the file name. Things like that. In such a case *all* LEDs will flash in the same color. The color indicates the reason of the error. On the next page you find a table of all *global error codes*.

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:

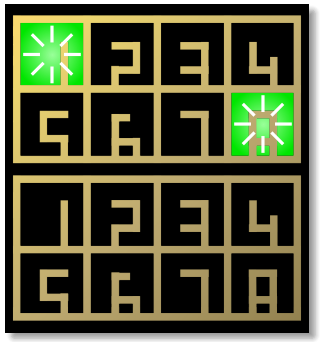
- The input LEDS 1 ... 8 indicate the *tens* of the line number. If the error happens to be in line 90, then LED 1 + 8 will flash. If it is in line 1 to 9, then no input LED flashes at all.
- The output LEDS 1 ... 8 indicate the *ones* and are added to that number. Again, if a 9 is needed, then 8 + 1 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 + ... + 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.

### 3.2.1 Examples for error codes

Invalid parameter value in line 81:



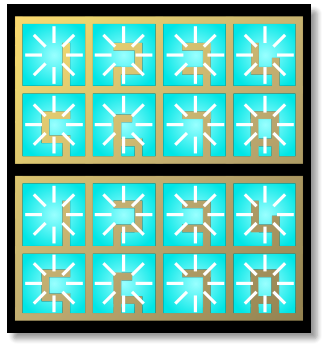
Undefined parameter in line 90:



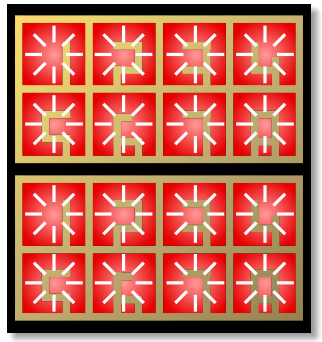
Invalid register in line 99:

Line too long in line 144:

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



Too many circuits or out of memory:



### 3.2.2 Table of error codes

*All LEDs flashing at once (global error)*

| | |
|---|---|
| **yellow** | **Patch not found**: This can happen in the following situations:<br>1. No file with the name **`droid.ini`** is present on the memory card.<br>2. You DROID started without having loaded a patch ever.<br>3. You did a factory reset without loading a patch afterwards. |
| **orange** | **Empty patch**: Your **`droid.ini`** file has a size of 0 bytes. Or it could not be read correctly. Check the file. Maybe re-insert the micro SD card and try again. |
| **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 *without* 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 41 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 11. |
| **orange** | **Unknown parameter name**: that circuit does not support that parameter. Please check the circuit references in chapter 10. |
| **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 37 you learn how to do a firmware upgrade. |
| **blue** | **Line too long**: One line in your patch exceeded the maximum allowed line length of 127 characters. |
| **green** | **Internal patch cable misused**: One of your internal patch cables (see page 15) is not properly used:<br>**1. No input:** One patch cable is only used as output.<br>**2. No output:** One patch cable is only used as input.<br>**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.<br>**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.<br>**3. Invalid parameter value**: Your parameter has an invalid value. Please checkout this manual about allowed values for parameters and their exact syntax. |

### 3.3 Basic structure of the patch file

Droid offeres 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 ouptuts 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.

### 3.4 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 the 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.

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 8$^{th}$ button on the 4$^{th}$ 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 67 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 **`R32`** let you override the function of the LEDs for the inputs, outputs and gates, also those 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.

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 G8 expander. Each can be used either as an input or as an output. |
| **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 R18 R19 R20 R21 R22 R23 R24** | output | The colored LED squares on the G8 expander |
| **R25 R26 R27 R28 R29 R30 R31 R32** | 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 2nd pot on your 3rd 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 6th push button on your 3rd controller. |
| **L1.1 L1.2 L2.1 … L2.1 L2.2 L2.3 …** | output | The LEDs in these push buttons |

And here is a table of some colors and their values that you need to send to the **R1** .. **R32** registers:

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

### 3.4.1 Status dump file

There is an easy method for getting the current value of all registers! Simply *double press* on the masters 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 **STATES.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 15).

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

Gates:
    G1:       1    G2:       0    G3:       0    G4:       1
    G5:       0    G6:       0    G7:       0    G8:       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
```

## 3.5 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.

### Voltages out of range

The DROID's hardware cannot work with voltages beyond $\pm$10 V. Anyway, Eurorack is limited to $\pm$12 V and barely any module reaches even 10 V at its output (in fact many digital modules are 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. It's completely normal. Some circuits even require such numbers. E.g. in the **`minifonion`** (see page 149) 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 300000000000000000000000000000000000000 (a 3 with 38 zeroes).

The smallest number greater than zero is approximately 0.00000000000000000000000000000000000011 (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 registers **`G1`** ... **`G8`** will set its output to 5 V, any other number to 0 V.

### 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 $\frac{1}{12\mathbf{V}}$, which is - when you convert the voltage back to a number - $\frac{1}{1.2}$, 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     = O1
```

This is exactly the same as:

```
[contour]
     loop       = 1.0
     output     = O1
```

## 3.6 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 113 for details on the LFO circuit). That pot can be addressed with **P1.1**:

```
[lfo]
    level = P1.1
    output = O1
```

The pot has a range from 0 to 1, which corresponds to 0 V ... 10 V. That's maybe too much for you 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 121).

### 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 = O1
    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 152 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 15).
- 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 121).

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.

### 3.7 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
     output = O1
```

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  = O1

[contour]
     trigger = _TRIGGER
     attack  = 0.5
     release = 0.8
     output  = O2
```

Note: There are two rules, which 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 7 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.

### 3.8 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 euclidean 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 = O1
```

### 3.9 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 registers. 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!
```

### 3.10 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 $\mu$s. 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 $\mu$s 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       = O1
```

Here an external trigger at **G1** (on the G8 expander) is being used to trigger an envelope randomly, which is then sent to **O1**. 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       = O1

[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 $\mu$s. This is not very long, but it can be easily 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 $\mu$s.

### 3.11 Parameter arrays

Some of the circuits have arrays of similar jacks, like **output1**, **output2**, **output3** and so on. Here you can always omit the digit **1** if you just want to address the first jack in the list. So **output** is just the same as **output1**.

### 3.12 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  = O2 # wired to foobar trigger
```

### 3.13 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 loosing 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 wether 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 ciruits 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), the **pot** (the current value of all up to eight virtual pots) and so on.

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

Please note: All these states are saved to the micro SD card into a file with the name **DROIDSTA.BIN**. That 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 and 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 **loose 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 loose your settings.

### 3.14 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 easily 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 **droid**$XY$**.ini**, where $X$ is the number of the controller and $Y$ the number of the button. Then **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 is extended 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.

### 3.15 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 12). That shows you the exact value of every single input, output, potentiomenter 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.

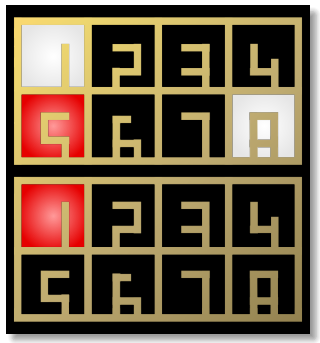
Here the experimental **X1** register helps. It's an output register. When you send a value there, all the LEDs of the front panel will show that value in a way similar to the line-error-encoding of the patch parser. Here is an example:

```
[p2b8]

[contour]
    attack = P1.1
    release = P1.2
    trigger = B1.1
    output = O1

[copy]
    input = P1.1
    output = X1
```

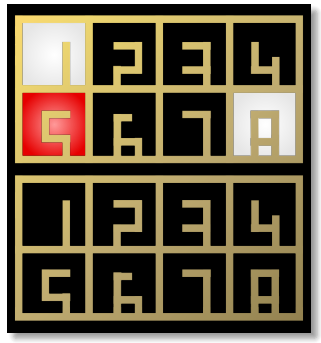
Now turn the knob **P1.1** for setting some nice attack value. As soon as you remove that from its zero-position, all LEDs will move around in red and white and show the current value of **P1.1** with three digits. Input LEDs are lit white and red. White digits account for 0.1 and red digits for 0.01.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



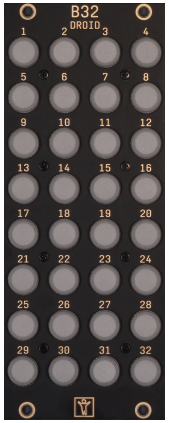
But what if digits in the input section collided? E.g. 0.550 would need the LED of input 5 to be red and white at the same time. Well, then it will blink between white and red:

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.

# 4 Controllers and Expanders

The DROID master can be extended with an ever growing range of controllers and other expanders. These are what makes the DROID ecosystem so flexible. You can attach up to 16 controllers to your DROID.




The **B32** controller provides you with a plentitude of 32 freely assignable buttons.


The **P2B8 controller** has two potentiometers (pots) and eight push buttons - thus the name P2B8. You can freely use these pots and buttons for any purpose in your DROID patch. Using controllers is very easy and adds lots of playability.


The **P4B2** is very similar to the P2B8. The only difference is that it has four big pots and just two buttons. They are useful if you need more controls of continuous values but the small pots of the P10 are too small for you.


The **P10** works very similar to the P2B8, just it has no buttons but 10 pots. Two are large and eight are small, but all of them have the same functionality. You can control any parameter with each of the pots.

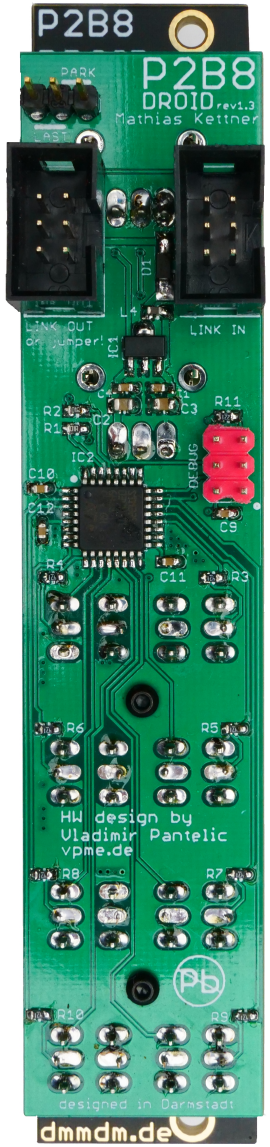

The **M4 Motor Fader Unit** brings four motorized faders to your DROID. These can be used either for easy switching between presets or for overloading one fader with lots of different functions at the same time.


The **S10** controller has two mechanical rotary switches with eight positions each and eight toggle switches with three positions. The large switches outputs number from 0 to 7. The small switches output 0, 1 or 2.


The **G8 expander** extends your DROID by eight gate inputs or outputs, which is perfect for clock, trigger and gate signals. Every jack can be used as a gate input or output. You can attach one G8 to your DROID.

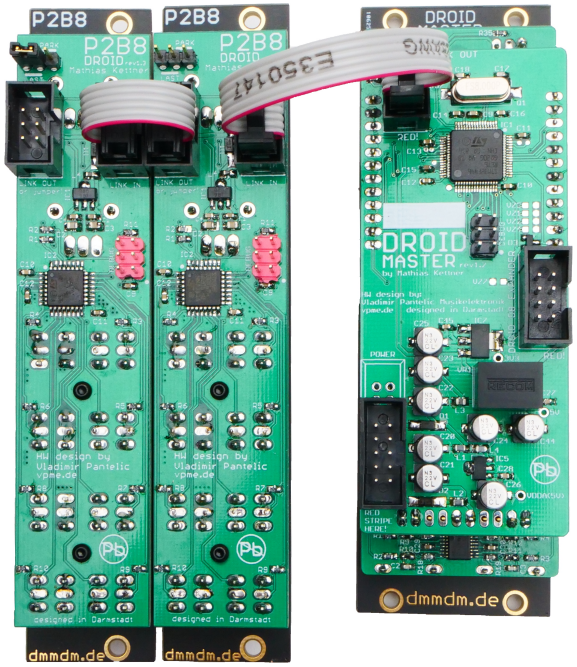

The **X7** expander provides three different functions: MIDI in + out, both via USB and DIN/TRS, supporting both Korg and Arturia standard (also known as MIDI-A and MIDI-B), direct access to the masters's SD card via USB and four additional gate outputs.

## 4.1 The P2B8 controller


Using controllers is very easy and adds lots of playability to your DROID patch. The P2B8 controller is the most common and flexible of the DROID controllers and it was the first one available. This chapter shows you how to install and use it. The same does apply for the P4B2, P10 and B32 controllers - just that the P10 has no buttons and the B32 has no pots. The M4 is special and will be described in a dedicated chapter once it is available.

### Installation




1. Wire the controller output of the master to the first controller by use of the **6 pin** ribbon cable. Make sure that you attach it to the **input** header of the P2B8 controller. Put the red stripe down on both modules.
2. If you use more than one controller, connect the LINK OUT header of each controller to the LINK ON header of the next one.
3. On the **last** controller, set the jumper to **Last**.
4. On **all other** controllers, **remove** the jumper or set it to **Park**.

**Note (1): do not mix up input and output. The right hand connector must be connected to the master, the left hand one to the next controller.**

**Note (2): do not use the 6 pin programming header (the one without the box) on your master or P2B8!**

The controller modules are powered by the master. When you switch on your system, all controllers will flash the LEDs for a short time, to show you that you have wired them correctly.

If you set the jumpers not correctly, the controllers will power up and flash their LEDs as usual, but the buttons and pots will not work.

If the LEDs on the first controller behave as they should but not the buttons and pots then you have probably set the jumpers incorrectly. Please check.

Another test is pressing a button: If you have correctly declared your controllers in your DROID patch, the LED in that button should be lit as long as you hold the button. This shows that the communication with the master is working fine.

## 4.2 The P10 controller


The P10 controller is very similar to the P2B8 controller. Please look at page 21 for how to connect the controllers to your DROID master and how to chain them. The only difference is that the P10 does not have any buttons (nor LEDs in these buttons) but instead eight small pots. That makes a total of 10 pots - 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 adressed 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.

## 4.3 How to use controllers in your patch

Before you can use the controllers in your patch, you need to declare them right at the top of your patch: Just write one line with the content **`[p2b8]`**, **`[p10]`**, **`[b32]`**, **`[p4b2]`**, **`[m4]`** for each for your controllers. The order of these declarations must exactly 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]
```

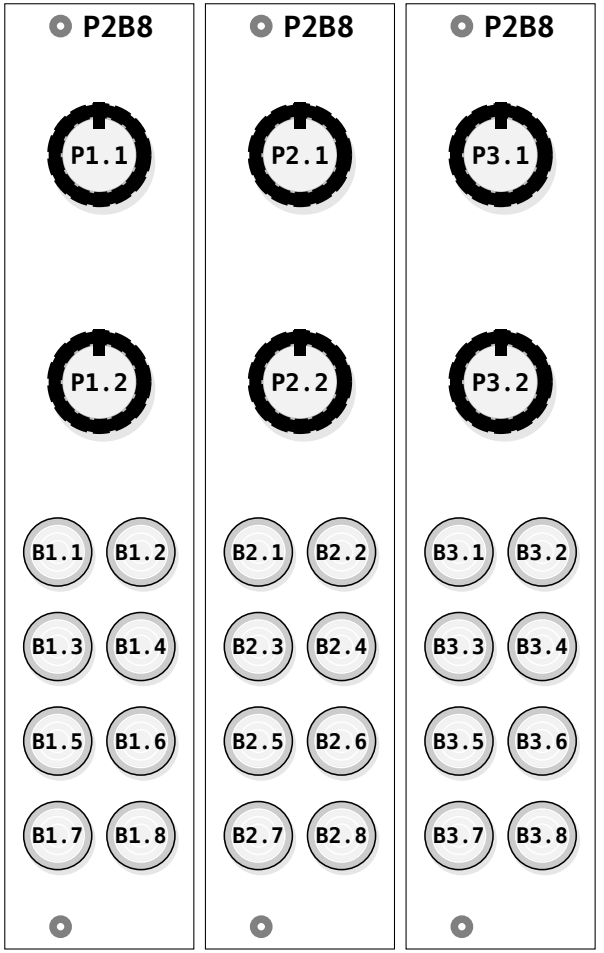
Now you can use the pots, buttons and LEDs by indicating these special registers in your patch as follows:

| | |
|---|---|
| **P***x.y* | potentiometers |
| **B***x.y* | buttons |
| **L***x.y* | LEDs in buttons |

Replace *x* with the number of the controller and *y* with the number of the pot or button 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 = O1
    attack = P2.5
    release = P2.6
```

### 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  in the table of inputs) you can set a pot here. An example would be an envelope generator:

```
[p10]

[contour]
    gate    = G1
    output  = O1
    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 14. Let's make attack just go from 0.0 to 0.3:

```
[p10]

[contour]
    gate    = G1
    output  = O1
    attack  = P1.3 * 0.3
    decay   = P1.4
    sustain = P1.5
    release = P1.6
[p2b8]
```

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  = O1
    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  = O1
```

For more complicated tasks about pots there is the circuit **pot** (see page 169). 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 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.

Here the circuit **button** helps (see page 67). 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    = O1
```

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 67 for all the details.

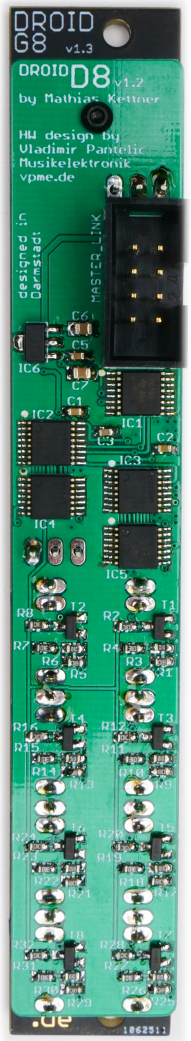
### 4.4 Controller latency

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.

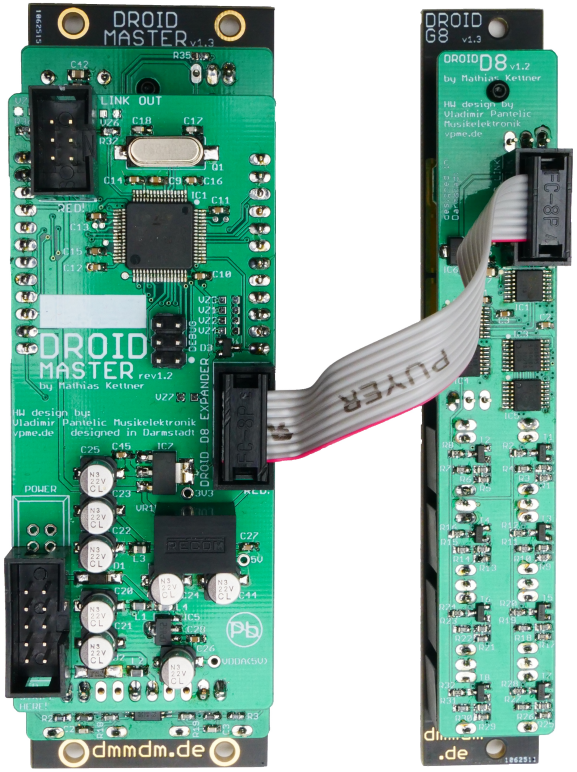
This serial line sends approximately 7200 bytes per second. Every controller needs a different number of bytes per update and for the P2B8 it's 11 bytes. So if you have just one P2B8, you get $\frac{7200}{11} = 654$ updates per second. That's roughly one update per 1.5 ms - which is pretty fast. That 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 get's 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).

### 4.5 The G8 expander


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 expander gives you 8 further digital inputs and outputs. These are accessible via **G1**, **G2** ...**G8**. They can be used as clock and reset inputs, trigger outputs and similar tasks.

- 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 to **G1** ...**G8**. And 0 V otherwise.

The G8 also has 8 multicolored LEDs. These will indicate inputs in blue lights and outputs in red when high. You can override the default function of LEDs in order to signal something. Use the registers **R17** ...**R24** for that purpose.

There is nothing special to do in your **droid.ini** for setting up the G8 expander. Using **G1** ...**G8** *without* actually having the expander will simply behave as if nothing was patched there.

One question aside: Why do the gates not output 10 V? Well, while this would be more logical, it was actually impossible to do in hardware easily since we use a very special chip here that is able to switch between input and output via software. And this chip does not support 10 V. 99.9% of all eurorack modules will happily accept 5 V as a valid trigger. If that's not the case for you, simply use one of the outputs of the DROID master.

# 5 The X7 expander

## 5.1 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 **midiin** (see page 133), **midiout** (page 140) and **midithrough** (page 148).
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.

## 5.2 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 real time CVs and triggers.
- 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

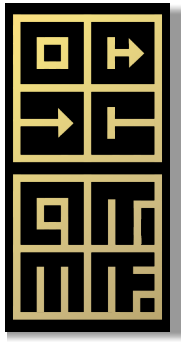
The X7 has the following jacks:

- 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.

## 5.3 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.

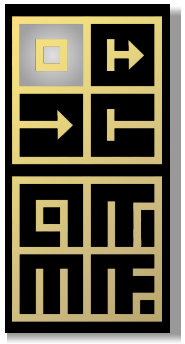
Just like all the controllers, the X7 has an *input* connector, which is at the top *right* side if you look from the back. On the *left* side is the *output* connector. Connect the master with the shipped 6 pin ribbon cable to the *input* connector. If you have any controllers, like P2B8, P10 and so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

## 5.4 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: Please use the USB-A ↔ USB-C cable that was shipped with the X7 or a similar one. USB-C ↔ USB-C cables do not work!**

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.

Now 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.

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 7) 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 42.

## 5.5 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:

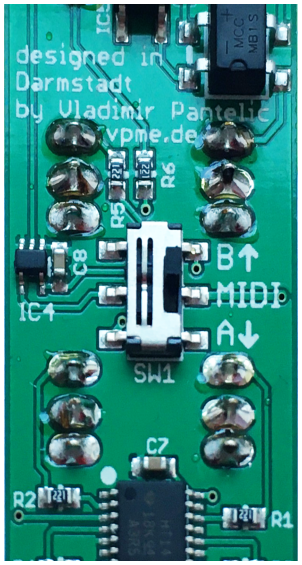
- Support for both MIDI → CV and CV → MIDI at the same time.
- Unlimited polyphony (number of simultaneuous notes) except that you run out of jacks.
- The MIDI streams of USB and TRS can be used independently in parallel, so you have two input and two output streams.
- Flexible "MIDI through" routing 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.

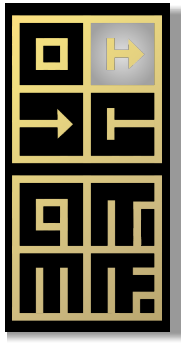
By the way: MIDI over USB is not restricted to the standard MIDI data rate of 31250 bits per second.

### The LEDs

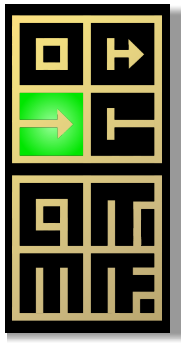
When working with MIDI, watch the corresponding LEDs. Here is what the colors mean:

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

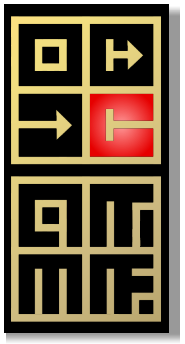
The top right LED shows the status of USB-MIDI:



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 hold.

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 **`midiin`** (see page 133). This example converts MIDI to a pitch CV at output **`01`** and a gate at output **`02`**:

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

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

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

Per default, **`midiin`** 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 simultaneuous notes:

```
[midiin]
    pitch1 = O1
    pitch2 = O2
    pitch3 = O3
    gate1 = O5
    gate2 = O6
    gate3 = O7
```

If you have a G8 expander (see page 25) you directly control eight analog voices:

```
[midiin]
    pitch1 = O1
    pitch2 = O2
    pitch3 = O3
    pitch4 = O4
    pitch5 = O5
    pitch6 = O6
    pitch7 = O7
    pitch8 = O8
    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 = O1
    gate = O2
    volume = O3
    modwheel = O4
    ccnumber1 = 17 # get CC number 17
    cc1 = O5       # output that on O5
```

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 149):

```
[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 = O1
```

This was just a quick overview and there are much more inputs and outputs available. Please have a look at page 133 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 **midiout** (see page 140) 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**:

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

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

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

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

```
[midiout]
    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 149 for details on the **minifonion** circuit):

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

[midiout]
    pitch = _PTICH
    gate = I2
```

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

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

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

[midiout]
    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 140 for all details on the **midiout** 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!

### 5.6 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 **midiout** (see page 140)) or processing the events (using **midiin** (see page 133)).

Use the **midithrough** (see page 148) 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 **midiin** and **midiout** 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 **midiout** 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.

### 5.7 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 G8 expander (see page 25), 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:

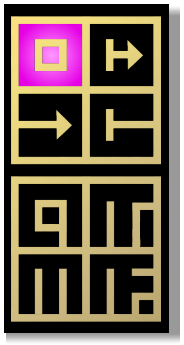
```
[midiin]
    clock = G9   # 16th notes
    clock8 = G10 # 8th notes
    clock4 = G11 # quarter notes
    start = G12  # trigger at MIDI start message
```

### 5.8 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 **R25** through **R32** for that purpose.

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

```
[p2b8]
[copy]
    input = P1.1
    output = R25
```



### 5.9 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.

There is a slight disadvantage, of course. And that is the fact that the sysex mode is a bit more complicated to setup and it needs *special software*. 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 including 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** relies on a MIDI framework called *Juce*. And that can be unstable for larger sysex files - just as those that we need for the DROID. If the most current version does not work try version **1.0.14**. It worked for me. And we have included a binary of that in utilities/mac/sysex that you can use.

### 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 outupt 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**:

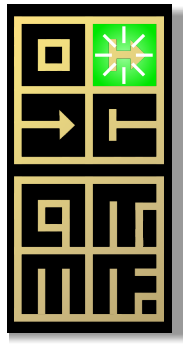
```
C:\Users\dmmdm\patches> sendmidi list
Microsoft GS Wavetable Synth
DROID X7 MIDI
```

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**:

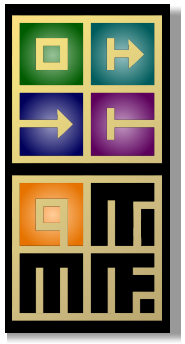
```
C:patches> sendmidi dev x7 syf yourpatch.syx
```

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

### 5.10 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:



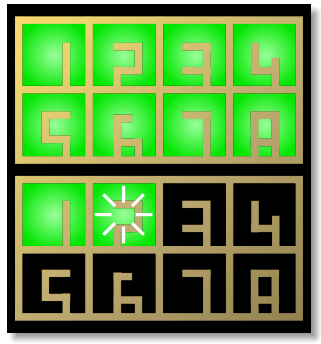
In order to make things as easy as possible for you, the software update for the X7 is done by the master. You don't need to change anything in your cabling for that. Leave the X7 attached as the first expander on the master.

Here are the steps for an X7 firmware upgrade:

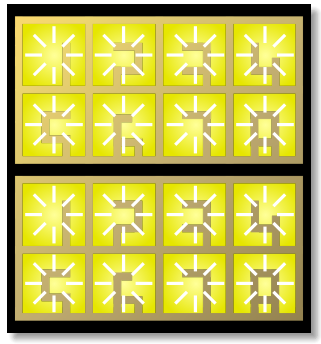
1. Copy the firmware file for the X7 (from Discord or from our Download page) to the SD card in the master.
2. Rename it to exactly **x7.fw**
3. Bring the master into the maintenance mode (see page 40 for details). Long things short: this is done by a very long button press.
4. Your maintenance menu 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.

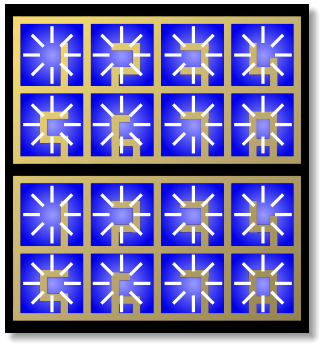
If everything goes well, you see a kind of progress bar running through all 16 master LEDs, while the X7 does the same kind of animation with its 8 LEDs.



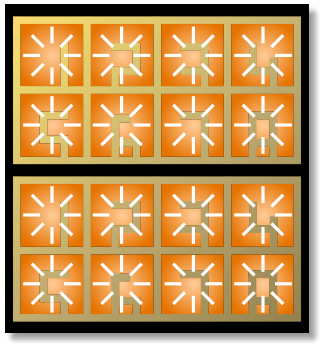
In case of an error, all 16 LEDs blink in one color. If all LEDs blink **yellow**, the firmware file is missing (which is strange, because it was there at the beginning):



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:

### 5.11 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 process 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.

# 6 The M4 motor fader controller


## 6.1 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 156), **`faderbank`** (see page 104), **`fadermatrix`** (see page 106) and **`motoquencer`** (see page 153) 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 a small number of minutes. While they are charging, here LEDs show a colored animation and go from red through yellow to green and finally off.

## 6.2 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 34 with the following additional notes:

- The firmware file on the SD card must have the name **`m4.fw`**.
- In the master's maintainance 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".

# 7 Firmware upgrade

## 7.1 What version do you have?

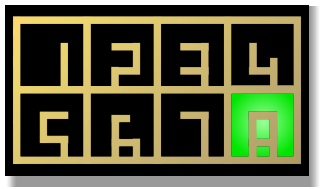
DROID is an active project, new features are being added, bugs are being fixed. Also new controller modules require changes in the software of the master module. All these things are reasons why, from time to time, we release a new firmware (software) version for the DROID master.

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 .

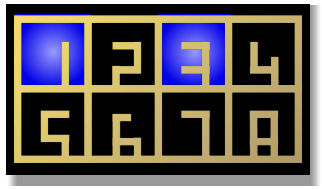
Unless most other software, DROID uses a combination of a color and a number in order to name a software version. For example the version this manual is written for is called **magenta-5**.

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!):

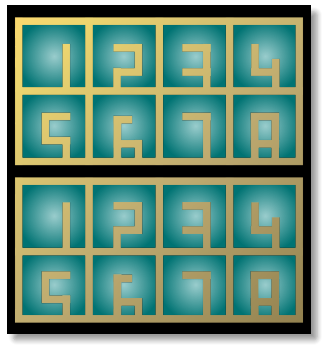


## 7.2 Normal update procedure

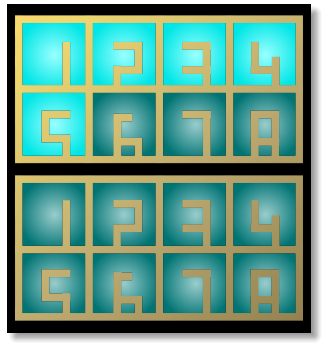
Here is how you upgrade the firmware of your DROID:

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:

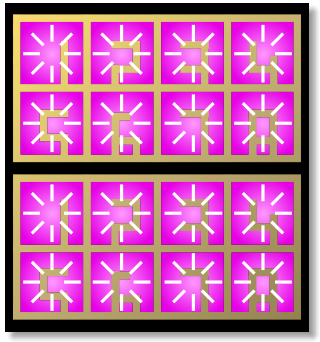
### Missing firmware file

If you have not copied the file **droid.fw** or missspelled 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

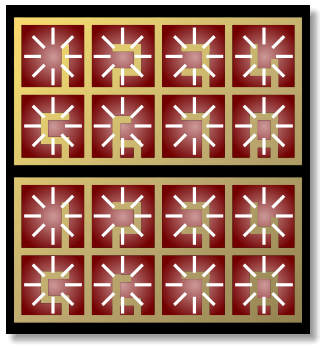
A magenta blink code means that your firmware file **droid.fw** is somehow not valid. It has the wrong size. This usually has one of two reasons:

- 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.

### 7.3 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.

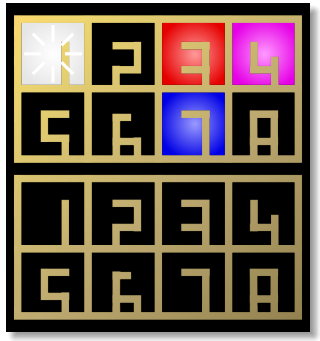
With the bootloader from the green firmware you will get all LEDs flashing magenta if you want to update to **blue-1** (or any other blue firmware). So in this case you need to do the following steps:

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-1** 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 40 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 uptodate (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 your 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 the be reprogrammed in our labs if that happens.**

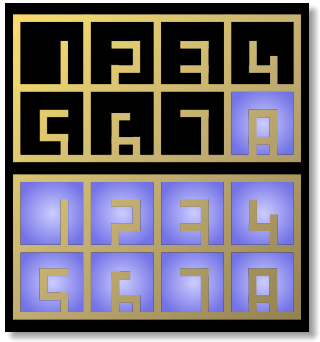
If you enter the maintenance menu again, the menu item 7 should have disappeared, since your bootloader is now up-to-date.

If you need any help, please post a question on our Discord community .

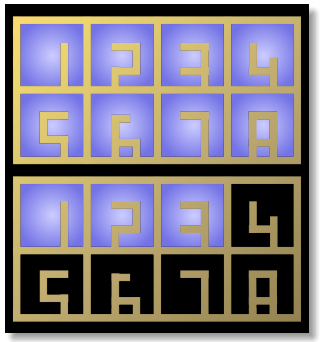
# 8 Calibration, Factory Reset other maintainance stuff

## 8.1 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.

## 8.2 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!

## 8.3 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 get's lost: When you switch of 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 an 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 looses it's 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.

## 8.4 Using your own SD card

### 8.4.1 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.

### 8.4.2 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 you card is **Untitled**. If not, please adapt the commands to your name:

```
mdutil -i off /Volumes/Untitled
cd /Volumes/Untitled
rm -rf .{,_.}{fseventsd,Spotlight-V*,Trashes}
mkdir .fseventsd
touch .fseventsd/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!

# 9 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 an optional G8 expander
- Expansion port 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 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

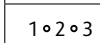
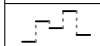
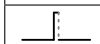
Power consumption:

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

# 10 Reference of all circuits

This is a reference of all circuits that are supported by firmware version magenta-5 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 *continous* 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 169 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 *stepped* 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 you wire an external input to such a jack, it will see a **1** at a voltage of at least 1 V and 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 *duration* 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.

### 10.1 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 = O1
    bit2 = O2
    bit3 = O3
```

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 |
|---|---|---|---|---|
| $-\infty$ ... 0.125 | **0** | **0** | **0** | **0** |
| 0.125 ... 0.250 | **0** | **0** | **1** | **1** |
| 0.250 ... 0.375 | **0** | **1** | **0** | **2** |
| 0.375 ... 0.500 | **0** | **1** | **1** | **3** |
| 0.500 ... 0.625 | **1** | **0** | **0** | **4** |
| 0.625 ... 0.750 | **1** | **0** | **1** | **5** |
| 0.750 ... 0.875 | **1** | **1** | **0** | **6** |
| 0.875 ... $\infty$ | **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 = O1
    bit2 = O2
    bit3 = O3
```

Now the table looks like this:

| input | **bit1** | **bit2** | **bit3** | bit value |
|---|---|---|---|---|
| $-\infty$ ... 0.15 | **0** | **0** | **0** | **0** |
| 0.15 ... 0.20 | **0** | **0** | **1** | **1** |
| 0.20 ... 0.25 | **0** | **1** | **0** | **2** |
| 0.25 ... 0.30 | **0** | **1** | **1** | **3** |
| 0.30 ... 0.35 | **1** | **0** | **0** | **4** |
| 0.35 ... 0.40 | **1** | **0** | **1** | **5** |
| 0.40 ... 0.45 | **1** | **1** | **0** | **6** |
| 0.45 ... $\infty$ | **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 97, 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. |

One **adc** circuit needs **116** bytes of RAM.

## 10.2 `algoquencer` - Algorithmic sequencer

The Algoquencer is a versatile sequencer with a strong focus to live performances. It implements a completely new approach: It combines a classical trigger sequencer with a turing machine and other randomization algorithms in order to create a very hands on pattern generator for live improvisation. It's 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 or 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  = O1
```

You can change the length to any other value up to 64 by using the **`length`** parameter:

```
[algoquencer]
    clock  = G1
    pitch  = O1
    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     = O1
```

**`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   = O1
    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       = O1
    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 = O1
    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 artifical beats if it is not zero.

Last but not least wire the output **trigger** to the trigger input of some drum voice.

```
    trigger = O1
```

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 = O1
```

### 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       = O3
```

Now while you hold the accent button the step buttons will switch over to showing the accents intead 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       = O3
```

### 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 two 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.

### 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 **buttongroup**, 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:

```
[buttongroup]
    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 |
|---|---|---|---|
| **preset** | 1∘2∘3 | **0** | This is the preset number to save or to load. Note: the first preset has the number **0**, not **1**! This circuit has 16 presets, so this number ranges from **1** to **16**. |
| **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. |
| **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 *exactly* **0** instead of a positive gate signal. In some cases this is more conventient. |
| **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. |

| Input | Type | Default | Description |
|---|---|---|---|
| **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 | **0** | 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. |
| **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. |
| **clear** |  | | 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. |
| **gatelength** |  | **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. |
| **lengthbutton** |  | | Map this to a button like **B1.1**. While you press and hold this button the sequencer switches to *change length mode*. 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. |

| Input | Type | Default | Description |
|---|---|---|---|
| **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 1st and 9th step are set by **button1**, 2nd and 10th by **button2** and so on.<br><br>Why should that be useful? Well - the difference shows up when you use **fills**, or **branches** or work with the *alternate* pattern. These three algorithms work based on *bars*. 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.<br><br>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.<br><br>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 A B.<br><br>With **branches = 2** you get a sequence of the form A B A C.<br><br>A value of **3** creates an even longer sequence that repeats itself after eight bars: A B A C A B A D.<br><br>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 then button once then all triggers are muted. Pressing again unmutes them. So this behaves like a toggle **[button]** in itself. You probably want to wire **muteled** 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. |
| **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 **unmuteled** 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. |

| Input | Type | Default | Description |
|---|---|---|---|
| **accentbutton** |  | | While this input is high you are in *accent editing mode*. 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 *alternate editing mode*. Every Algoquencer has an alternate set of steps. Each step that is currenty activated *toggles* 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** |  | **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. |
| **accenthigh** |  | **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** |  ☞ | | 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 (und thus a changing output pitch).<br><br>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'clock) 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 *additional* beats will be placed into you pattern in a random way until - at **1.0** - a trigger will happen at *every* beat.<br><br>**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 *have* 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. |
| **variation** |  | **0.0** | The *variation* 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 *activity* goes beyond **0.5**, additional beats will be placed, of course. And these are random.<br><br>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. |

| Input | Type | Default | Description |
|---|---|---|---|
| **dejavu** | 0 1 | **1.0** | The **dejavu** parameter controls what *random* 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.<br><br>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".<br><br>Any position in between will choose some of the steps as chaotic random and some of the steps as deterministic. |
| **morphs** | 0 1 | **0.0** | 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 *some* of the deterministic random decisions will be remade. Setting **morphs = 1** will essentially disable **dejavu**, since all decisions are redone every bar anyway then.<br><br>If you know the Turing Machine: In principle that has the same idea, but Algoquencer has a few improvements:<br><br>· 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.<br>· Changes only appear at the beginning of each bar. If you use **branches**, they will appear whenever you sequence is over.<br>· Small settings will introduce just one morph each $64^{th}$ step. |
| **offbeats** | 0 0.5 1 | **0.5** | Whenever random beats are being placed then this setting controlls whether *downbeats* or *offbeats* should be preferred. At at 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 *even* number, like 2, 4, 6 and so on. Value smaller than **0.5** will prefer downbeats.<br><br>Offbeats sound more "complex" and downbeats more simple or "down to earth". |
| **distribution** | 0 0.5 1 | **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. |
| **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 *remove* or change some instead. |

| Input | Type | Default | Description |
|---|---|---|---|
| **fillorder** | 1∘2∘3 | **0** | This integer number controls how fills are being placed:<br>**0** every bar<br>**1** every second bar<br>**2** small fill in bar 2, big fill in bar 4<br>**3** tiny fill in bar 2 and 6, medium fill in bar 4, big fill in bar 8 |
| **rolls** |  | **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 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. |

| 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. |

| Output | Type | Description |
|---|---|---|
| **led1 ... led16** |  | 1st ... 16th 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. |
| **muteled** |  | Wire this to the LED in your mute button. It will then be lit while the voice is muted. |
| **unmuteled** |  | 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. |
| **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** |  | 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. |

One **algoquencer** circuit needs **1976** bytes of RAM.

## 10.3 `arpeggio` - Arpeggiator - pattern based melody generator

### Introduction

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. 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 parametes 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 78) and **`minifonion`** (see page 149). 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.

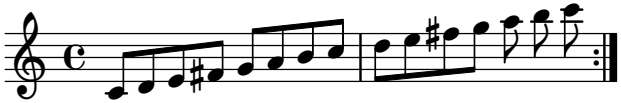
### The simplest possible example

As always, we start with the simplest possible example. And it is simple, indeed, since each of the many parameters 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 = O1
```

Patch **`I1`** to an external clock and **`O1`** 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:



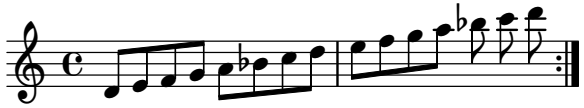
When it reaches the end it immediately starts over again. So the second "bar" is really just 7 eights here!

### 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 149) for an explanation of these parameters. Let's go for a D minor (natural) scale as an example:

```
[arpeggio]
    clock = I1
    output = O1
    root = 2
    degree = 7
```

Now we get:

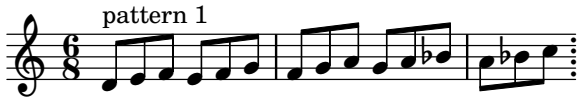


### Patterns

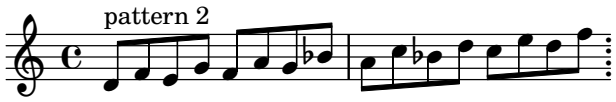
This "go through the scale" mode is just one of several possible patterns. The pattern is selected with the **`pattern`** input. And the default value of **`0`** produces the result we just have seen. Let's look at pattern **`1`**. This goes two steps forward and one step backward in the scale:

```
[arpeggio]
    clock = I1
    output = O1
    root = 2
    degree = 7
    pattern = 1
```

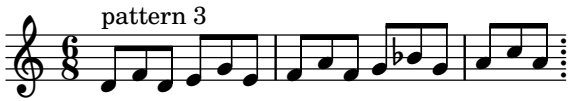
Since pattern 1 repeats its structure every three notes it's best to display it in a metric that is divisible by three:



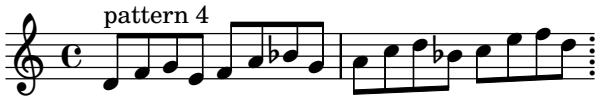

Pattern **`2`** is similar, but makes one double step forward instead of two single steps:




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 entirely impossibl 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 chose 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 = O1
    pitch = P1.1
    range = P1.2
```

### Changing the playing direction

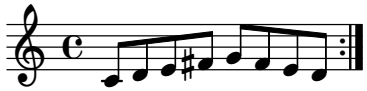
So far all pattern where going more or less upwards. From lower notes to higher notes. This can be changed by setting **direction** to **1**. Now the arpeggiator starts with the highest allowed note and reverses the pattern for going downwards. Why not map this setting to a nice toggle button?

```
[p2b8]

[button]
    button = B1.1
    led = L1.1
    output = _DIRECTION

[arpeggio]
    clock = I1
    output = O1
    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 = O1
    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.

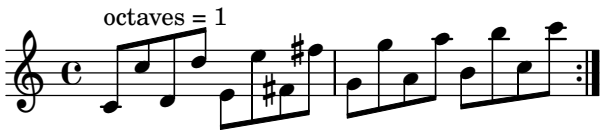
### Octaves up and down

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**.

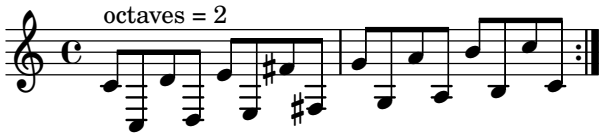
When octaves is **1**, each note is directly followed by the same note one octave above. That octave note is ignoring the **range**-parameter. It is always in addition to the selected range. Here is an example:

```
[arpeggio]
    clock = I1
    output = O1
    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:

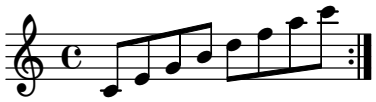



### Dropping

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:

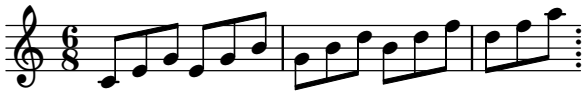
```
[arpeggio]
    clock = I1
    output = O1
    drop = 1
```

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 149) and the chord generator (page 78).

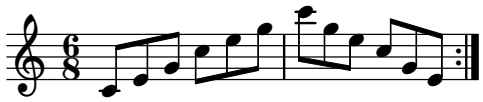
The point is that you are not restricted to the seven notes of a scale. For this there are seven inputs **`select1`**, **`select3`**, ... **`select13`** that select the notes of the current scale and another five inputs **`selectfill1`** ... **`selectfill5`** that select the notes not in the current 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`** | $3^{rd}$ | **`III`** | E | E♭ |
| **`select5`** | $5^{th}$ | **`V`** | G | G |
| **`select7`** | $7^{th}$ | **`VII`** | B | B♭ |
| **`select9`** | $9^{th}$ = $2^{nd}$ | **`II`** | D | D |
| **`select11`** | $11^{th}$ = $4^{th}$ | **`IV`** | F | F |
| **`select13`** | $13^{th}$ = $6^{th}$ | **`VI`** | A | A♭ |

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 = O1
    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 = O1
```

Now you can switch on and off scale notes for being part of the patterns. Have fun!

| Input | Type | Default | Description |
|---|---|---|---|
| **root** | 1∘2∘3 | **0** | Set the root note here. **0** means *C*, **1** means $C\sharp$, **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. |
| **degree** | 1∘2∘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. |
| **select1** | ⎍ | ☞ | Gate input for selecting the *root* note 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.<br><br>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. |
| **select3** | ⎍ | ☞ | Gate input for selecting the 3rd. |
| **select5** | ⎍ | ☞ | Gate input for selecting the 5th. |
| **select7** | ⎍ | ☞ | Gate input for selecting the 7th. |
| **select9** | ⎍ | ☞ | Gate input for selecting the 9th (which is the same as the 2nd). |
| **select11** | ⎍ | ☞ | Gate input for selecting the 11th (which is the same as the 4th). |
| **select13** | ⎍ | ☞ | Gate input for selecting the 13th (which is the same as the 6th). |
| **selectfill1** | ⎍ | **off** | Selects the alternative 9th (i.e. the 9th that is *not* in the scale. |
| **selectfill2** | ⎍ | **off** | Selects the alternative 3rd (i.e. the 3rd that is *not* in the scale). |

| Input | Type | Default | Description |
|---|---|---|---|
| **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 *not* in the scale). |
| **selectfill5** |  | **off** | Selects the alternative $7^{th}$ (i.e. the $7^{th}$ that is *not* 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 ment 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. |
| **pitch** | ♪ $\frac{1V}{Oct}$ | **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** | ♪ $\frac{1V}{Oct}$ | **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. |
| **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 |

| Input | Type | Default | Description |
|---|---|---|---|
| **drop** | 1∘2∘3 | **0** | Selects a scheme of skipping some of the allowed scale notes. Four different values are allowed:<br>**0** Do not skip any notes ❶❷❸❹❺❻<br>**1** Skip every second selected note ❶②❸④❺⑥<br>**2** Skip every third selected note ❶❷③❹❺⑥<br>**3** Skip the 2nd and 3rd note of each group of three ❶②③❹⑤⑥ |
| **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. |
| **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 7th as start note. |

| Output | Type | Description |
|---|---|---|
| **output** |  | This is what it's all about: here comes the pitch CV for the current arpeggio note. |

One **arpeggio** circuit needs **504** bytes of RAM.

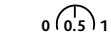
### 10.4 `bernoulli` - Random gate distributor

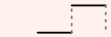
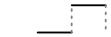
This circuit implements a "bernoulli gate". For each gate or trigger received at **`input`** there is made a random decision of whether to forward that gate to **`output1`** or **`output2`**. The probability for each of the outputs can be shifted with the parameter **`distribution`**. It determines the 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`** |  | **`0.5`** | 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. |

One **`bernoulli`** circuit needs **60** bytes of RAM.

### 10.5 `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   = O1
```

When a trigger arrives at **I1**, the output **O1** 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   = O1
```

Similar to the circuit **lfo** (see page 113), there is a third input for selecting the speed: **rate**. This works on a 1 V/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   = O1
```

**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   = O1
```

#### 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 = O1
```

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 199) 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 steady clock here and the burst will run at the speed of that clock - albeit optionally modified by **rate**. At least two clock ticks are needed for synchronisation, but always the last three ticks are averaged. |
| **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∘2∘3 | **1** | Number of triggers to send in one burst. |
| **skip** | 1∘2∘3 | **0** | Number of time slots to wait before starting with the burst. |

| Output | Type | Description |
|---|---|---|
| **output** |  | The triggers are output here. |

One **burst** circuit needs **160** bytes of RAM.

## 10.6 `button` - Doos 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 loose 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 **startvalue** for setting a start value. This make sense for the CV looper since the loop is apparently empty anyway when your DROID starts. By the way: **off** is a synonym for **0**.

```
[button]
    button      = B1.4
    led         = L1.4
    startvalue = off

[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 113). 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      = O1
```

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.

### 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.

They 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 71).

| 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** | 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 *exactly* **0** instead of a positive gate signal. In some cases this is more conventient. |
| **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 trigger here will reset the button to its start value (which is off, unless you have changed **startvalue**). |
| **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 *double click mode* when set to **1**. In that mode the button only toggles it's 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 clock mode is only makes sense if the number of states is 2. |
| **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** |  | ☞ | State of the push button when you switch on your system. Setting this to **on** or **off** will force the button into that state and ignore the setting that is saved on the SD card. If you have three states, the start value needs to be **0**, **1** or **2**. With four states, it can also be **3**. Using this input 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 state is **on**, a value of **1.0** will be sent to that output - regardless of the values in **onvalue** and **offvalue**. If the number of states is 3 or 4 the output get's 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 **L1.1**, **L1.2** and so on. |
| **output** |  | This output will output the current button states. This is usually **0** for off and **1** for on. If **states** is 3 or 4, the values **2** or **3** are output for the additional states. You can modify all four values with the inputs **offvalue**/**value1**, **onvalue**/**value2**, **value3** and **value4**. Note: if you haven't changed any of these inputs and **states** is unchanged or 1 or 2, the **led** output will output the same values. |
| **inverted** |  | The same as **output**, but sends **onvalue** when the button is off and **offvalue** when the button is on. If **states** is 3 or 4, the order of the four output values will be mirrored (probably a feature that is rarely of any use). |
| **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**. When **states** is 3 or 4, this output will be **1** if the button is off and **0** in the other three states. |
| **longpress** |  | Emits a trigger, when any button is pressed for at least 1.5 seconds. If this outputs is used, the effect of a short button press is delayed until the button is *released*. This will avoid double actions for long presses. |

One **button** circuit needs **232** bytes of RAM.

## 10.7 `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    = O1
```

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 buttons 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    = O1
```

### 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 149). 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 |
|---|---|---|---|
| **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 *exactly* **0** instead of a positive gate signal. In some cases this is more conventient. |
| **minactive** | 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 **output** will be set to **0.0** in that case. |
| **maxactive** | 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. |
| **button1 ... button32** |  | | $1^{st}$ ... $8^{th}$ 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 withing the allowed range. |
| **value1 ... value32** |  | ☞ | Value that will be sent to the output if the $1^{st}$ ... $32^{nd}$ button is active. These inputs default to **0** for **value1**, **1** for **value2** and so on and **31** for **value32**. |

| Output | Type | Description |
|---|---|---|
| **led1 ... led32** |  | This output will be on / 1.0, whenever the $1^{st}$ ... $8^{th}$ button is active and off / 0.0 otherwise. Wire this to the LED in the button. |
| **output** |  | The sum of the values of all active buttons will be sent here. if no button is active then 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 *either* at **buttonpress** or at **longpress**, not at both. |

One **buttongroup** circuit needs **1296** bytes of RAM.

## 10.8 `calibrator` - VCO Calibrator

### Introduction

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 are 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 149). 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 calibrator 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 |
|---|---|---|---|
| **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 *exactly* **0** instead of a positive gate signal. In some cases this is more conventient. |
| **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. |
| **reset** |  | | Resets all tunings to 0 - or to the values of the according **tune...** inputs if they are used. |
| **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 $\pm$ 1 Octave ( at a value of $\pm 12$). |
| **tunelowtail** |  | **0.0** | Tuning adaption for the negative voltage range. A value of 1 means an upwards tuning of one semitone *per octave*, -1 likewise downwards. |
| **tunehightail** |  | **0.0** | Tuning adaption for voltages $>$ 8 V. A value of 1 means an upwards tuning of one semitone *per octave*, -1 likewise downwards. |

| Output | Type | Description |
|---|---|---|
| **output** | $\frac{1V}{Oct}$ | The calibrated pitch goes out here. |
| **ledup** | 0  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. |
| **leddown** | 0  1 | This is the LED for **nudgedown**, which indicates downwards adjustment. |

One **`calibrator`** circuit needs **364** bytes of RAM.

## 10.9 `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. Or 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 149) 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♭ |
| **select5** | 5th | **V** | G | G |
| **select7** | 7th | **VII** | B | B♭ |
| **select9** | 9th = 2nd | **II** | D | D |
| **select11** | 11th = 4th | **IV** | F | F |
| **select13** | 13th = 6th | **VI** | A | A♭ |

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 = O1
    output2 = O2
    output3 = O3
    output4 = O4
```

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 = O1
    output2 = O2
    output3 = O3
    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 |
|---|---|---|---|
| **root** | 1∘2∘3 | **0** | Set the root note here. **0** means *C*, **1** means $C\sharp$, **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. |
| **degree** | 1∘2∘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. |
| **select1** |  | ☞ | Gate input for selecting the *root* note 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.<br><br>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. |
| **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 $6^{th}$). |
| **selectfill1** |  | **off** | Selects the alternative $9^{th}$ (i.e. the $9^{th}$ that is *not* in the scale. |
| **selectfill2** |  | **off** | Selects the alternative $3^{rd}$ (i.e. the $3^{rd}$ that is *not* 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 *not* in the scale). |
| **selectfill5** |  | **off** | Selects the alternative $7^{th}$ (i.e. the $7^{th}$ that is *not* in the scale). |

| Input | Type | Default | Description |
|---|---|---|---|
| **tuningmode** |  | **off** | While this is **1**, the circuit will output the value set by **tuningpitch** instead of the actual pitch. This is ment 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. |
| **pitch** | $\frac{1V}{Oct}$ | **0V** | This sets the minimum pitch of the lowest note of the chord. |
| **spread** | $\frac{1V}{Oct}$ | **0V** | Selects the range between the lowest and highest note of the chord. A spread of 0 means that all chord notes are within one octave. Positive values will be added to that, e.g. a spread value of 1 V (or 0.1) means that the chord will be spread out up to two octaves. |
| **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. |

| Output | Type | Description |
|---|---|---|
| **output1 ... output4** | $\frac{1V}{Oct}$ | $1^{st}$ ... $4^{th}$ pitch output |

One **chord** circuit needs **412** bytes of RAM.

### 10.10 `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. 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   = O1
    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   = O1
    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   = O1
    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    = O1
    dutycycle = 20% # same as 0.2
```

The CV **`delay`** can be used to delay the clock signal - assumed that the input clock is *steady*. 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    = O1
    delay     = 0.1 # same as 10%
```

Please note that this is *not* a trigger delay, since it requires a steady input clock. Otherwise funny and strange things can happen. But: in exchange for that limitation it can also shift a clock *ahead*. Using a small negative number will result in a clock that is always slightly *before* the original clock:

```
[clocktool]
    clock     = I1 # patch a clock here
    output    = O1
    delay     = -0.1
```

Feeding a trigger sequencer (like the **`algoquencer`**, see page 47) with a shifted clock allows you fine tuning 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
    ...
```

#### 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** | 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** | Number to multiply the clock with. Same considerations hold as for divide. |
| **dutycycle** |  | ☞ | 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** | This CV allows you to shift the *input* 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% *ahead*. But this is exactly the same as **0.9**. |

| 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 it 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 |

One **clocktool** circuit needs **216** bytes of RAM.

## 10.11 `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 = O1
```

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 = O1
```

### 10.11.1 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 = O1
```

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 fluctiations. 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 = O1 # 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 = O1
```

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 = O1
```

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 = O1
```

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 **O1** 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 = O1
```

| 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 non 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. |

One **compare** circuit needs **132** bytes of RAM.

## 10.12 contour - Contour generator

This circuit implements an enhanced version of the classic ADSR-envelope generator. It has 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 **O1**:

```
[contour]
    gate   = I1
    output = O1
```

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
    decay   = P1.2
    sustain = P2.1
    release = P2.2
    output  = O1
```

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**. If 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:

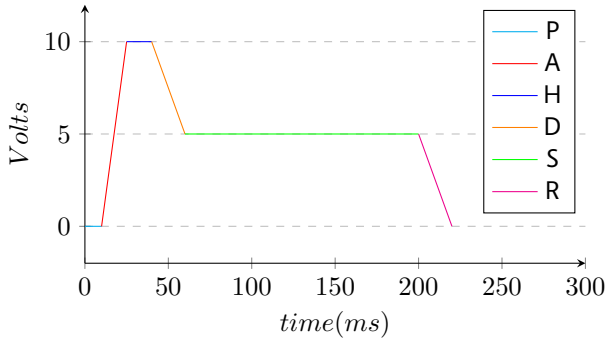
```
[p2b8]
[p2b8]

[contour]
    gate    = I1
    attack  = P1.1 * 20
    decay   = P1.2
    sustain = P2.1
    release = P2.2
    output  = O1
```

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

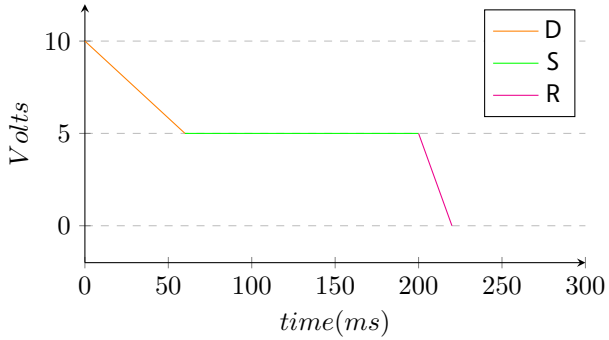
The phases attack, decay, release are phases where the level of the envelope starts at one level and then approaches another level within a certain time. In the upper example all these phases had a *linear* characteristic. That means that the output voltage changes by a constant amount per time.

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**.

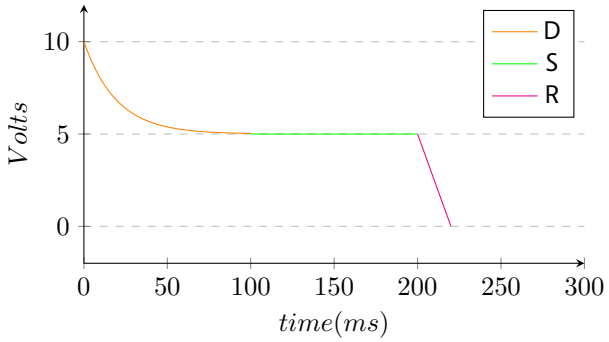
Let's take decay as an example. During the decay phase the envelopes voltage falls from the maximum level of 10 V (you can change this with the input **level**) to the sustain level defined by the input **sustain**. For simplicity let's assume that you have not used these inputs, so the maximum level 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** then 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**.



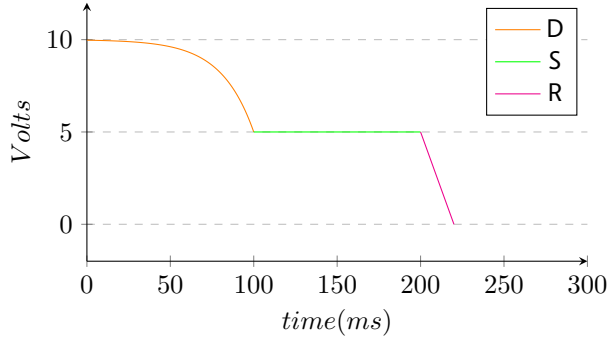

Now, if you set **decayshape** to **0.0** then curve is completely *exponential*:




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 **0.0** 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:

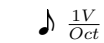
```
[contour]
    gate        = I1
    decayshape  = P1.1
    output      = O1
```

If the shape gets a value greater than **0.5** then 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 **1.0**:




| 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** then there are the following differences:<br>• The duration of the trigger signal is being ignored.<br>• There is no decay / sustain phase. Attack and hold are immediately followed by release. The inputs **sustain** and **decay** have no impact anymore.<br>• 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** then 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.<br><br>Note: This setting is only functional when the **gate** input is being used for triggering the envelope. If you use **trigger**, then 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.<br><br>**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 none-zero then the envelopes lingers a certain amount of time at its maximum level after the attack and before the decay phase. A value of **0.5** (this is 5 V) will create a hold time of 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**, then 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** | *energy* of the attack: 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 *accent* output here and add some offset. Sudden changes in this input will not affect the shape of the envelope. |
| **pitch** |  | **0V** | This is a *one volt per octave* 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 then nothing changes. Clocks faster than 120 BPM will speed up the envelope. Clocks slower than 120 BPM will slow it down. |
| **shape** |  | **0.5** | If you use this jack then 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 *not* 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 *positive* voltages but is inverted relative to the level of the envelope. When the normal output outputs 0 V, then 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. |

One **contour** circuit needs **512** bytes of RAM.

### 10.13 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 = O1

[copy]
    input = O1
    inverted = O2
```

| Input | Type | Default | Description |
|---|---|---|---|
| **input** |  | **0.0** | Connect the signal you want to copy here. |
| **minimum** |  | ☞ | This sets a lower limit to the input signal. If it falls below it will be set to this value. |
| **maximum** |  | ☞ | This sets a upper limit to the input signal. If it is above it will be set to this value. |

| Output | Type | Description |
|---|---|---|
| **output** |  | The resulting signal will be sent here. |
| **inverted** |  | An inverted version of the signal will be sent here (after **min** and **max** has been applied). Inverted means, that it is mirrored within the range of 0 … 1. For example the inversion of 0.2 is 0.8, the inversion of 0.5 is 0.5 and the inversion of 0.0 is 1.0. If you need a *negated* version, simply multiply the input by -1.0.<br><br>If the signal is negative, the inverted signal will also be negative and is now mirrored within the range -1 … 0. So the inversion of -0.8 is -0.2 and so on. |

One **copy** circuit needs **72** bytes of RAM.

### 10.14 `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 $0.5 \times$**I1** $+ 0.5 \times$**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 |

One **crossfader** circuit needs **168** bytes of RAM.

## 10.15 `cvlooper` - Clocked CV looper

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 **`O1`**. 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        = O5
    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 *not* the same as bypass, since in bypass mode the original CV will 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. |
| **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 *while keeping the loop's content untouched*. 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 |

One **`cvlooper`** circuit needs **17508** bytes of RAM.

### 10.16 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, **but3** 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 45), 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. |

One **dac** circuit needs **248** bytes of RAM.

### 10.17 `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.

| 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`** | 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.<br><br>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. |
| **`lpfilter1 ... lpfilter8`** |  | **`0.25`** | 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.<br><br>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.<br><br>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).<br><br>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. |

One **`droid`** circuit needs **288** bytes of RAM.

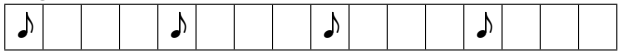
### 10.18 `euklid` - Euclidean rhythm generator

This circuit creates trigger patterns according to the well-known *Euclidean rhythms*. 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

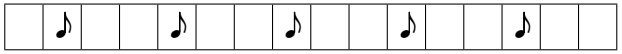
The number of beats are distributed as evenly as possible in the pattern - but of course are all placed precisely on clock beats. Here are a few examples of various patterns:

length: 16, beats: 4, offset: 0

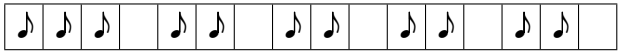


length: 16, beats: 5, offset: 0

length: 16, beats: 5, offset: 1



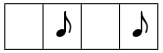
length: 16, beats: 11, offset: 0



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.

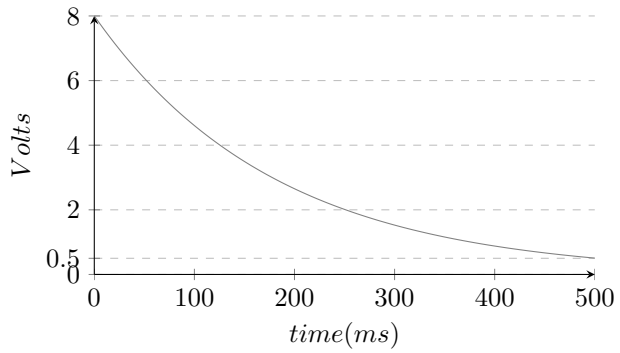
| Input | Type | Default | Description |
|---|---|---|---|
| **clock** |  | ☞ | Patch a clock signal here. It does not need to be steady - even if this is the most usual application. Note: this input is classified as a  gate input, since the length of the gate is being preserved when forwarded to **output** 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 pattner. The gate length is directly taken from the input clock - just as the voltage. |
| **offbeats** |  | Here those impulses will be output where there is *no* beat in the pattern. |

One **euklid** circuit needs **132** bytes of RAM.

### 10.19 `explin` - Exponential to linear converter

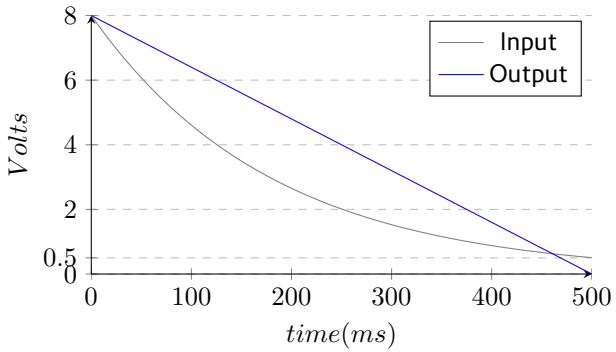
This circuit converts an exponential input curve into a linear output curve. Image you have an analog envelope outputting an exponential curve like the following one:




The curve starts at 8 V and reaches 0.5 V at about 500 ms later.

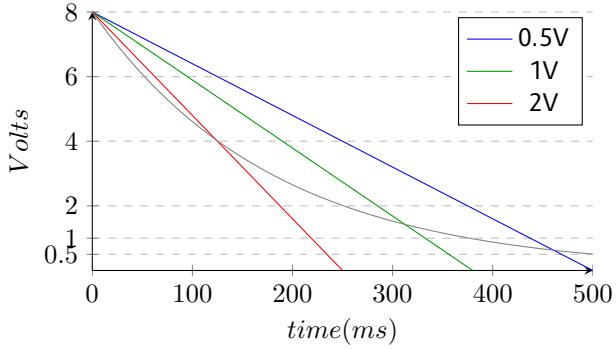
The following droid patch will convert this into a linear curve:

```
[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`** | 0 1 | **`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 used values $>$ **`1.0`**. A value of **`2.0`** will overcompensate and bend the curve beyond linearity into a curve some modularists would call *logarithmic*. |

| Output | Type | Description |
|---|---|---|
| **output** |  | Here comes the resulting linear output |

One **explin** circuit needs **84** bytes of RAM.

## 10.20 faderbank - Create multiple virtual faders in M4 controller

This circuit is very similar to **motorfader** (see page 156) but controls up to 16 faders at once. It's only purpose is to reduce the number of **motorfader** circuits in situations where you control banks or arrays of parameters in a similar way.

Please first check out the chapter about **motorfader** (see page 156). I won't repeat the same things here but just tell the differences. And these are:

- Instead of **fader** you set **firstfader** to the first fader to control.
- Instead of **output** you have **output1**, **output2** and so on. The number of outputs you use automatically 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.

Here is an example of a small fader bank of the three faders 3, 4 and 5 (spreading over two M4s). We use a pot to select 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 = O1
    output2 = O2
    output3 = O3
```

| Input | Type | Default | Description |
|---|---|---|---|
| **preset** | 1∘2∘3 | **0** | This is the preset number to save or to load. Note: the first preset has the number **0**, not **1**! This circuit has 6 presets, so this number ranges from **1** to **6**. |
| **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. |
| **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 *exactly* **0** instead of a positive gate signal. In some cases this is more conventient. |
| **firstfader** | 1∘2∘3 | **1** | First M4 fader of the virtual fader bank (starting with 1). |
| **notches** | 1∘2∘3 | | Number of artifical 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**. |
| **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. |

| 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. |

One **faderbank** circuit needs **1016** bytes of RAM.

### 10.21 `fadermatrix` - Matrix of 4x4 virtual motor faders

| Input | Type | Default | Description |
|---|---|---|---|
| **`preset`** | 1∘2∘3 | **`0`** | This is the preset number to save or to load. Note: the first preset has the number **`0`**, not **`1`**! This circuit has 6 presets, so this number ranges from **`1`** to **`6`**. |
| **`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. |
| **`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 *exactly* **`0`** instead of a positive gate signal. In some cases this is more conventient. |
| **`fader`** | 1∘2∘3 | **`1`** | |
| **`rowcolumn`** | 1∘2∘3 | | |
| **`notches1 ... notches4`** | 1∘2∘3 | | |
| **`ledcolor1 ... ledcolor4`** |  | | |
| **`ledvalue11 ... ledvalue14`** |  | | |
| **`ledvalue21 ... ledvalue24`** |  | | |
| **`ledvalue31 ... ledvalue34`** |  | | |
| **`ledvalue41 ... ledvalue44`** |  | | |

| Output | Type | Description |
|---|---|---|
| **`output11 ... output14`** |  | |
| **`output21 ... output24`** |  | |

| Output | Type | Description |
|---|---|---|
| **output31 ... output34** |  | |
| **output41 ... output44** |  | |
| **button11 ... button14** |  | |
| **button21 ... button24** |  | |
| **button31 ... button34** |  | |
| **button41 ... button44** |  | |

One **fadermatrix** circuit needs **1116** bytes of RAM.

### 10.22 `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.

| 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** | 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 *exactly* **0** instead of a positive gate signal. In some cases this is more conventient. |
| **outputlevel1 ... outputlevel16** | 0  1 | | |
| **mainoutput** | 1∘2∘3 | **1** | |
| **phonesoutput1, phonesoutput2** | 1∘2∘3 | | |
| **outputmix1in1 ... outputmix1in16** | 0  1 | | |
| **outputmix2in1 ... outputmix2in16** | 0  1 | | |
| **outputmix3in1 ... outputmix3in16** | 0  1 | | |
| **outputmix4in1 ... outputmix4in16** | 0  1 | | |
| **outputmix5in1 ... outputmix5in16** | 0  1 | | |
| **outputmixjin1 ... outputmixjin16** | 0  1 | | |

| Input | Type | Default | Description |
|---|---|---|---|
| **outputmix7in1 ... outputmix7in16** |  | | |
| **outputmix8in1 ... outputmix8in16** |  | | |
| **outputmix9in1 ... outputmix9in16** |  | | |
| **outputmix10in1 ... outputmix10in16** |  | | |
| **outputmix11in1 ... outputmix11in16** |  | | |
| **outputmix12in1 ... outputmix12in16** |  | | |
| **outputmix13in1 ... outputmix13in16** |  | | |
| **outputmix14in1 ... outputmix14in16** |  | | |
| **outputmix15in1 ... outputmix15in16** |  | | |
| **outputmix16in1 ... outputmix16in16** |  | | |
| **postfader1 ... postfader16** |  | | |
| **pan1 ... pan16** |  | | |
| **unmute1 ... unmute16** |  | | |
| **update** |  | | |

One **firefacecontrol** circuit needs **5640** bytes of RAM.

### 10.23 `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 substracted 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 |

One **fold** circuit needs **84** bytes of RAM.

### 10.24 `fourstatebutton` - Button switching through 4 states (OBSOLETE)

**This circuit has been superseded by the new circuit `button` (see page 67). `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 **`O1`** - 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  = O1
```

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 **`O1`**:

```
[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   = O1
```

| 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 ∘ 2 ∘ 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 |

One **`fourstatebutton`** circuit needs **140** bytes of RAM.

## 10.25 `lfo` - Low frequency oscillator (LFO)

This circuit implements a very flexible low frequency oscillator (LFO) with seven different waveforms, each of which is available at its own output as well as on a common output with waveform selection. It offers phase modulation, a flexible sync mechanism, randomization and other interesting features.

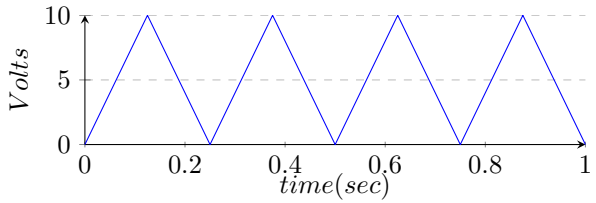
Please note also that this LFO is not intended to be used at audio rate. It can probably operate until roughly 1000-1500 Hz, but will sound ugly, distorted and with many digital artefacts - especially the waveforms with steep edges like square, ramp and sawtooth. If that's exactly what you intend, then 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`**:

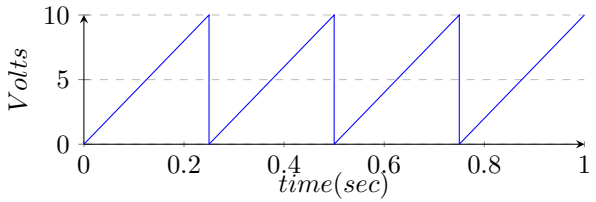
```
[lfo]
    hz       = 4
    triangle = 01
```

The resulting output looks like this:

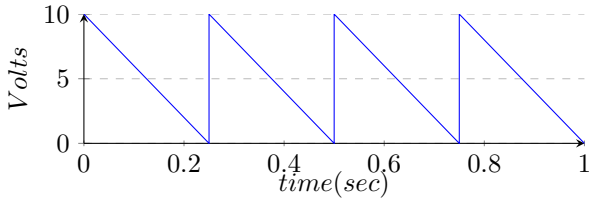



This is how the **`sawtooth`** output looks like:

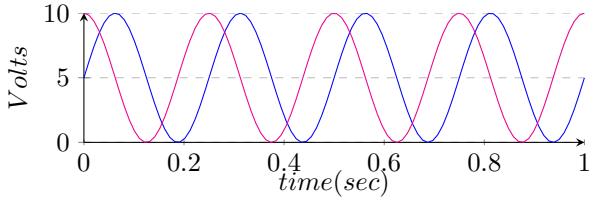
```
[lfo]
    hz       = 4
    sawtooth = 01
```



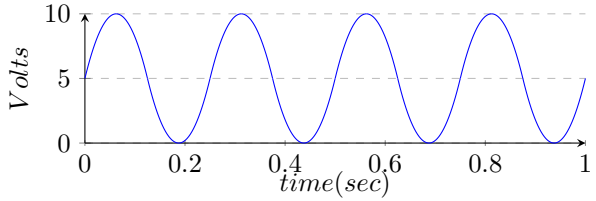

The **`ramp`** is similar but falling instead of rising:



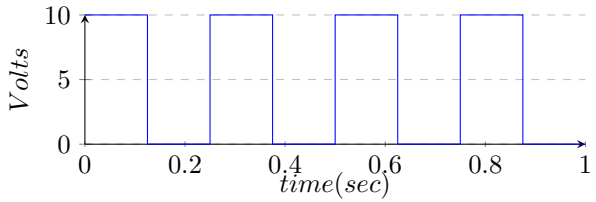

The waveforms **`sine`** and **`cosine`** are similar but are one quarter cycle (90°) apart:




**`paraboloid`** is *very* similar to sine, but is constructed based on quadratic equations (which is faster):



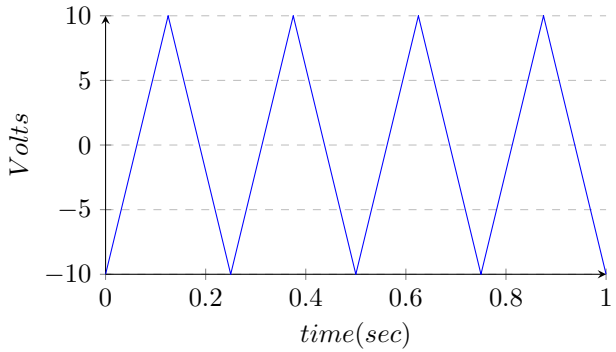

Maybe the simplest waveform is **`square`**:




### Bipolar output, Level and Offset

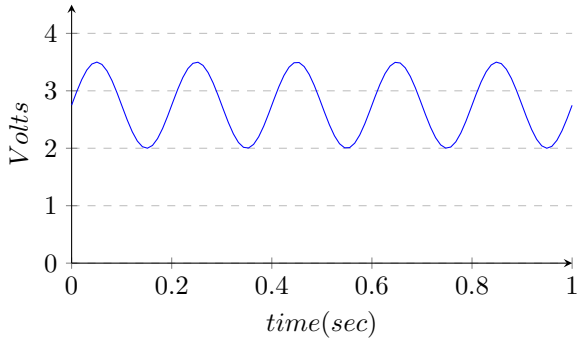
Please note that the LFO outputs just positive voltage ranges until you set **`bipolar = on`**. That extends the waveform to negative voltages (while doubling the peak-to-peak voltage):

```
[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 O4 that is gently oscillating between 2 V and 3.5V:

```
[lfo]
    hz          = 5
    level       = 1.5V
    offset      = 2V
    sine        = O4
```




### 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 (90 BPM):

```
[lfo]
    hz          = 1.5
    rectangle   = O3
```

A more eurorack-like way is using the **rate** input, which works on a 1 V / 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  = O1
```

The third way is to use *tap tempo* by sending a steady clock into **taptempo**. The LFO than 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    = O2
```

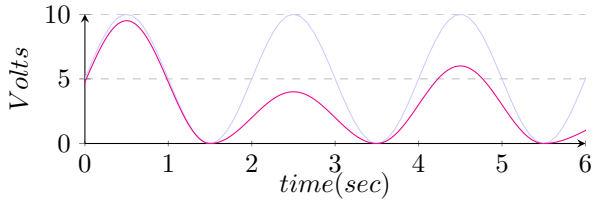
### Randomization

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        = O1
```

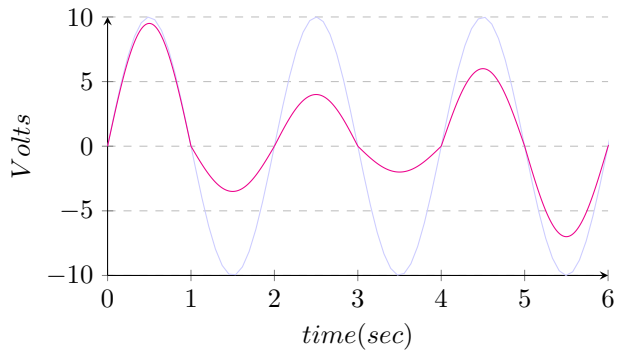
The original wave if printed light and the and the randomized wave at output **O1** is **magenta**:




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        = O1
    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 indiviual 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
```

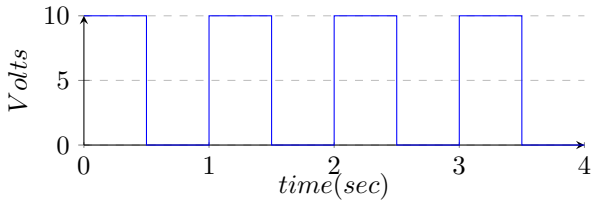
Here is the complete list of available waveforms:

| | |
|---|---|
| **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:

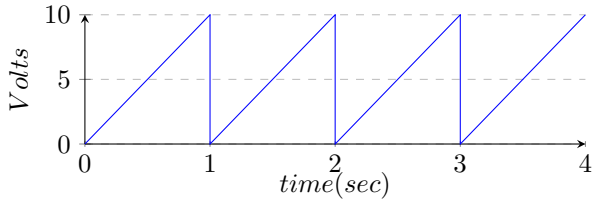
```
[lfo]
    hz = 4
    output = O1
    waveform = 0.0
```

And this is what it looks like:



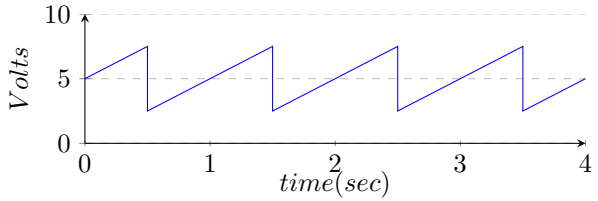

At **1.0** we get a saw tooth:

```
[lfo]
    hz = 4
    output = O1
    waveform = 1.0
```




And in between - at **0.5** - we get some mixture:

```
[lfo]
    hz = 4
    output = O1
    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 or 60 BPM if you like). Each volt doubles the frequency. So an input of 1 V (a number of **0.1**) speeds up the LFO to 2 Hz (120 BPM), 2 V (**0.2**) create 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 steady clock here and the LFO will run at the speed of that clock - albeit optionally modified by **rate**. |
| **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 *hill* 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. I 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 |
|---|---|---|---|
| **waveform** |  | **0.0** | If you use **output** - rather than the individual waveform outputs like **square**, **saw** and so on - this input selects the Wave form. 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 |
|---|---|---|
| **output** |  | Main output of the LFO. |
| **square** |  | A square waveform - modified by **pulsewidth**. |
| **sawtooth** |  | Outputs a sawtooth waveform - i.e. a rising ramp |
| **triangle** |  | Outputs a triangle waveform - modified by **skew**. |
| **ramp** |  | Outputs a falling ramp - like a sawtooth that is mirrored. Note: if the LFO is set to bipolar then this is the negation of **sawtooth**. If it is set to unipolar then this is not the case. The waveform will be positive then! |
| **paraboloid** |  | 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 **level**. |
| **sine** |  | A sine waveform. |
| **cosine** |  | 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 **sine** and **cosine** at a later stage. |

One **lfo** circuit needs **428** bytes of RAM.

## 10.26 `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 = O1
```

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 |
|---|---|---|---|
| **input1 ... input8** |  | ☞ | 1st ... 8th input. Note: this input is declared as a  gate input, but in fact you can use it as a CV input in combination with various or random values set for the **threshold**. |
| **threshold** |  | **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. |
| **lowvalue** |  | **0.0** | Output value that is output for logic low, false or **off**. |
| **highvalue** |  | **1.0** | Output value that is output for a logic high, true or **on**. |
| **countvalue** |  | **0.1** | Value added to the **count** output for each input with a high level |

| Output | Type | Description |
|---|---|---|
| **and** |  | 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** |
| **or** |  | A logic OR operation on all patched inputs: This output is set to **highvalue** if at least one of the inputs is high |
| **xor** |  | 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. |
| **nand** |  | Like AND but the outcome is negated. |
| **nor** |  | Like OR but the outcome is negated. |
| **negated** |  | Logical negate of **input1** (which can abbreviated as **input**). Note: The inputs **input2** ... **input7** are ignored here. Another note: If you use **input1** anyway, **negated** always outputs exactly the same as **nand** and **nor**. It's just more convenient to write and easier to understand. Hence a dedicated output for a logic negate. |
| **count** |  | Adds **countvalue** to this output for each input that is high. |
| **countlow** |  | Adds **countvalue** to this output for each input that is low. |

One **logic** circuit needs **240** bytes of RAM.

### 10.27 `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 $\frac{\mathbf{I1}}{\mathbf{I2}}$:

```
[math]
    input1   = I1
    input2   = I2
    quotient = O1
```

Example for computing the square root of **`I1`**:

```
[math]
    input    = I1
    root     = O1
```

Note: As long as you do not send a value directly to an output like **`O1`**, 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 ... +1 (which corresonds to -10 V ... +10 V).

| Input | Type | Default | Description |
|---|---|---|---|
| **`input1, input2`** |  | | The two inputs |

| Output | Type | Description |
|---|---|---|
| **`sum`** |  | **`input1`** $+$ **`input2`** |
| **`difference`** |  | **`input1`** $-$ **`input2`** |
| **`product`** |  | **`input1`** $\times$ **`input2`** |
| **`quotient`** |  | **`input1`** $/$ **`input2`**. If **`input2`** 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. |
| **`modulo`** |  | **`input1`** modulo **`input2`**. This needs some explanation: With this operation you can "fold" the value from **`input1`** into the range 0 ... **`input2`**. For example if **`input2`** 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 **`input2`** is zero or negative, the output will be zero. |
| **`power`** |  | **`input1`** to the power of **`input2`**. 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 **`math`** circuit behaves in the following way:<br>• If **`input1`** $< 0$, **`input2`** is rounded to the nearest integer.<br>• If **`input1`** $= 0$ and **`input2`** $< 0$, a very large number is output. |
| **`average`** |  | The average of **`input1`** and **`input2`** |

| Output | Type | Description |
|---|---|---|
| **maximum** |  | The maximum of **input1** and **input2** |
| **minimum** |  | The minimum of **input1** and **input2** |
| **negation** |  | $-$**input1** |
| **reciprocal** |  | $1 \,/\,$ **input1**. If **input1** is zero, a very large number is being output, while the sign is being kept. |
| **amount** |  | The absolute value of **input1** (i.e. $-$**input1** if **input1** $< 0$, else **input1**) |
| **sine** |  | The sine of **input1** in a way, the input range of 0.0 … 1.0 goes exactly through one wave cycle. Or more mathematically expressed: $sin(2\pi \times$ **input1**$)$. |
| **cosine** |  | The cosine of **input1** in a way, the input range of 0.0 … 1.0 goes exactly through one wave cycle. Or more mathematically expressed: $cos(2\pi \times$ **input1**$)$. |
| **square** |  | **input1**$^2$ |
| **root** |  | $\sqrt{\textbf{input1}}$. Please note that you cannot compute the square root of a negative number. In order to output something useful anyway, the result will be $-\sqrt{-\textbf{input1}}$, if **input1** $< 0$. |
| **logarithm** |  | The natural logarithm of **input1**: $ln$ **input1**. The logarithm is only defined for positive numbers. **mathcircuit** behaves like this:<br>• If **input1** $= 0$, a negative very large number is output.<br>• If **input2** $< 0$, $-ln$ **input1** is output. |
| **round** |  | The integer number nearest to **input1** |
| **floor** |  | The largest integer number that is not greater than **input1** |
| **ceil** |  | The smallest integer number that is not less than **input1** |

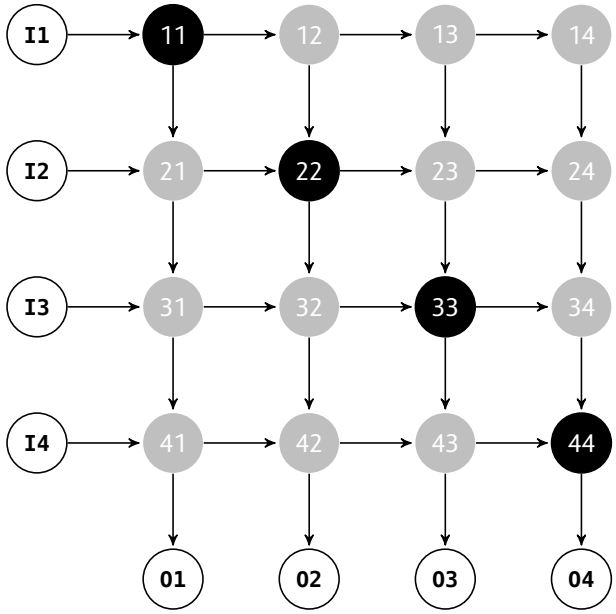
One **math** circuit needs **128** bytes of RAM.

## 10.28 `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:



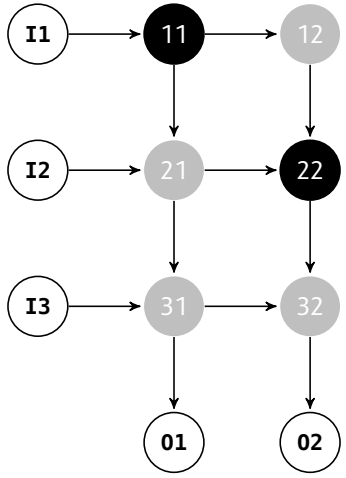

As an alternative operation, instead of summing the enabled signals you can compute the *maximum* signal. This is useful when combining envelope signals - e.g. from different rhythmic patterns. Adding envelope signals would either make them "too loud" or even distort them.

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  = O1
    output2  = O2
    button11 = B1.1
    button12 = B1.2
    button21 = B2.1
    button22 = B1.3
    button31 = B1.4
    button32 = B2.3
    led11    = L1.1
    led12    = L1.2
    led21    = L2.1
    led22    = L1.3
    led31    = L1.4
    led32    = L2.3
```

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 |
|---|---|---|---|
| **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 *exactly* **0** instead of a positive gate signal. In some cases this is more conventient. |
| **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 outputs is the maximum of the enabled inputs. Any number in between will create a weighted average between these two values. |
| **startvalue** | 1∘2∘3 | ☞ | If you use this input, persisting the state on the SD card is disabled. Rather the matrix starts with a standard configuration of which buttons are active. This configuration is set by the value of **startvalue**:<br><br>**0** All buttons are cleared<br>**1** The buttons on the diagonal are active, so **input1** is sent to **output1**, **input2** to **output2** and so on.<br>**2** All buttons are set<br><br>If you don't use this jack, the matrix starts for the very first time with the buttons of the diagonal being active and saves the status to the SD card from then on. |
| **reset** |  | | A trigger here resets the matrix to its initial state, which can be configured with **startvalue**. |
| **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. |

| Input | Type | Default | Description |
|---|---|---|---|
| **button41 ... button44** |  | | These four buttons decide, to which of the four outputs **input4** is being mixed. |

| 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** |

One **matrixmixer** circuit needs **576** bytes of RAM.

## 10.29 `midifileplayer` - MIDI file player

### Introduction

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 = O1
    gate = O2
```

Now patch **`O1`** to the 1V/Oct of a synth voice and **`O2`** 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 **pitch**$X$ and a **gate**$X$ output (and an optional **velocity**$X$ 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 = O1
    pitch2 = O2
    pitch3 = O3
    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 = O1
    gate = O2
    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 $16^{th}$ note's time equivalent of content:

```
[midifileplayer]
    pitch = O1
    gate = O2
    clock = G1
```

Note: this does *not* mean that the notes are quantized to $16^{th}$ 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 = O1
    gate = O2
    error = R1
```

Please see the table of outputs below for the various errors and their color codes.

| Input | Type | Default | Description |
|---|---|---|---|
| **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 it's 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. |
| **roundrobin** |  | **0** | 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.<br><br>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.<br><br>Note: When all outputs are currently used by a note, **roundrobin** has no influence. Here **voiceallocation** selects which of the notes will be dropped. |

<table>
<tr><th>Input</th><th>Type</th><th>Default</th><th>Description</th></tr>
<tr><td>voiceallocation</td><td>1◦2◦3</td><td>0</td><td>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:<br><br><table><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></td></tr>
<tr><td>notegap</td><td>  </td><td>0.0</td><td>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”.<br><br>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.<br><br>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.<br><br>Note: the notegap parameter does not affect the trigger outputs.</td></tr>
<tr><td>ccnumber1 ... ccnumber4</td><td>1◦2◦3</td><td>0</td><td>You can listen 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.</td></tr>
<tr><td>lowestnote</td><td>1◦2◦3</td><td>0</td><td>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.</td></tr>
<tr><td>highestnote</td><td>1◦2◦3</td><td>127</td><td>Sets an upper limit to the note being played, similar to lowestnote. The “Notegates” are not being affected by this bound.</td></tr>
<tr><td>note1 ... note16</td><td>1◦2◦3</td><td>☞</td><td>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 hold. One application is to use just one midifileplayer or midiin 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.</td></tr>
</table>

| 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 *not* 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**. |

| Output | Type | Description |
|---|---|---|
| **pitch1 ... pitch8** |   | 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 **pitch***X* outputs, wire the individual **pressure***X* 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. |

| Output | Type | Description |
|---|---|---|
| **trigger1 ... trigger8** |  | Trigger outputs for the up to eight simultaneous note outputs. The difference to the gate outputs is, that these just send a short trigger of 5 ms at the start of the note. This can be interesting in situations where the notes have no *gaps* in between so that gate will never go low. |
| **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 *MIDI program change* 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 *change*. |
| **program** |  | The number of the last program change. This starts at **0**. |
| **bank** |  | 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. |
| **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 limiter for creating portamento effects (see page 182). |
| **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. |
| **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. |

| Output | Type | Description |
|---|---|---|
| **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**: |

| value | color | what happened? |
|---|---|---|
| **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). |

One **midifileplayer** circuit needs **7132** bytes of RAM.

## 10.30 `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 26 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 = O1
    gate = O2
```

It's really as simple as that! Connect your MIDI keyboard or sequencer with the X7 MIDI input, wire **O1** to the 1V/Oct input of a synth voice and **O2** 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 = O1
    pitch2 = O2
    pitch3 = O3
    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 = O1
    notegate2 = O2
```

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: **midiout** 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 = O1
```

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 82) 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 = O1
    modwheel = O2
    ccnumber1 = 10 # get update from CC#10
    cc1 = O3 # send current CC value to O3
```

### 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 food 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 182).
- *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 consequtive notes together by keeping **gate** at **1** between notes.

| Input | Type | Default | Description |
|---|---|---|---|
| **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 it's 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. |
| **roundrobin** |  | **0** | 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.<br><br>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.<br><br>Note: When all outputs are currently used by a note, **roundrobin** has no influence. Here **voiceallocation** selects which of the notes will be dropped. |

<table>
<tr><th>Input</th><th>Type</th><th>Default</th><th>Description</th></tr>
<tr><td>voiceallocation</td><td>1◦2◦3</td><td>0</td><td>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:<br><br><table><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></td></tr>
<tr><td>notegap</td><td>  </td><td>0.0</td><td>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”.<br><br>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.<br><br>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.<br><br>Note: the notegap parameter does not affect the trigger outputs.</td></tr>
<tr><td>ccnumber1 ... ccnumber4</td><td>1◦2◦3</td><td>0</td><td>You can listen 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.</td></tr>
<tr><td>lowestnote</td><td>1◦2◦3</td><td>0</td><td>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.</td></tr>
<tr><td>highestnote</td><td>1◦2◦3</td><td>127</td><td>Sets an upper limit to the note being played, similar to lowestnote. The “Notegates” are not being affected by this bound.</td></tr>
<tr><td>note1 ... note16</td><td>1◦2◦3</td><td>☞</td><td>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 hold. One application is to use just one midifileplayer or midiin 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.</td></tr>
</table>

| 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. |
| **channel** | 1∘2∘3 | ☞ | Select the MIDI channel to listen on. Default is to listen on *all* channels - and basically ignore the channel number. There are 16 channels, numbered from 1 to 16. |
| **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. |

| Output | Type | Description |
|---|---|---|
| **pitch1 ... pitch8** |  | 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 **pitch***X* outputs, wire the individual **pressure***X* 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. |
| **trigger1 ... trigger8** |  | Trigger outputs for the up to eight simultaneous note outputs. The difference to the gate outputs is, that these just send a short trigger of 5 ms at the start of the note. This can be interesting in situations where the notes have no *gaps* in between so that gate will never go low. |
| **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**. |

| Output | Type | Description |
|---|---|---|
| **programchange** |  | Sends a trigger whenever a *MIDI program change* 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 *change*. |
| **program** |  | The number of the last program change. This starts at **0**. |
| **bank** |  | 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. |
| **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 limiter for creating portamento effects (see page 182). |
| **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. |
| **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. |

One **`midiin`** circuit needs **1240** bytes of RAM.

### 10.31 `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 26).

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.

#### 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 $16^{th}$ 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 ment to be a quarter note. 120 quarter notes a minute means two quarter notes a second and that means eight $16^{th}$ 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 $16^{th}$ 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 artifical clocks 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.

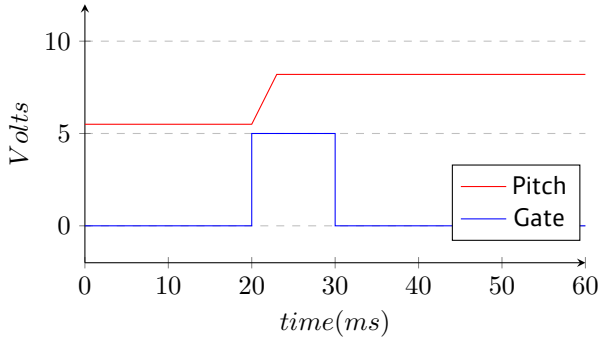
### Pitch tracking

Pitch tracking is an advanced feature that works in monophonic setups. Here **midiout** 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 **midiout** 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 *moving*. 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:

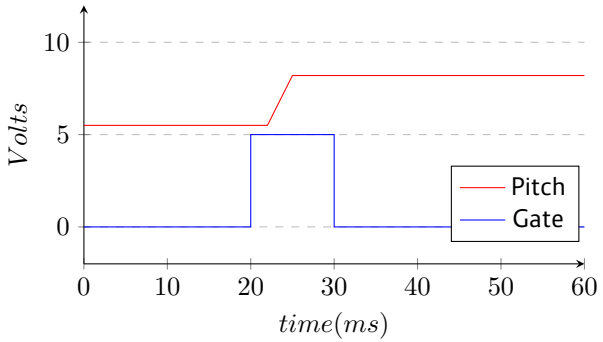
```
[midiout]
    pitch = I1
    gate = I2
    pitchstabilization = 0
```

The second method is introducing a fixed delay of the gate signal with the input **triggerdelay**. Using that parameter automatically disables pitch stabilization:

```
[midiout]
    pitch = I1
    gate = I2
    triggerdelay = 3.5 # delay gate by 3.5 ms
```

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**:

```
[midiout]
    pitch = I1
    gate = I2
    triggerdelay = 2
    pitchstabilization = 1
```

| 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** | 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 *exactly* **0** instead of a positive gate signal. In some cases this is more conventient. |
| **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** | If **usb = 0**, selects the TRS (3.5mm stereo jack) port of the X7 to send on. This is the default. Set **usb = 1** for sending the MIDI data via the USB-C port. |

| Input | Type | Default | Description |
|---|---|---|---|
| **pitch1 ... pitch8** |  | **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** |  | **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** |  | ☞ | MIDI also sends a velocity at the *end* of a note. The idea is to model the speed with which a key is being *released*. 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** |  | ☞ | 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 *per key*.<br><br>If nothing is patched here, no pressure events are sent. |
| **channelpressure** |  | ☞ | 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.<br><br>If nothing is patched here, no channel pressure events are sent. |
| **pitchstabilization** |  | **1** | Enables or disables pitch stabilization. It is on per 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. |
| **triggerdelay** |  | **0.0** | Introduces a delay between in 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 *before* the pitch stabilization. So it is a *minimum* 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. |

| Input | Type | Default | Description |
|---|---|---|---|
| **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 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 use by the notegates. In order to keep it simple, this velocity is used for note on *and* 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 to 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 *all* 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 *not* behave in a 1V/oct way! |
| **pitchtracking** | 1∘2∘3 | **0** | Pitch tracking is an advanced feature that allows you to track continuous changes in the incoming pitch CV *while the note is already playing*. It does this by listening to the input CV and converting any change into a MIDI "pitch bend" change.<br><br>This feature has two limitations:<br>1. 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.<br>2. 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.<br><br>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.<br><br>Note: When you use pitch tracking at the same time as **pitchbend**, both pitch alterations will add up. |

| Input | Type | Default | Description |
|---|---|---|---|
| **pitchbendrange** | ♪ $\frac{1V}{Oct}$ | $\frac{1}{6}$**V** | Defines the range of the effect of pitch bend at the target device on a 1V/oct base. Note: You cannot *change* that actual range here. You just can make sure that this circuit has the correct assumption of that range.<br><br>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 *no* effect on the **pitchbend** CV. |
| **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** | 0 ◠ 1 | ☞ | 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).<br><br>If you don't patch anything here, no CC events will be sent, of course. |
| **cctrigger1 ... cctrigger8** | ⎍ | | Usually **midiout** will send out a new CC event every time the input value of a CC has changed (with some rate limit in order to to flood the MIDI stream).<br><br>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.<br><br>This is useful for target devices that use CCs just as *messages*, i.e. as one time events and not for updating a continous value. |
| **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 |
|---|---|---|---|
| **running** |  | | This is an alternative to the jacks **start** and **stop**. 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. |
| **systemreset** |  | | A trigger here will send the MIDI real-time message "RESET", that is supposed to bring the device into some start state. |
| **allnotesoff** |  | | A trigger here will send the MIDI CC#123 "ALL NOTES OFF", which is essentially the same as releasing all currently held keys. |
| **allsoundoff** |  | | A trigger here will send the MIDI CC#120 "ALL SOUND OFF", which is supposed to make the device silent as soon as possible. |
| **damper** |  | **0** | 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 those notes* as long as the pedal is held. When the pedal is released, all notes that had been held be the pedal should be released. |
| **portamento** |  | **0** | Controls the portamento pedal. The receiver is meant to activate some kind of glide effect as long as this gate is high. |
| **sostenuto** |  | **0** | 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. |
| **soft** |  | **0** | Controls the soft pedal. The receiving synth voice is meant to play notes softer while this pedal is hold down. |
| **legato** |  | **0** | Controls the legato pedal, which ties subsequent notes together. |
| **clock** |  | | If you feed a steady clock here, a MIDI clock signal will be derived from this and sent through the output wire. The *MIDI beat clock* or simply *MIDI clock* 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 **clock** jack is meant to retrieve a modular master clock, multiplies this by 6 and creates a MIDI clock from it. |
| **midiclock** |  | | This is an alternative to **clock**: don't use both at the same time. Here you can directly send the MIDI clock in 24 PPQN. |
| **activesensing** |  | **1** | This is a switch that disables or enabled **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 (und turn all sound off) if these messages stop. Active sensing is enabled per default. You can disable it here by setting **activesensing = 0**. |

| Input | Type | Default | Description |
|---|---|---|---|
| **updaterate** |  | **50.0** | 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.<br><br>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. |

One **midiout** circuit needs **2680** bytes of RAM.

### 10.32 `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 **midiin** (see page 133) and **midiout** (see page 140) 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 **midiout** 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** |  | **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** |  | **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. |

One **midithrough** circuit needs **232** bytes of RAM.

### 10.33 `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 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 58) and **chord** (see page 78).

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 (♭6 but ♯7) |
| **9** | phr | Phrygian minor scale (with ♭9) |
| **10** | dim | Diminished scale (whole/half tone) |
| **11** | aug | Augmented scale (just whole tones) |

If you are a Sinfonion user, please note that the inputs **root** and **degree** of the Minifonion are *not* based on semitones like the Sinfonion, but simply expect whole numbers like **0**, **1**, **2** and so on (which corresponds to the CVs 0V, 10V, 20V, etc.). So if you want those CV inputs to be compatible, you have to multiply the values with the factor of 120 before sending them to the Minifonion:

```
[minifonion]
    input  = I1
    output = O2
    root   = I2 * 120 # base on semitones
    degree = I3 * 120 # base on semitones
```

| Input | Type | Default | Description |
|---|---|---|---|
| **root** | 1∘2∘3 | **0** | Set the root note here. **0** means *C*, **1** means $C\sharp$, **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. |
| **degree** | 1∘2∘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. |
| **select1** |  | ☞ | Gate input for selecting the *root* note 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.<br><br>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. |
| **select3** |  | ☞ | Gate input for selecting the 3rd. |
| **select5** |  | ☞ | Gate input for selecting the 5th. |
| **select7** |  | ☞ | Gate input for selecting the 7th. |
| **select9** |  | ☞ | Gate input for selecting the 9th (which is the same as the 2nd). |
| **select11** |  | ☞ | Gate input for selecting the 11th (which is the same as the 4th). |
| **select13** |  | ☞ | Gate input for selecting the 13th (which is the same as the 6th). |
| **selectfill1** |  | **off** | Selects the alternative 9th (i.e. the 9th that is *not* in the scale. |
| **selectfill2** |  | **off** | Selects the alternative 3rd (i.e. the 3rd that is *not* in the scale). |
| **selectfill3** |  | **off** | Selects the alternative 4th or 5th. In most cases this is the diminished 5th. |
| **selectfill4** |  | **off** | Selects the alternative 13th (i.e. the 1st3 that is *not* in the scale). |
| **selectfill5** |  | **off** | Selects the alternative 7th (i.e. the 7th that is *not* 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 ment 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. |
| **input** | ♪ $\frac{1V}{Oct}$ | **0V** | Patch the unquantized input voltage here |

| Input | Type | Default | Description |
|---|---|---|---|
| **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 *scale* 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 *selected* scale notes up or down (if negative). If you use **noteshift** at the same time, *first* **selectnoteshift** is applied, then **noteshift**. **selectnoteshift** is applied when quantization takes places, so it also is sensible to the **trigger** input. |

| Output | Type | Description |
|---|---|---|
| **output** |  | 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. |

One **minifonion** circuit needs **400** bytes of RAM.

### 10.34 `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 |
|---|---|---|---|
| **`input1 ... input8`** |  | **`0.0`** | $1^{st}$ ... $8^{th}$ mixing input |

| Output | Type | Description |
|---|---|---|
| **`output`** |  | Sum of all patched inputs |
| **`maximum`** |  | 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. |
| **`minimum`** |  | Minimum of all patched inputs of this circuit. |
| **`average`** |  | Average of all patched inputs of this circuit. |

One **`mixer`** circuit needs **160** bytes of RAM.

### 10.35 `motoquencer` - Sequencer using motor faders (EXPERIMENTAL)

Please note: This circuit is still experimental. It's not complete. It might be replaced by something different. Also the M4 controller, which is needed for this circuit, is not yet available. So please be a bit patient...

| Input | Type | Default | Description |
|---|---|---|---|
| **root** | 1∘2∘3 | **0** | Set the root note here. **0** means *C*, **1** means $C\sharp$, **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. |
| **degree** | 1∘2∘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. |
| **select1** |  | ☞ | Gate input for selecting the *root* note 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.<br><br>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. |
| **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 $6^{th}$). |
| **selectfill1** |  | **off** | Selects the alternative $9^{th}$ (i.e. the $9^{th}$ that is *not* in the scale. |
| **selectfill2** |  | **off** | Selects the alternative $3^{rd}$ (i.e. the $3^{rd}$ that is *not* 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 *not* in the scale). |
| **selectfill5** |  | **off** | Selects the alternative $7^{th}$ (i.e. the $7^{th}$ that is *not* 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 ment to be a help for tuning your VCOs. |

| Input | Type | Default | Description |
|---|---|---|---|
| **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. |
| **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 *exactly* **0** instead of a positive gate signal. In some cases this is more conventient. |
| **faderblock1, faderblock2** | 1∘2∘3 | | |
| **clock** |  | | |
| **pitchbase** |  | **0.0** | |
| **pitchrange** |  | **0.2** | |
| **bar** | 1∘2∘3 | **1** | |
| **stepoffset** | 1∘2∘3 | **0** | |
| **autoreset** | 1∘2∘3 | **0** | |
| **fadermode** | 1∘2∘3 | **0** | |
| **buttonmode** | 1∘2∘3 | **0** | |
| **quantize** |  | **1** | |
| **snapshot** |  | | |
| **recall** |  | | |
| **clear** |  | | |
| **cvdefault1 ... cvdefault3** | 1∘2∘3 | | |

| Output | Type | Description |
|---|---|---|
| **pitch** |  | |
| **gate1 ... gate3** |  | |
| **cv1 ... cv3** |   | |

One **motoquencer** circuit needs **1056** bytes of RAM.

## 10.36 `motorfader` - Create virtual fader in M4 controller

The traditional way of using motor faders is that you have several *presets* (i.e. different fader positions) and switch between them for example with buttons. In the digital domain, however, there is a second way to use the faders: to assign multiple *independent functions* to one fader. For example one single fader could control attack, decay, sustain and release of an envelope. The DROID motor faders are designed to do both.

The most basic and elementary way to use faders in your patch is using the **motorfader** circuit. When you are creating patches with banks of many faders, please also have a look at **faderbank** (see page 104) and **fadermatrix** (see page 106). Those circuits manage a collection of faders with a single circuit.

### Presets

Let's make a simple example for using presets with motor faders. We use one P2B8 and one M4 controller:

```
[p2b8]
[m4]
```

We use the first fader as a simple CV source to be output on **O1**. And four buttons should select four different presets of that fader. Those are grouped into a button with the circuit **buttongroup** (see page 71):

```
[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 = O1
```

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 **O1** 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 = O1

[motorfader]
    fader = 1
    select = _FUNCTION
    selectat = 1
    output = O2

[motorfader]
    fader = 1
    select = _FUNCTION
    selectat = 2
    output = O3

[motorfader]
    fader = 1
```

```
    select = _FUNCTION
    selectat = 3
    output = O4
```

As you can see: each fader has a **selectat** input matching 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 = O1

[motorfader]
    fader = 1
    select = L1.2
    output = O2

[motorfader]
    fader = 1
    select = L1.3
    output = O3

[motorfader]
    fader = 1
    select = L1.4
    output = O4
```

### 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 artifical "notches" or "dents". Try that out be 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!

The maximum number of notches is **25**.

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.

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 |
|---|---|---|---|
| **preset** | 1∘2∘3 | **0** | This is the preset number to save or to load. Note: the first preset has the number **0**, not **1**! This circuit has 8 presets, so this number ranges from **1** to **8**. |
| **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. |
| **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 *exactly* **0** instead of a positive gate signal. In some cases this is more conventient. |
| **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** |  | | If you use this input, the virtual fader will start with this value when you start your DROID. Also it will disabling saving the virtual fader state to the SD card. |
| **notches** | 1∘2∘3 | **0** | Number of artifical 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 it's output with the next **motorfader** circuit. |

| Output | Type | Description |
|---|---|---|
| **output** |  | Output the current value if the virtual motor fader (don't use this if you are using **sharewithnext**). |

One **motorfader** circuit needs **248** bytes of RAM.

### 10.37 `notchedpot` - Helper circuit for pots (OBSOLETE)

**This circuit has been superseded by the new circuit `pot` (see page 169). `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  = O1 # Send -10V ... +10V to O1
```

| Input | Type | Default | Description |
|---|---|---|---|
| **`pot`** | 0 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`** | 0 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. |

One **`notchedpot`** circuit needs **76** bytes of RAM.

### 10.38 `notebuttons` - Note Selection Buttons

This simple utility combines 12 buttons, just like radio buttons, into a selector for a note such as C, $C\sharp$, 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 |
|---|---|---|---|
| **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 *exactly* **0** instead of a positive gate signal. In some cases this is more conventient. |
| **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. |

| 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. |

One **notebuttons** circuit needs **328** bytes of RAM.

### 10.39 nudge - Modify - "nudge" - a value 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     = O1
```

**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     = O1
    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     = O1
    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 **amount**s.

| 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** | 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 *exactly* **0** instead of a positive gate signal. In some cases this is more conventient. |
| **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 **reset** 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. |
| **reset** |  | | A trigger here will reset the value to its start value |
| **persist** |  | **1** | Set this to **0** if you do not like the current value to be saved and reloaded from flash after a restart of your modular system. The default is **1**, which means that the current value will automatically saved. |

| Output | Type | Description |
|---|---|---|
| **ledup** |  | Wire this to the LED in the button for nuding up. It will indicate the current value. |
| **leddown** |  | Wire this to the LED in the button for nuding down. It will indicate the current value. |
| **output** |  | The output of the current value plus value if **offset**. |

One **nudge** circuit needs **228** bytes of RAM.

### 10.40 `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  = O1
    output2  = O2
    output3  = O3
    spread   = P1.1
    detune   = P1.2
```

Patch the 1 V / octave inputs of three VCOs at **O1**, **O2** and **O3**. 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 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  = O1
    output2  = O2
    output3  = O3
    spread   = _RANDOM * P1.1
```

Now **P1.1** controls the depth of random octave animation.

| Input | Type | Default | Description |
|---|---|---|---|
| **`input`** |  | **`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 *not* 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 ist 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. |

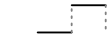
| Output | Type | Description |
|---|---|---|
| **`output1 ... output3`** |  | Outputs for the 1 V / octave of the three VCOs. **`output1`** is an exact copy of **`input`** so you could omit that and rather patch VCO 1 to the original pitch CV. |

One **`octave`** circuit needs **92** bytes of RAM.

### 10.41 `polytool` - Change number of voices in polyphonic setups

| Input | Type | Default | Description |
|---|---|---|---|
| **`pitchinput1 ... pitchinput16`** | ? | | The pitches of up to 16 input voices. |
| **`gateinput1 ... gateinput16`** |  | | The gates of up to 16 input voices. |
| **`roundrobin`** |  | **`0`** | 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.<br><br>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.<br><br>Note: When all outputs are currently used by a note, **`roundrobin`** has no influence. Here **`voiceallocation`** selects which of the notes will be dropped. |
| **`voiceallocation`** | 1∘2∘3 | **`0`** | 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:<br><br>**`0`** The oldest note will be cancelled (default)<br>**`1`** The new note will not be played and simply be omitted<br>**`2`** The lowest note will be cancelled<br>**`3`** The highest note will be cancelled |

| Output | Type | Description |
|---|---|---|
| **`pitchoutput1 ... pitchoutput16`** | ? | The pitches of up to 16 output voices. |

| Output | Type | Description |
|---|---|---|
| **gateoutput1 ... gateoutput16** |  | The gates of up to 16 output voices. |

One **polytool** circuit needs **780** bytes of RAM.

## 10.42 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  = O1 # Send -10V ... +10V to O1
```

Have a look into the table of jacks below about further useful things like splitting the pot's way in two halfs.

### Center notch

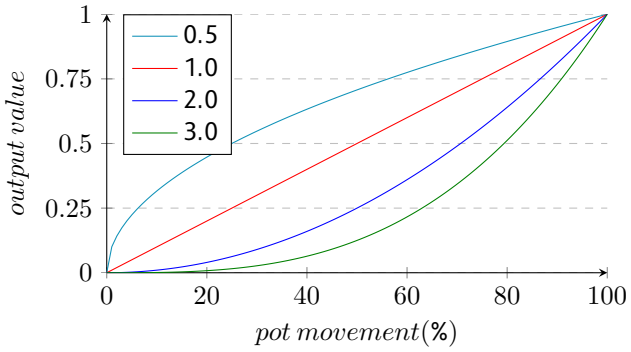
**pot** can simulate a potentiometer with a notch at the center. It helps to exactly select the center position by defining a "range of tolerance" that is considered to be the center. This range is called "notch" and is given in a percentage of the available range. I suggest using 10% so you don't loose to 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 "streched 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.

### Sharing 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 71):

```
[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 = O1

[contour]
    trigger = I2
    release = _RELEASE2
    output = O2

[contour]
    trigger = I3
    release = _RELEASE3
    output = O3

[contour]
    trigger = I4
    release = _RELEASE4
    output = O4
```

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.

### Picking up the pots

Pots are no motorized faders and no encoders. So when reusing a pot for more than one function at a time there is always the problem when set to one pot function the pot ist likely 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 **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**.

We solve this 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.

### Splitting the pot into two hemisperes

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 = O1
    righthalf = O2
```

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 **O1**) 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.

| 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 *exactly* **0** instead of a positive gate signal. In some cases this is more conventient. |
| **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**. |
| **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 |
|---|---|---|---|
| **ledgauge** |  | ☞ | 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 **select** 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.<br><br>Furthermore, by illuminating the inner four LEDs, the gauge shows when the pot hits *exactly* 0.5. This can only happen if you use the **notch** parameter. Otherwise its practically impossible to hit exactly.<br><br>The LED gauge is automatically activated if you use **select**. If you don’t like the LED gauge, you can turn it off with **ledgauge = off**. Otherwise **ledgauge** set’s the color of the indicator in the same way as the **R**-registers do and at the same time *enables* the gauge even if you don’t use **select**.<br><br>Here are some color examples that you can use for the value of **ledgauge**:<br><br>0.2 cyan<br>0.4 green<br>0.6 yellow<br>0.73 orange<br>0.8 red<br>1.0 magenta<br>1.1 violet<br>1.2 blue |
| **startvalue** |  | ☞ | This parameter only makes sense if you work with the **select** input in order to create overlayed virtual pots. Then the *virtual* pot value will be set to this value when your DROID starts. Also the current value will no longer be persisted to the SD card. Otherwise it starts at **0.5** at the very first time and is than persisted to the SD card. Please also have a look at the **reset** input. |
| **reset** |  | | A trigger here resets the virtual pot value to **0.5** or - if **startvalue** is patched - to that value. This only makes sense if you use **select** for creating overlayed virtual pots. |

| Output | Type | Description |
|---|---|---|
| **output** |  | Your pot output comes here. |

| Output | Type | Description |
|---|---|---|
| **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. |

One **pot** circuit needs **208** bytes of RAM.

### 10.43 `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 here:

```
[quantizer]
    input  = P1.1 * 1V
    output = O1
```

Note[1]: 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.

Note[2]: 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  = O1
```

| 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`** |  | **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 |

One **`quantizer`** circuit needs **92** bytes of RAM.

### 10.44 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 - und 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 = O4
```

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 = O1
    output2 = O2
    output3 = O3
```

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** | 1∘2∘3 | ☞ | 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 defaults 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 1st, 2nd, ... 8th cell. |

One **queue** circuit needs **468** bytes of RAM.

### 10.45 `random` - Random number generator

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
```

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 triggger: 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`**, $\frac{\texttt{minimum}+\texttt{maximum}}{2}$ and **`maximum`** and so on... |

| Output | Type | Description |
|---|---|---|
| **`output`** |  | Output of the random number / voltage |

One **`random`** circuit needs **88** bytes of RAM.

### 10.46 `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      = O1
```

| 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`** | 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.<br><br>Now if you trigger the **`sample`** circuit with the sequencer's gate you will essentially sample the *previous* pitch CV instead of the new one. Or maybe something in between.<br><br>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.<br><br>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. |
| **`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. |

One **`sample`** circuit needs **108** bytes of RAM.

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 instance 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 = O1
    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 = O1
    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 = O1
    gateoutput  = O2
    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 174) and **minifonion** (page 149) 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 = O1
    gateoutput = O2
    outputscaling = 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 = O1
    gateoutput = O2
    outputscaling = 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. |
| **outputscaling** |  | **1.0** | The output pitch is multiplied by this parameter. |
| **gatelength** |  | ☞ | 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. |
| **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 *used* **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. |
| **gateoutput** |  | The gate output. |

One **sequencer** circuit needs **844** bytes of RAM.

### 10.48 `slew` - Slew limiter

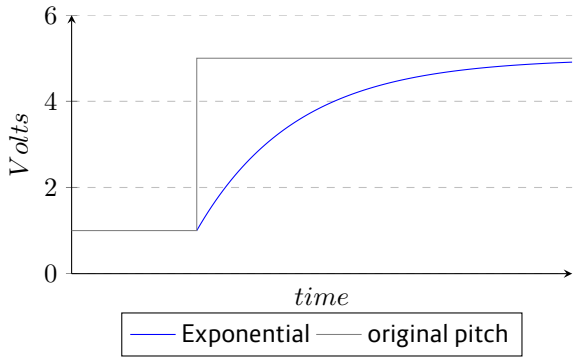
This is a CV controllable slew limiter for CVs. Special about it is that it implements three alternative algorithms. The traditional exponential algorithm (as is commonly implemented in analog circuits), a linear algorithm and a special S-shaped curve.

Here is a simple example for a slew limiting on **`I1`** ⟶ **`O1`** which is controlled with the pot **`P1.1`**:

```
[slew]
    input       = I1
    slew        = P1.1
    exponential = O1
```

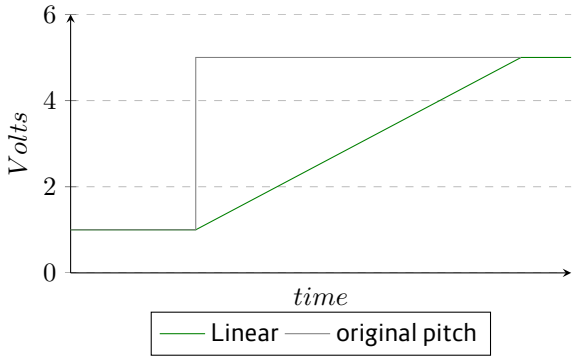
#### Exponential shape

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

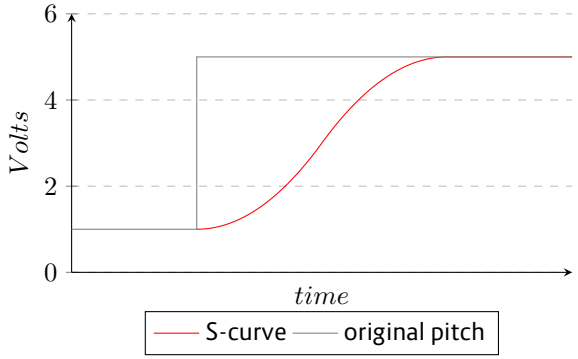
The *linear* algorithm simply limits the voltage change per time to a certain change rate, e.g. to 10 V per second. If the input voltage changes faster (for example suddenly jumps up), the output voltage follows that with that maximum rate. At a pot position of **`0.5`** the maximum slew is 120 V per 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.



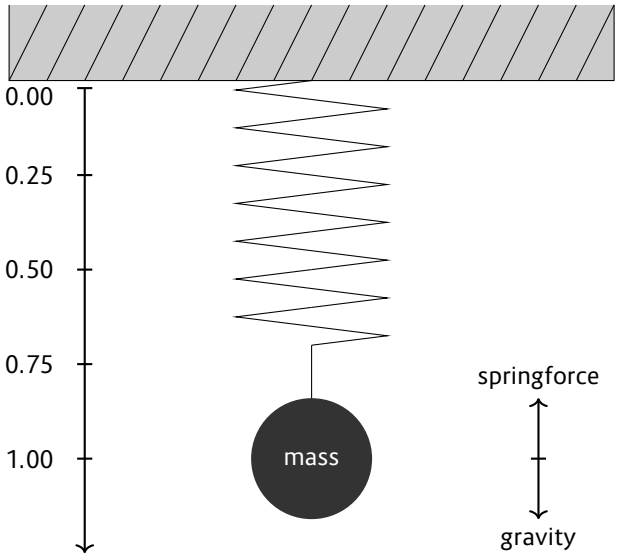
| Input | Type | Default | Description |
|---|---|---|---|
| **input** |  | | Wire the CV that you wish to slew limit here. |
| **slew** |  | **1.0** | This controls the slew rate. A value of **0.0** disables slew limiting. The output immediately follows the input without any delay. A value of for example **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 *upwards*. 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. |

One **slew** circuit needs **124** bytes of RAM.

### 10.49 `spring` - Physical spring simulation

This circuits implements a physical simulation of a mass hanging from on an ideal spring, like in the following drawing. This can create interesting CV sources.




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 = O1
    velocity = O2
```

Now, this could be done more easily with the LFO circuit (see page 113). 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 = O1
    velocity = O2
```

#### 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 $1\frac{N}{kg}$. The force of the spring is $1\frac{N}{m}$.

In order to avoid anomalies or infinities, the velocity of the mass is limited to $\pm 10\frac{m}{s}$ and the position is limited to the range of $\pm 10\,m$.

| 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** |  | **1.0** | This parameter speeds up (or slows down) the perceived time. It works on a 1V/Oct base. So if you set speed to **2V** or **0.2** it will speed up the movement by 100%. |
| **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. |

One **spring** circuit needs **208** bytes of RAM.

## 10.50 `superjust` - Perfect intonation of up to eight voices

### Introduction

This circuit automatically creates a perfect pure intonation for up to eight input pitches. 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 tempered 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 then 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 = O1
    output2 = O2
    output3 = O3
    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 74). 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 = _O1
    output2 = _O2
    output3 = _O3

[calibrator]
    input = _O1
    output = O1
    nudgeup = B1.1
    nudgedown = B1.3

[calibrator]
    input = _O2
    output = O2
    nudgeup = B1.2
    nudgedown = B1.4

[calibrator]
    input = _O3
    output = O3
    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** | 1V/Oct | ☞ | 1st ... 8th 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** | 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** | 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** | 1V/Oct | 1st ... 8th pitch output |

One **superjust** circuit needs **244** bytes of RAM.

### 10.51 `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 **`O1`** be means of a trigger at **`forward`**. At the beginning **`I1`** is wired to **`O1`**. 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  = O1
    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 = O1
    output2 = O2
    output3 = O3
    output4 = O4
    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 = O1
    output2 = O2
    output3 = O3
    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 **`O1`**) or 1 (10 V) for selecting **`O2`**:

```
[switch]
    input   = I1
    output1 = O1
    output2 = O2
    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 = O1
    output2 = O2
    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 = O1
    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 |
|---|---|---|---|
| **input1 ... input8** |  | **0.0** | 1st ... 8th input |
| **forward** |  | | 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. |
| **backward** |  | | Similar then **forward**, but switches backwards |
| **reset** |  | | Resets the switch to its initial position. Assuming **offset** is at **0**, **input1** is connected to **output1**, **input2** to **output2** etc.<br><br>If **reset** and a trigger at **forward** / **backward** 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. |
| **offset** | 1∘2∘3 | **0** | 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 **forward** 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 **I1** here, you probably want to multiply with some useful number. |

| Output | Type | Description |
|---|---|---|
| **output1 ... output8** |  | 1st ... 8th output |

One **switch** circuit needs **324** bytes of RAM.

## 10.52 `switchedpot` - Overlay pot with multiple functions (OBSOLETE)

**This circuit has been superseded by the new circuit `pot` (see page 169). `pot` can do all `switchedpot` can do and much more. So `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    = O1
```

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** |  | | 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 substracting 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. |

One **switchedpot** circuit needs **244** bytes of RAM.

### 10.53 `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.

| 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. |
| **`timing1 ... timing8`** |  | ☞ | Specifies a *relative* timing for each step in relation to the input clock. A **`timing`** of 0.3 will shift the respective beat 30% of a clock cycle behind, while -0.3 will make it 30% early. |

| Output | Type | Description |
|---|---|---|
| **`output`** |  | Here comes the modified output clock |

One **`timing`** circuit needs **220** bytes of RAM.

## 10.54 `togglebutton` - Create on/off buttons (OBSOLETE)

**This circuit has been superseded by the new circuit `button` (see page 67). `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        = O1
```

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 **`O1`**. 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      = O1
```

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 (resulting 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 *double click mode* when set to **1**. In that mode the button only toggles it's 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 1st ... 4th 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**. |

One **`togglebutton`** circuit needs **136** bytes of RAM.

### 10.55 `transient` - Transient generator

This circuit creates transients. It outputs a voltage that starts at a start value and moves linearly 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 = O1
```

Here the duration is meant to be 600 seconds (10 minutes). So at the beginning **O1** 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 = O1
    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 = O1
```

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). |

One **transient** circuit needs **192** bytes of RAM.

### 10.56 `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 *seconds*. 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 *seconds* - or if **`clock`** is being used - in clock cycles. |
| **`repeats`** |  | **`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 |
|---|---|---|---|
| **clock** |  | | When you patch this input, the trigger delay runs in clocked mode. In this mode **delay** is relative to one clock cycle. I.e. a delay if **0.5** will delay the trigger by half a clock cycle. The same holds for **gatelength**. That is measured in clock cycles, too. |

| Output | Type | Description |
|---|---|---|
| **output** |  | Outputs the delayed triggers/gates, while keeping the gate length - unless you have changed that |
| **overflow** |  | 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. |

One **triggerdelay** circuit needs **356** bytes of RAM.